# SnowSim - Final Report

**Zixi Cai**
zixi_cai@berkeley.edu

**Xiaoyu Yang**
xiaoyu_yang@berkeley.edu

**Chris Powers**
chris.powers@berkeley.edu

## ABSTRACT

Snow is a special material that has both solid- and fluid-like properties, which makes it hard to create realistic and believable snow effects. Large amounts of snow particles take tremendous computational time to simulate, making it expensive to produce high quality scenes. In this project, we build a high-performance snow simulator based on Material Point Method (MPM) [7]. Moreover, we apply CUDA to further accelerate computation and reduce rendering time. We demonstrate the performance with various snow results.

**Author Keywords**
snow simulation, material point, particle simulation with CUDA

## 1. INTRODUCTION

Simulating natural phenomena is an important application that remains challenging. For realistic simulation, a particular representation should be chosen, typically depending on the properties of the physical problem to be solved. Snow simulation is extremely hard because it has both fluid- and solid-like properties. Fluid dynamics are handled using a Cartesian grid[6], where a fluid is represented by the parts/cells of the grid it occupies at a given time step. In contrast, substances like sand are naturally represented in particle form, where each particle has its own status and properties. Snow, however, is a more complex substance. On one hand, it consists of individual grains and flakes of snow, but on the other hand, the particles are tiny so it sometimes behaves like a fluid. To handle the complex properties of snow, we use MPM to effectively simulate believable snow results.

With MPM, we store snow data in both grid and particle form. Information is transformed back and forth between the two forms during simulation. With this method, self-collision and fracture are handled automatically. Moreover, iterating through each particle is unnecessary as we only consider the contribution of neighboring particles to the target particle. Near neighbor search is highly effective with the data structure we build. More details will be discussed in Section 2.

Changing the simulation parameters can drastically affect the style and behavior of the snow. Decreasing Young's modulus and the hardening coefficient means less stress is required to stretch the snow by the same amount, giving a muddy effect. Decreasing the critical compression and critical stretch makes the snow break down earlier, giving a fine-grained, powdery effect. Increasing the coefficient of friction, gives a rigid effect. More discussion is in Section 3.

*CHI'20,* April 25–30, 2020, Honolulu, HI, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-6708-0/20/04...$15.00

DOI: https://doi.org/10.1145/3313831.XXXXXXX

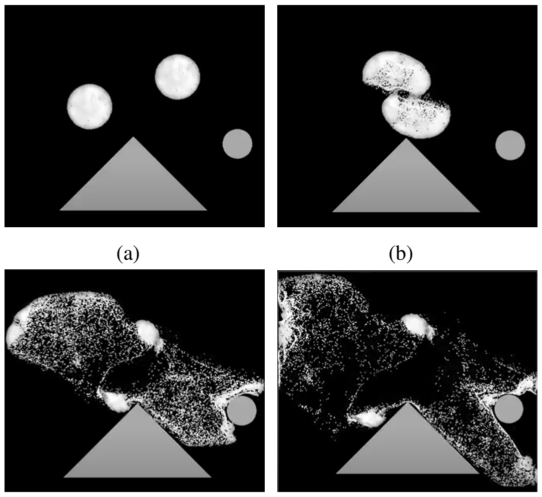


Figure 1: Snowballs colliding with each other and objects. Screenshots of different timesteps are shown.

To further accelerate simulation, we apply CUDA to our algorithm. We utilize the fact that each grid cell and particle is relatively independent to parallelize the computation with CUDA. In each step, each thread takes over only one or a few particles or cells and does the same computation. Atomic operations are used to prevent multiple threads writing to the same unit at the same time. We compared the performance of the CPU and GPU versions on a scene with the same number of particles and time steps. In the collision scene in Figure 1, the CUDA version takes about 1/3 of time of the CPU version. This allows us to simulate complicated scenes in a tolerable amount of time.

