WebGL based Real-time 3D Snow Simulation

Fan Zhang*†
EECS Department
University of California, Berkeley
fan.zhang@berkeley.edu

Yu-chieh Lan*†
EECS Department
University of California, Berkeley
yclan2@berkeley.edu

Zhaoxiong Cui*†
EECS Department
University of California, Berkeley
zhaoxiong.cui@berkeley.edu

ABSTRACT

Snow simulation is a challenging task in computer graphics since it has both the solid-like and fluid-like properties. Based on the Material Point Method(MPM) snow simulation work published by Disney in 2013, we transfer it to the WebGL with Three.js which allows us to render the snow in the browser. We have developed a vectorized 3D MPM framework in WebGL which can handle both 2D and 3D MPM simulation. Also, we designed a customized shader to render the snow particles and the related user interface in the webpage to interact with the simulation. We rendered it in real time and the result demonstrated that MPM can well define the physical dynamics properties of snow.

1 INTRODUCTION

Snow is a common natural phenomenon in the real world which covers the ground all the white in the winter. However, in computer graphics, it is a challenging task to model the snow digitally. Different from other weather phenomena, snow has both the solid-like and fluid-like properties which are difficult to handle. In the meantime, the dynamic scenes of snow can be divided into three parts, namely, snow falling, snow accumulation, snow wind interaction. What we need is not only to render a unique texture of snow, but also to recover the behavior of snow particles. Since snow is composed of small ice particles, it is a granular material, so we can representation is with the particle system [Tan and Fan 2011]. Luckily, researchers have figured out the way to model the snow as a particle system [Stomakhin et al. 2013]. Since snow has continuously varying phase effects, sometimes behaving as a rigid/deforming solid and sometimes behaving as a fluid. Thus, instead of discrete coupling we must simultaneously handle a continuum of material properties efficiently in the same domain, even though such a solver may not be most efficient for a single discrete phenomenon.

The most famous work for snow simulation [Stomakhin et al. 2013] developed a semi-implicit Material Point Method(MPM)[Sulsky et al. 1995] in order to efficiently treat the wide range of material stiffnesses, collisions and topological changes arising in complex snow scenes. MPM methods combine the Lagrangian material particles with Eulerian Cartesian grids. Based on it, our work implemented the MPM with WebGL to render the snow interaction effect in the browser.

Currently, most of the computer graphics works are implemented in C++ with OpenGL since it supports GPU. However, to render a demo in OpenGL, we need to download the source code, compile it and then render it, which is complicated. Inspired by a really cool WebGL demo, we wanted to implement the snow simulation based on WebGL [Parisi 2012]. Our goal is to construct snowy scenes and accomplish the animation simulation in real-time. The scenes could include different shapes of snow on different terrains. Because it is a web page, we should also include a user interface that enables users to design and interact with the scene. Tuning bars could also be provided on the interface to adjust some parameters within reasonable ranges. Real-time interaction may be an interesting function, users can catch the falling snow by dragging different containers.

Overall, our work was developed mostly with *Three.js*[Danchilla 2012] which is a 3D library in WebGL. The figure 1 shows the basic code structure of our work. Among them, the most important three important parts are:

- MPM implementation on WebGL
- Shader for snow particles
- User Interface of webpage

We will introduce these elements in detail in the following sections.

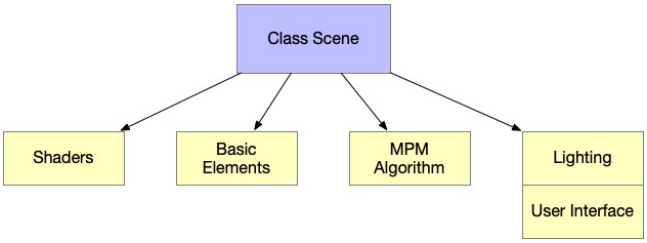

Figure 1: Javascript code structure of our work

2 RELATED WORK

In this section, we will briefly introduce the research related to snow simulation.