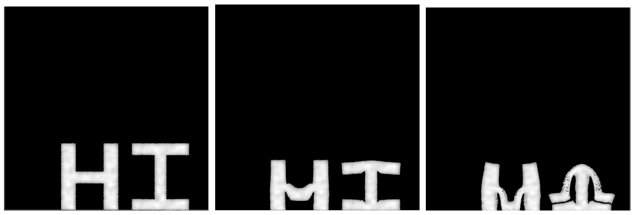
The paper continues with a detailed discussion of the technical approach in Section 2. In Section 3 we present various snow results with different parameters and extended results in 3D scenes.

## 2. TECHNICAL APPROACH

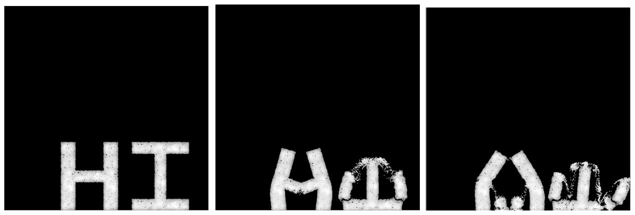
### MPM Implementation

*Overview*

We follow the Material Point Method for snow simulation described in Stomakhin *et al.* [7]. An elasto-plastic constitutive

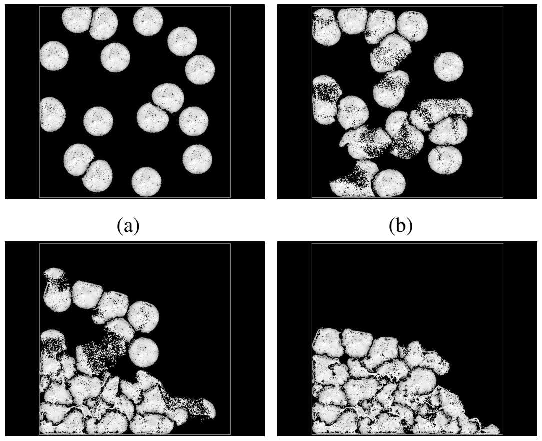


(a) With parameters $E_0 = 10^4$, $\varepsilon_0 = 7.5$, $\theta_C = 10^{-2}$, and $\theta_S = 2.5 \times 10^{-3}$, the snow is slightly powdery, but still far too muddy, and looks like it's melting instead of collapsing.



(b) Interestingly, the solution is not to make the snow more solid by increasing $E_0$ and $\varepsilon_0$, since this prevents the letters from collapsing at all. Instead, drastically increasing to $\theta_C = 0.1$ and $\theta_S = 0.05$ gives the desired crumbly effect.

Figure 2: Letters collapsing



(a) (b) (c) (d)

Figure 3: Simulate 16 snowballs' collision with CUDA. There are 80,000 particles in the scene in total.

model is adopted to model dynamics of snow and a hybrid process combining Lagrangian particles and Cartesian grids is leveraged to update particle positions and velocities at each timestep. Figure 4 shows some visualizations of the particles alongside the active grid cells. The method consists of 9 steps.

**Step 1**: Rasterize particle data to the grid. Given the positions, masses, and velocities of the particles, we transfer these properties to the grid cells.

**Step 2**: Compute particle volumes and densities. In the force computation, volumes of particles are needed. So we deduce the volume of each particle from its density, which is derived from the densities of surrounding grid cells. This step is only executed once during the first timestep.

**Step 3**: Compute grid forces. The force on each grid cell is computed using the volumes and deformation gradients of neighboring particles.

**Step 4**: Update velocities on the grid. Given the force on each cell, acceleration as well as change in velocity can be calculated using Newton's Second Law. Note that we apply gravity here by adding it to the stress-based force when calculating acceleration, which is not described in Stomakhin *et al.* [7].

**Step 5**: Grid-based body collisions. When a grid cell is found to be inside an object and has relative velocity component towards the object, we handle the collision by modifying the velocity. Specifically, we zero out the normal component of the relative velocity and apply a friction-based decay to the tangent component. Note that in Stomakhin *et al.* [7], they use level sets to represent objects and leverage algebraic methods to determine whether a point is inside an object as well as the normal at that point. In our project, since we only consider simple geometries such as planes and spheres, geometrical methods are used to make these judgements.