Snow Simulation. We mentioned in Section1 that snow has mainly three properties. For the snow accumulation, [Feldman and O'Brien 2002; Zhu and Yang 2010] can efficiently and accurately create snow-covering effects and let snow interact with external objects. [Kim et al. 2006] also succeeded in simulating the related phenomena of ice and frost formation. For snow dynamics simulation, the snow system is alwared regarded as a granular material system and some work from granular material somulation [McAdams et al. 2009] , which was not used to simulate the snow at first, was also introduced to the snow simulation field. In the meantime, there is extensive engineering literature related to the modeling and simulation of snow. Researchers found an elasto-plastic constitutive relation worked well for generating realistic dynamics for a wide range of visual phenomena. This representation, as well as finite-element-based discretization, is very common in the engineering literature[Cresseri et al. 2010; Dutykh et al. 2011].

*All authors contributed equally to this research.
†The github repo of this work is https://github.com/Czxck001/webgl-snow

3 MATERIAL POINT METHOD

A body's deformation can be described as a mapping from its undeformed configuration X to its deformed configuration $\boldsymbol{x}$ by $\boldsymbol{x} = \boldsymbol{\phi}(\boldsymbol{X})$, which yields that deformation gradient $\boldsymbol{F} = \partial\boldsymbol{\phi}/\partial\boldsymbol{X}$. The deformation $\phi(\boldsymbol{X})$ changes according to consercation of mass, momentum and the elasto-plastic constitutive relation:

$$\frac{D\rho}{Dt} = 0, \quad \rho\frac{Dv}{Dt} = \nabla\cdot\sigma + \rho g, \quad \sigma = \frac{1}{J}\frac{\partial\Psi}{\partial\boldsymbol{F}_E}\boldsymbol{F}_E^T \tag{1}$$

In the equation 1, ρ is density, t is time, $\sqsubseteq$ is velocity, σ is the Cauchy stress, g is the gravity, Ψ is the elasto-plastic potential energy density. $\boldsymbol{F}_E$ is the the elastic part of the deformation gradient $\boldsymbol{F}$ and $J = \det(\boldsymbol{F})$.

The idea of material point method is to combine the Lagrangian method and Euler method together to deal with snow elements. Each snow is regarded as a particle that has properties of position $\boldsymbol{x}_p$, velocity $\boldsymbol{v}_p$, mass m_p, and the deformation gradient $\boldsymbol{F}_p$. With the Lagrangian treatment of these quantities, we can discretize the $\frac{D\rho}{Dt}$ and $\frac{Dv}{Dt}$ from equation 1. However, only with it can't resolve the interactions between particles, which is the most important part in physical simulation. The lack of mesh connectivity between particles complicates the computation of derivatives needed for stress-based force evaluation. To address this, the regular background Eulerian grid is used to interpolate discretize the $\nabla\cdot\sigma$. The equation 2 proposed by [Steffen et al. 2008] is used as the grid basis functions.

$$N_i^h\left(\boldsymbol{x}_p\right) = N\left(\frac{1}{h}\left(x_p - ih\right)\right)N\left(\frac{1}{h}\left(y_p - jh\right)\right)N\left(\frac{1}{h}\left(z_p - kh\right)\right) \tag{2}$$

In the equation 2, $\boldsymbol{i} = (i, j, k)$ is the grid index, $\boldsymbol{x}_p = \left(x_p, y_p, z_p\right)$ is the evaluation position, h is the grid spacing. Different from the work from [Stomakhin et al. 2013] who used the cubic kernel, in this work we used the quadratic kernel[Jiang et al. 2016]:

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

For more compact notation, we will use $w_{ip} = N_i^h\left(x_p\right)$ and $\nabla w_{ip} = \nabla N_i^h\left(\boldsymbol{x}_p\right)$.