**Step 6**: Update deformation gradients. The elastic and plastic parts of the deformation gradients of particles are updated given the grid velocities. We use the Eigen library to compute the SVD in the computation.

**Step 7**: Update particle velocities. Particle velocities are updated as a weighted sum of PIC velocities and FLIP velocities.

**Step 8**: Particle-based body collisions. The same routine as step 5 is applied to update particle velocities due to collision. One thing we do, which is not done in Stomakhin *et al.* [7], is that besides updating particle velocities, we also update particle positions by pushing them out of the objects they were inside. We do this because chances are that the velocity change alone would fail to drag the particle out of the object, leaving it stuck inside the object where it could not interact with other particles anymore.

**Step 9**: Update particle positions. Particles are moved according to their computed new velocities.

We do not implement semi-implicit integration since it is complicated and time-consuming. Additionally we can eliminate the impact of explicit integration's instability by choosing small timesteps.

### *Data Structure for Near Neighbor Search*

One of the most important tricks we take advantage of to accelerate simulation is using a special data structure to save time for near neighbor search. Many steps involve particle-grid interaction, in which we need to iterate over all particles/cells and for each particle/cell, we need to take its surrounding particles/cells into consideration. Take step 3 as an example. For a given grid cell, only particles that live in cells that are no more than 2 cells away have a non-zero contribution to the force, so we do not need to visit particles outside of those ones.

To take advantage of this property, we need a data structure that can efficiently locate particles in a grid provided the grid location. Here we use a hash map, in which keys are the

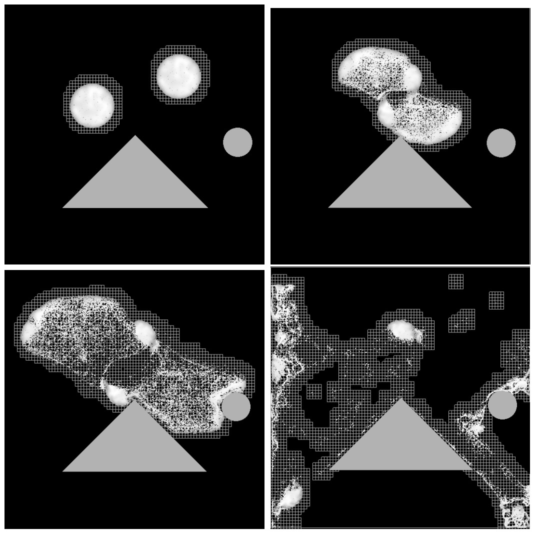


Figure 4: Visualization of snow particles and the active grid cells. Some grid cells are active despite not containing any snow particles because the algorithm also considers nearby particles.

discrete positions of cells and values are arrays of particles that live in the corresponding cells.

We also use a hash map to store grid information, with keys storing the discrete cell locations and values storing the information of each cell. We use a hash map instead of a simple array or matrix because most grid cells are inactive and will not make any contributions. Storing only the active cells takes much less memory.

## CUDA Acceleration

We further accelerate simulation by using CUDA. For many steps of the algorithm, we either update particle information according to surrounding grid information or update grid information according to surrounding particle information. This means that the computation for each particle/grid cell is relatively independent during each step. This allows us to efficiently parallelize computation for each particle/cell. Moreover, computation for each particle/cell is highly similar. This is where CUDA, which is good at handling a large amount of independent and similar work, comes in.

The CUDA version is not much different from the CPU version, except that each thread only handles one or a few particles or cells. That is to say, we only need to modify the original `for` loop so that thread $i$ iterate over particles or cells with index $i$, $i+K$, $i+2K$, ..., where $K$ represents total number of threads launched.

To take full advantage of the power of CUDA, we need to rearrange the computations in step 1 and step 3, which update grid information according to particle information. The most straightforward approach is to iterate over all active cells and for each cell, iterate over all surrounding particles. Unfortunately, this method gives at most get $M$ times speedup, where $M$ is the number of active cells, which is very small compared to the number of particles. Thus, in order to saturate the usable threads, we instead iterate over particles, and for each particle, we update its surrounding cells.