The full update procedure for MPM with snow particles is listed as:

(1) **Rasterize the particles to the grid.** Since every particle has its own mass and velocity properties, we need to transfer them into the appropriate grid with weightings. The above equation 2 is used to calculate the weighted mass and velocity of each grid with the following equations:

$$m_i^n = \sum_p m_p w_{ip}^n \tag{4}$$

$$\boldsymbol{v}_i^n = \sum_p \boldsymbol{v}_p^n m_p w_{ip}^n / m_i^n \tag{5}$$

(2) **Compute the grid forces.** Using the following equation to compute the force on grid node i resulting from elastic stresses with the form of Cauchy stress.

$$\boldsymbol{f}_i(\hat{\boldsymbol{x}}) = -\sum_p V_p^n \boldsymbol{\sigma}_p \nabla w_{ip}^n \tag{6}$$

(3) **Update velocities on grid.** With the following equation:

$$\boldsymbol{v}_i^\star = \boldsymbol{v}_i^n + \Delta t m_i^{-1} \boldsymbol{f}_i^n \tag{7}$$

(4) **Grid-based Calculation.** Using rigid and deforming collsion types to calculate the body collisions and solve the linear system to let $\boldsymbol{v}_i^{n+1} = \boldsymbol{v}_i^\star$. Here we didn't use the semi-implicit integration, instead, we used the explicit time integration.

(5) **Update deformation gradient.** The deformation gradient for each particle is updated as:

$$\boldsymbol{F}_p^{n+1} = \left(\boldsymbol{I} + \Delta t \nabla \boldsymbol{v}_p^{n+1}\right)\boldsymbol{F}_p^n \tag{8}$$

Here we used $\nabla \mathbf{v}_p^{n+1} = \mathbf{C}_p^{n+1}$ to calculate the ∇v_p^{n+1}.

(6) **Update particle velocities.**

$$\boldsymbol{v}_p^{n+1} = (1-\alpha)\boldsymbol{v}_{\mathrm{PICp}}^{n+1} + \alpha\boldsymbol{v}_{\mathrm{FLPp}}^{n+1} \tag{9}$$

where

$$\boldsymbol{v}_{\mathrm{PIC}p}^{n+1} = \sum_i \boldsymbol{v}_i^{n+1} w_{ip}^n \tag{10}$$

$$\boldsymbol{v}_{\mathrm{FLIP}p}^{n+1} = \boldsymbol{v}_p^n + \sum_i \left(\boldsymbol{v}_i^{n+1} - \boldsymbol{v}_i^n\right) w_{ip}^n \tag{11}$$

(7) **Update particle positions**

$$\boldsymbol{x}_p^{n+1} = \boldsymbol{x}_p^n + \Delta t \boldsymbol{v}_p^{n+1} \tag{12}$$

Most of our code is based on the 88-Line version of High-Performance MLS-MPM [Hu et al. 2018]

4 WEBGL IMPLEMENTATION

4.1 MPM in Javascript

There are a bunch of existing open-source 3D MPM implementations for snow simulation, such as [Isaac 2016; Joel 2015]. However, all of these implementation is in C++. There also exists a Javascript MPM implementation called mls-mpm.js [Roberto 2019], but it's a 2D implementation which means the location and velocity is restricted in 2D space.

The goal of our project is to have a 3D implementation of MPM algorithm in Javascript and WebGL, which will enable it running within a browser. To achieve this goal, we have two options with regard to engineering. The first option is to translate the 3D C++ implementation into a pure-Javascript one. The second option is to extend the 2D Javascript implementation so that it can work with particles in 3D space.

We first put collective efforts into the first option, but unfortunately we failed. The reason of this failure is two-fold. First, C++ language is essentially different from Javascript. C++ is compiled, strong-typed, static-typed and aiming to eliminate all redundant operations, while Javascript is interpreted, dynamic and weak typed and driven by a event loop engine provided by the browser. The rigid features of C++ make the existing MPM code written in C++

flooding with custom data-types (with overloaded operators) and a huge amount of boiler-plates for abstraction. The code is hard to read and understood. It's also hard to rewrite the idiomatic C++ code into Javascript in terms of line-by-line remapping.