This brings a new problem. Multiple threads handling different particles may try to update the same cell at the same time, which will cause a race condition and make the final results uncertain. To prevent this, we apply the atomic operation `atomicAdd`. In step 1, we use it to update masses and velocities of cells. In step 3, we use it to update grid forces.

Another issue comes with SVD. It is a lot of trouble to use the Eigen library in CUDA. To bypass this, we exploit the fact that we are handling 2 by 2 matrices and use the explicit formula the SVD of a 2 by 2 matrix described in Blinn *et al.* [3]. For a matrix $M = \begin{bmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{bmatrix}$, its SVD $M = U\Sigma V^\top$ is given by

$$E = \frac{m_{00}+m_{11}}{2} \qquad F = \frac{m_{00}-m_{11}}{2}$$
$$G = \frac{m_{10}+m_{01}}{2} \qquad H = \frac{m_{10}-m_{01}}{2}$$
$$Q = \sqrt{E^2+H^2} \qquad R = \sqrt{F^2+G^2}$$
$$\theta = \frac{1}{2}[\text{atan2}(H,E) - \text{atan2}(G,F)]$$
$$\phi = \frac{1}{2}[\text{atan2}(H,E) + \text{atan2}(G,F)]$$
$$U = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} \qquad V = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$
$$\Sigma = \begin{bmatrix} Q+R & \\ & Q-R \end{bmatrix}$$

## Test Framework and Video Creation

We created a test framework, so that we can focus separately on the implementation of the algorithm and the visualization of the results. The finished algorithm can be integrated into this framework to create a demo, via which we check the algorithm's correctness.

The test framework consists of a visualizer, a data generator, and a simulation launcher. The data generator generates initial data in the form of a series of particles. The simulation launcher takes in the data and run the simulation loop. For each time step of the loop, it calls the *simulate_one_step* function, which updates the information of the particles. The visualizer then renders these particles at each time step accordingly. Provided with this test framework, we are able to fully concentrate on the implementation of the *simulate_one_step* function.

The visualizer is based on CGL. It is responsible for transforming coordinates from the world space, in which we do simulation, into screen space. Given particle positions, the visualizer draws them as points. The intensity of each particle is set based on its density, which is computed in step 2. The denser a particle is, the high value of intensity we assign to it.

Since the simulation is much slower than reality, we want to speed it up in order to determine whether or not the dynamics look correct. To do this, we invented a playback mechanism.

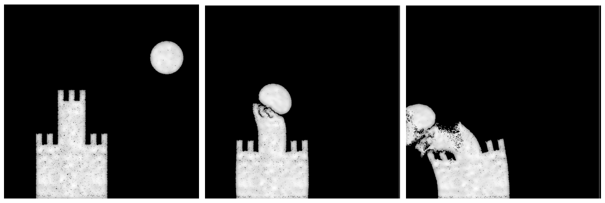


Figure 5: Visualization of a snowball crashing into a snow castle. The castle was constructed by sampling from multiple rectangular regions. The number of samples in each region is proportional to its area, making the density constant.

First we separated simulation and visualization: we can simulate and save all particle information for each step to a file, and after simulation we can replay the whole process by reading from the file. This makes motion look faster and more natural. Additionally, we can adjust the speed of playback by specifying $S$, which means we display 1 frame for every $S$ ticks of the simulation. This playback mechanism also lets us easily record videos of our results.

One problem we faced is that the visualization described above does not work for the CUDA version, because we rely on a remote server to use GPU, which makes CGL that requires X11 impossible. Therefore we adopt a totally different visualization scheme for the CUDA version. We leverage OpenCV to render the scene to images that will be saved to files, and we can later create a video by concatenating these images given a certain FPS. Note that we don't render and save every simulation step, because we do not need so many frames to create a video, and more importantly, it takes time to transfer data from GPU to CPU.