The second reason of the failure is that there is no mature approach to embed the C++ code and Javascript code into each other. This makes it extremely risky to translate and refactor the code. If we can embed a line of Javascript code into the existing C++ codebase in substitution of a line of original C++ code, we can then re-run the whole program (or unit-tests if we have) to make sure this substitution is correct. Thus, we can gradually move-on and make sure all our previous move is correct and solid, until we substitute all the C++ code with Javascript code. However, since the embedding mechanism doesn't exist, we have to write the Javascript code completely starting from scratch. If we were done but the code was not working, we will have to do the debugging all over again.

After we realized the reasons of the failure, we decided to try with option two, i.e. to extend the 2D Javascript implementation and make it support 3D particles. This approach overcomes the two disadvantage in C++ translation approach. First, the source reference code and target code are both written in Javascript, which means there is literally no translation cost. Second, if we need to substitute a line of code to a newer implementation, we can simply comment out the code and then plug in the new code. This makes all of the refactor and rewrite seamless.

4.1.1 Vectorization. Still, the 2D to 3D extension is also challenging regarding to two aspects. The first aspect is, although originally written in Javascript, the original mls-mpm.js implementation is supported by a set of handwritten 2D low-level operations. Such 2D operations are essentially Javascript functions accepting only 2D vectors (or 2-by-2 matrices). The top level code that directly implements MPM is the caller of these low-level helper functions.

Calling these 2D-only helper functions results in a issue that we can't simply switch the input of MPM from 2D particles to 3D ones. A better approach would be implementing the MPM using helper functions that can work for both 2D vectors and 3D vectors. We need to first rewrite the 2D implementation in a such way. We call this procedure the vectorization because this will make each basic operation work for input with arbitrary dimensionality.

This vectorization is done with the help of Math.js. Math.js is a Javascript numerical computing library that supports vectorized operations as long as the inputs of such operations match with each other. The whole rewritting processed is done in a safe and stepping way. We rewrite a line of code in original MPM implementation in a equivalent but vectorized way. Then we rerun the whole program again, make sure it works well. We do this procedure line by line until the whole top level module is vectorized.

Next, based on this vectorized transcription, we switch the input from 2D particles to 3D particles. Fantastically, most lines are working well except for some minor issues. After these minor issues are fixed, a 3D pure-Javascript MPM implementation is achieved.

4.1.2 Adaptation. Switching from 2D to 3D setting, some physical constants may no longer be appropriate. Some of them are modified. The Young's Module is changed from 10000 to 10 in order to stabilize the simulation. The time step is enlarged from 0.0001 to 0.001 in order to accelerate the simulation.

4.2 Shader

Since WebGL uses the same shading pipeline with OpenGL, we can use *GLSL* format to write our customized shaders to render snow. In *Three.js*, the default shading material is *MeshBasicMaterial* and we need to replace it to our own shader material with *ShaderMaterial*. A ShaderMaterial has some basic parameters: vertexShader, fragmentShader and uniforms.

- vertexShader: the GLSL code for the vertex manipulation
- fragmentShader: the GLSL code for the fragment manipulation
- unifoms: a list of variables that are shared by both the vertex and the fragment shader

Spheres are the elementary unit of each snow particle. Since in the real world the snow particle should be abnormal, we used the displacement shading to simulate it. The main idea here is disturbing each vertex along the direction of its normal. Imagine that there are lines that go from the center of our sphere to each vertex, on line per vertex. Initally, all those lines are the same length (the radius of the sphere). If we make some longer, and some shorter, we made the disturbed mesh. The figure 2 shows the relation between old and new positions.

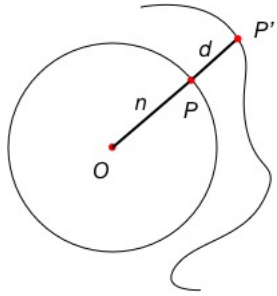

Figure 2: The idea of making sphere abnormal with vertex shading

Equation 13 shows how to calculate the new position of each vertex. Here O is the origin of the sphere, $\vec{p}$ is the original vertex point, $\hat{n}$ is the vertex normal, d is the noise distortion and $\vec{p}'$ is the new vertex point. we want to multiply the normal by some scalar factor so it scales (the line from the center to the vertex shrinks or grows, and since it's defining the vertex position, the vertex itself moves inwards or outwards). That's where we get a noise value. The coordinates for the noise are based on the normal before being modified, and the noise value is modulated to fit the desired scale.

$$\vec{p}' = \vec{p} + \hat{n}d \tag{13}$$

Here we added a random noise to calculate the new position of each vertex. However, arbitrarily choosing a random number is chaotic and not very appealing. We wanted the disturbance to be biased on some random but controllable function. Specifically, we used the **Perlin Noise**[Perlin 2002] to calculate the noise distortion. The perlin noise is a procedural texture primitive, a type of gradient noise which is widely used to increase the appearance of realism in computer graphics. Also, we added an additional distortion based on a larger noise(a low frequency noise) to disturb the sphere shape.

When working with the noise functions, we need to guarantee that the mesh won't change it's shape abruptly every frame. This is achieved by using some value that is the same every frame for both vertex and fragment shaders. Here we used the UV coordinates, the position or the normal. Also, we stored the noise calculated in vertex shader as a fake ambient occlusion factor which was used to highlight raised regions against sunken regions in fragment shader. Figure 3 shows the effect of vertex shader and fragment shader.

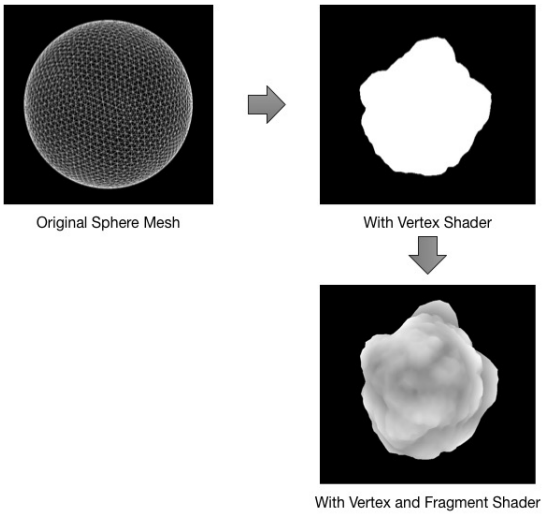

Figure 3: The shader of a snow particle

4.3 User Interface

We used an graphics user interface provided by `dat.gui` package. We implemented five buttons for custom adjustments - **clear**, **start**, **model**, **N** and **p_size**. The panel of our GUI is shown in Figure **??**. The start button triggers the initialization of particles (`init_snow` method), the initialization is according to the last three parameters listed in the UI panel. In order to deal with the error that caused by user's multiple clicks on the start button, we introduced a variable - `initialized`; the variable is set to true after the `init_snow` method is called and false after snow is cleared. The variable is checked before generating new snow particles to ensure that the scene will not initialized several times.

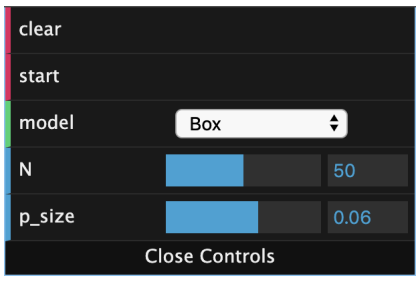

Figure 4: A snapshot of our GUI panel.