As for the data generating scheme, we specify the shape, the size and the position of the snow chunk we want to create. The generator draws a certain number of samples uniformly in the area, which are used as the positions for the particles. We assign the mass of each particle as a random value around a mean mass of a particle. The mean mass $m$ of a particle is computed according to the density $\rho$, the area $A$ and the number of samples $n$ as $m = \frac{\rho A}{n}$. An example of our data generation is in Figure 5, where a castle was constructed by sampling from multiple rectangles, and a snowball with some initial velocity was constructed by sampling from a circle.

### 3D Extension

We also try to extend our algorithm as well as the visualizer to 3D, shown in Figure 6. The algorithm itself is similar to 2D version. As for visualization, we apply the diffusion component of the Blinn-Phong shading model [4] to surfaces of objects to make scenes look cooler.

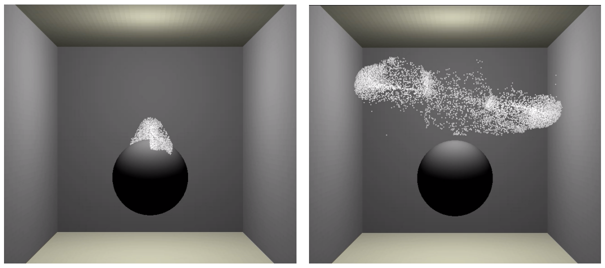

(a) Snowball falling on an object (b) Snowballs colliding

Figure 6: 3D simulation and visualization for two scenes.

## 3. RESULTS

### Parameter Search

As mentioned in the introduction, we test the effect of varying certain simulation parameters, to see if we can reproduce the effects that would be expected based on physics and the simulation code. For these tests, we use a scene of a snowball rolling down two ramps. The resulting images can be seen in Figure 7. The default parameters, the same as the defaults from [7], can be seen in Table 1.

| Parameter Name | Notation | Default Value |
|---|---|---|
| Hardening coefficient | $\varepsilon_0$ | 10 |
| Young's modulus | $E_0$ | $1.4 \times 10^5$ |
| Critical stretch | $\theta_S$ | $7.5 \times 10^{-3}$ |
| Critical compression | $\theta_C$ | $2.5 \times 10^{-2}$ |
| Coefficient of friction | $\mu$ | 0.1 |

Table 1: Default simulation parameters

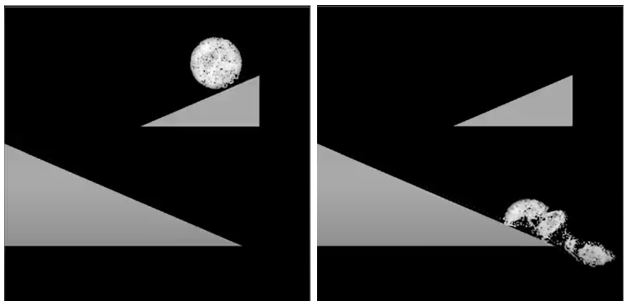
First, we increase Young's modulus and the hardening coefficient. From a physical interpretation [2], Young's modulus is $E_0 = \frac{\sigma}{\varepsilon}$, the ratio of stress to strain. Thus increasing $E_0$ will decrease the amount of strain, or deformation, generated per unit stress, or force, applied. Increasing the hardening coefficient $\varepsilon_0$ should have a similar effect. This matches our expectations from how these parameters are used in the simulation code. Namely, the initial Lamé parameters [1] are:

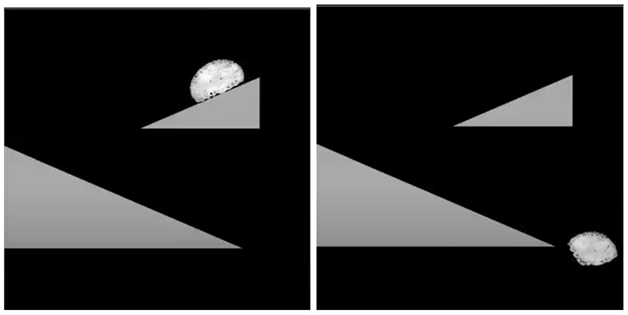
$$\mu_0 = \frac{E_0}{2(1+\nu)}$$
$$\lambda_0 = \frac{E_0 \lambda}{(1+\nu)(1-2\nu)}$$

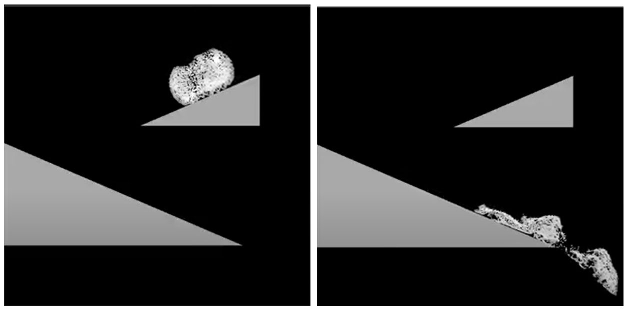
where $\nu$ is the Poisson ratio, a parameter we keep constant. The takeaway is that $\mu_0$ and $\lambda_0$ increase proportionally to Young's modulus. In step 3 of the algorithm, these values are used to compute the per-particle stress $\sigma$ according to the formulae from [8] and [5]:

$$J_E = \det \mathbf{F_E} \qquad J_p = \det \mathbf{F_p}$$
$$\mu = \mu_0 e^{\varepsilon_0(1-J_p)} \qquad \lambda = \lambda_0 e^{\varepsilon_0(1-J_p)}$$
$$\sigma = \frac{2\mu}{J_p}(\mathbf{F_E} - \mathbf{R_E})\mathbf{F_E}^T + \frac{\lambda}{J_p}(J_E - 1)J_E \mathbf{I}$$

where $\mathbf{F_E}$ and $\mathbf{F_P}$ are the elastic and plastic components of the deformation gradient. This shows that the ratio of stress to deformation is proportional to $E_0 e^{\varepsilon_0(1-J_p)}$, which predicts that increasing $E_0$ and $\varepsilon_0$ should decrease the deformation for a given amount of stress. Our experiment confirms that with an increased $E_0$ and $\varepsilon_0$, the snowball becomes more solid and icy because it is less prone to deformation. Decreasing Young's


(a) Icy Snow, $E_0 = 5 \times 10^5$ and $\varepsilon = 20$


(b) Muddy Snow, $E_0 = 5 \times 10^3$ and $\varepsilon = 5$


(c) Powdery Snow, $\theta_C = 10^{-2}$ and $\theta_S = 2.5 \times 10^{-3}$

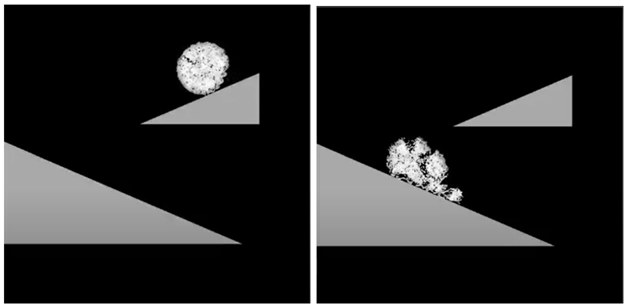

(d) High Friction, $\mu = 0.5$

Figure 7: Effects of Different Parameters

modulus and the hardening coefficient gives the reverse effect of muddy snow.

Next, we decrease the critical compression and critical stretch parameters. According to [7], this should make the snow powdery instead of chunky. In the simulation code, they are used to clamp the singular values $\sigma$ of $\mathbf{F_E}$ to the range $[1-\theta_C, 1+\theta_S]$. The singular values of $\mathbf{F_P}$ scale with $\frac{1}{\sigma}$. Thus, decreasing these parameters should bring the singular values closer to 1. Then the determinants $J_P$ and $J_E$ will also get closer to 1, since the determinant is the product of the singular values. This is still hard to interpret physically, but we can at least see how this change in parameters will affect the stress computation. Our experiments confirm that decreasing $\theta_C$ and $\theta_S$ gives us smaller, powdery snow particles.