The parameters **model**, **N** and **p_size** control how the new snow particles are generated and what they look in the initialization. They would only be effective in the next generation of snow particles. We provide three options for the model parameter, which denotes what model to load in for the snow generation - Box, Heart and Uncracked_Egg. The Box option doesn't load in model but constrain the randomly generated particles in two boxes, and the other two original models are shown in Figure **??**. We put a snow particle on each vertex of the model's mesh to create snow in the shape of the model. We observed that the loaded model have duplicate vertices, which did not influence the visual effect but profoundly slow down the computation because of the superfluous particles. We deal with the problem by adding a hashmap; the 3 dimensional position is mapped to a single float number and the float number is used as a key for the hashmap. The index of the vector is stored in the value field for random points generation.

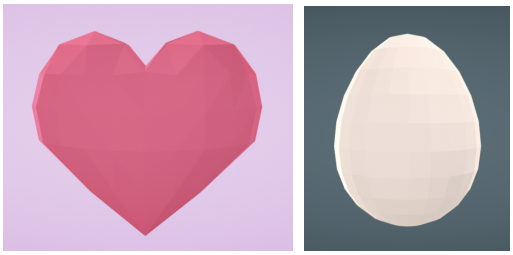

Figure 5: The models we used for shaping snow. (a) Heart. (b) Egg. Created by Poly by Google, via Poly.

The parameter **N** controls the number of points being randomly generated. For the box scene doesn't load a model as mentioned, but here we generate random points inside two boxes. The boxes are halfly overlapped in the x-axis, contingent to each other in y-axis, and fully overlap in z-axis. For the Heart and Uncracked_Egg scene, we randomly choose two points among the filtered vertices (without duplicates) and place a random point between them by randomly weighting the two points. We observed some particle points lie outside of the model shape but we do not figure out how to fix this bug. The p_size parameter simply controls the size of the particles. It ranges from 0.001 to 0.1. Finally, the **clear** button clears out the *SnowParticle* instances stored in the *SnowGroup* instances and also clears out the grid in *MPMGrid* instance. These actions clear all the snow particles that present in our scene and get the scene ready for another start of particle generation.

5 RESULT

Since we implemented the MPM without any parallelism and we want to have a real-time effect, we can only render a maximum number of 100 particles with our own computer.

Figure 6 shows the result of physical simulation of snow particles. The snow can fall down with an increasing velocity due to gravity and interact with each other to form the effect of snow accumulation.

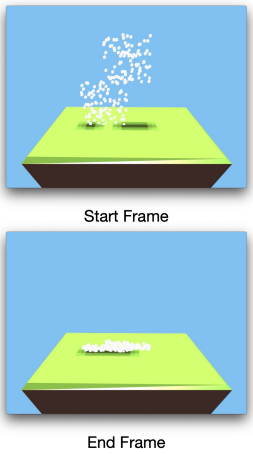

Figure 6: Snow Particle Interaction Effect

As we mentioned in Section 4.3, we designed the UI to let snow particles form a specific shape with *.json* format. In our current user interface we used the heart and egg shape from figure 4. Figure 7a shows the render of heart shape and figure 7b shows the render of egg shape.

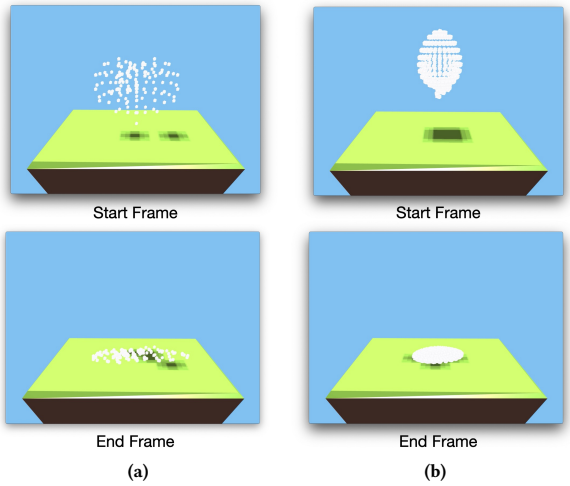

Figure 7: (a): Snow particles with the heart shape; (b): Snow particles with the egg shape