Lastly, we increase the coefficient of friction $\mu$. The physical meaning of this is simply increasing the frictional force that prevents the snow from moving as much. In the simulation code, this parameter is used at the grid and particle level to update velocity after a collision. As expected, this slowed the snow down, preventing it from rolling all the way down both ramps. It also has the unexpected affect of making the snow roll much more during its descent, instead of simply sliding down the ramp.

Overall, even slightly changing the parameters has a large impact on the behavior of the snow, which demonstrates the wide range of capabilities of this algorithm. Just by changing a few coefficients, we can go from solid ice, to a loose collection of particles, to a slushy blob. The two experiments with snow in the shape of letters in Figure 8 show how creativity can come into play in tuning parameters to get your desired effect. It takes some time to strike the right balance between the two extremes of highly deformable snow, which doesn't retain the shape of the letters at all, and very stiff snow, which doesn't collapse, to get snow that slowly collapses and crumbles.

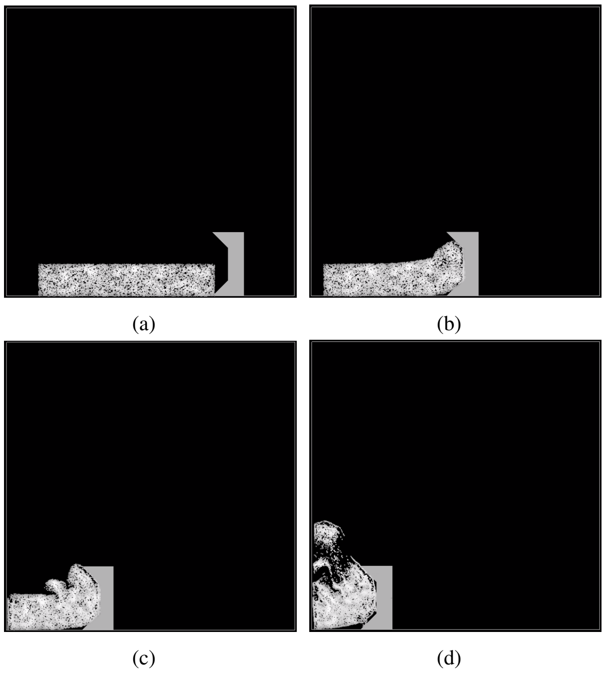

(a) (b) (c) (d)

Figure 8: Interacting with a "snowplow"

### CUDA speedup

We compare the time cost of rendering the same scene using the CPU version and the CUDA version. We experiment on a "two-snowball collision" scene (Figure 1). The total number of particles is 10,000, and we run 10,000 steps which equals to 2 seconds. The CPU version takes 1289 seconds and the CUDA version takes 411 seconds, which is roughly 1/3 of the time the CUDA version takes.

## 4. CONTRIBUTIONS

**Zixi Cai** Built up the test framework and the 2D visualizer; debugged and optimized the algorithm; extended the algorithm to CUDA version.

**Xiaoyu Yang** Finished step 5, 6 and 8 of the algorithm; extended the algorithm to 3D; built up the 3D visualizer.

**Chris Powers** Finished step 1, 2, 3, 4, 7 and 9 of the algorithm; invented the playback mechanism; created lots of interesting scenes.

## 5. REFERENCES

[1] 2017. Lame constants. (2017). `https://www.encyclopediaofmath.org/index.php/Lamé_constants`.

[2] 2020. Young's Modulus. (2020). `https://en.wikipedia.org/wiki/Young%27s_modulus`.

[3] Jim Blinn. 1996. Consider the lowly 2 x 2 matrix. *IEEE Computer Graphics and Applications* 16, 2 (1996), 82–88.

[4] James F Blinn. 1977. Models of light reflection for computer synthesized pictures. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*. 192–198.

[5] Thomas Breekveldt. 2017. Analysis of MPM for Snow. (2017).

[6] Chenfanfu Jiang, Craig Schroeder, Andrew Selle, Joseph Teran, and Alexey Stomakhin. 2015. The affine particle-in-cell method. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 1–10.

[7] Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. 2013a. A material point method for snow simulation. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 1–10.

[8] Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. 2013b. *Material Point Method for Snow Simulation*. Technical Report. 2 pages.