6 CONCLUSION

In this project, we implemented the Material Point Methods in WebGL with *Three.js* to render the snow in real time in the browser. From the result, we can see that the particles interaction works well including the deformation and movement properties. Also, we designed the UI of our wegpage to let users interact with simulation. However, due to the limitation of computation power and the real time rendering, we can only render up to 100 particles, which makes the result not like the real snow. Since in the real world, a snow ball may contains millions of snow particles and we can't render such a large number of particles in the browser with our own laptops. Also, we only used the displacement shading to add the noise on vertex shading without special fragment shading algorithms. We may try to parallel the naive mpm code in the future and find appropriate shading algorithm for snow simulation.

REFERENCES

Silene Cresseri, Francesco Genna, and Cristina Jommi. 2010. Numerical integration of an elastic–viscoplastic constitutive model for dry metamorphosed snow. *International journal for numerical and analytical methods in geomechanics* 34, 12 (2010), 1271–1296.

Brian Danchilla. 2012. Three. js framework. In *Beginning WebGL for HTML5*. Springer, 173–203.

Denys Dutykh, Céline Acary-Robert, and Didier Bresch. 2011. Mathematical Modeling of Powder-Snow Avalanche Flows. *Studies in Applied Mathematics* 127, 1 (2011), 38–66.

Bryan E Feldman and James F O'Brien. 2002. Modeling the accumulation of wind-driven snow. In *ACM SIGGRAPH 2002 conference abstracts and applications*. ACM, 218–218.

Yuanming Hu, Yu Fang, Ziheng Ge, Ziyin Qu, Yixin Zhu, Andre Pradhana, and Chenfanfu Jiang. 2018. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 150.

Nygaard Isaac. 2016. gitAzmisov/snow. https://github.com/Azmisov/snow. Accessed: 2019-04-30.

Chenfanfu Jiang, Craig Schroeder, Joseph Teran, Alexey Stomakhin, and Andrew Selle. 2016. The material point method for simulating continuum materials. In *ACM SIGGRAPH 2016 Courses*. ACM, 24.

Gross Joel. 2015. JAGJ10/Snow. https://github.com/JAGJ10/Snow. Accessed: 2019-04-30.

Theodore Kim, David Adalsteinsson, and Ming C Lin. 2006. Modeling ice dynamics as a thin-film stefan problem. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Eurographics Association, 167–176.

Aleka McAdams, Andrew Selle, Kelly Ward, Eftychios Sifakis, and Joseph Teran. 2009. Detail preserving continuum simulation of straight hair. In *ACM Transactions on Graphics (TOG)*, Vol. 28. ACM, 62.

Tony Parisi. 2012. *WebGL: up and running*. " O'Reilly Media, Inc.".

Ken Perlin. 2002. Improving noise. In *ACM transactions on graphics (TOG)*, Vol. 21. ACM, 681–682.

Toro Roberto. 2019. r03ert0/mls-mpm.js. https://github.com/r03ert0/mls-mpm.js. Accessed: 2019-05-14.

Michael Steffen, Robert M Kirby, and Martin Berzins. 2008. Analysis and reduction of quadrature errors in the material point method (MPM). *International journal for numerical methods in engineering* 76, 6 (2008), 922–948.

Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. 2013. A material point method for snow simulation. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 102.

Deborah Sulsky, Shi-Jian Zhou, and Howard L Schreyer. 1995. Application of a particle-in-cell method to solid mechanics. *Computer physics communications* 87, 1-2 (1995), 236–252.

Jian Tan and Xiangtao Fan. 2011. Particle system based snow simulating in real time. *Procedia Environmental Sciences* 10 (2011), 1244–1249.

Bo Zhu and Xubo Yang. 2010. Animating Sand as a Surface Flow.. In *Eurographics (Short Papers)*. 9–12.