Material Point Method Solver in Houdini

Masters Project

Han Dance
MSc Computer Animation and Visual Effects

August 2020

Contents

1 Introduction

The research undertaken, as well as implementation of the Material Point Method (MPM) in Houdini, will be discussed within this thesis. MPM has gained traction in recent years due to the ability to simulate varied types of materials, as well as the ability to handle topological changes with ease compared to other methods. A vast amount of research in the area since its first implementation in 2013 has been undertaken, to both increase its efficiency and simulate new objects. Houdini has no native MPM system, although SideFX have created ones internally, proving the challenge is possible. Each of the ten MPM steps have been implemented in Houdini using a combination of VEX and gas microsolvers.

This masters project has three goals. The main goal is to create a MPM solver within Houdini from scratch, providing a simple basis that can be easily: accessed, understood and built upon. The secondary goals are, firstly to use this MPM solver to simulate snow, and secondly make a tool where a customisable snow scene can be set up and simulated. The final outcome of the project is mostly successful. The basis for a MPM solver is fully tested, works well and is simple to understand. Furthermore, the tool is easy to follow giving the user basic controls to adjust the simulation. However, the forces and parameters used to create realistic snow behaviours need to be researched in more depth, as the behaviour is not completely similar to that of snow. Unfortunately, this could not be finalised within the time frame of the project.

2 Previous Work

Sulsky et al. [1995] introduced MPM as an extension to FLIP for solid mechanics problems that require compressibility. Similar to FLIP [Zhu and Bridson (2005)], it is a hybrid particle and grid method that combines Lagrangian material points (particles) and a background Eulerian grid. The general idea for MPM is to transfer particle data to the background grid, equations of motion are then solved on the grid, data is transferred back to particles and the particle positions are updated. MPM has many advantages over non-hybrid methods. Firstly, the grid based integration is independent from the number of particles. Changes in topology are easy as there is a lack of mesh connectivity between material points, and therefore there is no problem of tangled meshes and remeshing (a typical problem in FEM). Consequently, a broader array of materials can be simulated than a purely Lagrangian method. Furthermore, MPM has automatic splitting and merging behaviours because of the particle based material representation [Jiang *et al.* (2016)] as well as the automatic self collision and contact. Additionally, external forces can be easily applied.

MPM was first applied to Computer Graphics by Stomakhin et al. [2013] where snow is simulated and solved using a ten step process. It was chosen and implemented for its usefulness in simulating multiphase materials and ability to handle plasticity and fracture. The overall ten step implementation from this paper has been followed to create the custom solver from scratch in Houdini. Since the introduction of MPM to the graphics community, there has been a large amount of research within the area. Stomakhin et al. [2014] looked further at phase changing materials with the addition of heat in the context of MPM to simulate melting. Ram et al. [2015] add a volume-preserving Oldroyd-B rate-based description of plasticity for plastic flow of viscoelastic fluids. Klár et al. [2016] produce sand animations using MPM due to its natural treatment of contact, whilst Wretborn et al. [2017] extend to animating cracks, also due to the treatment of collisions. Qi et al. [2018] extend to cloth simulations with frictional contact. A method for simulating realistic bread and other baked foods has recently been developed accounting for the intricacies of cooking and baking [Ding *et al.* (2019)]. Most recently MPM has been used to simulate anisotropic elastoplastic material behaviours such as the dissolution of fibrous phenomena for example shedding bales of hay [Schreck and Wojtan (2020)].

Houdini currently has no native MPM system, the current way to create snow is using Grains, which apply Position Based Dynamics (PBD). Although this method is simple it does not describe the details of snow completely accurately. Houdini does however have a FLIP solver for fluid objects. The background grid is represented by fields, a box with a position, size, and orientation, subdivided into a 3D grid of voxels, with a value stored in each voxel [SideFX (2019b)]. The FLIP solver is very complex and mostly consists of gas microsolvers, which perform various specific mathematical tasks to fields, wiring these solvers together can produce complex results [Claes (2009)]. Currently Houdini has approximately one hundred microsolvers, some of which are digital assets made up of other microsolvers.

A selection of microsolvers useful in the context of FLIP / MPM and their purpose include:

- Gas Field to Particle - Samples the values of a field to a geometry point attribute using trilinear interpolation.
- Gas Field Wrangle - Runs a snippet of VEX on every voxel on the input volume. This is a very powerful tool and a whole MPM system could be made entirely of wrangle nodes.
- Gas Linear Combination - Combines multiple fields together with a choice of operation.
- Gas Match Field - Rebuilds a field to match the size and reference of a different field.
- Gas Particle to Field - Stamps a value of point attribute from geometry to a field using the Elendt kernel.
- Gas Resize Field - Changes the size of fields, can be based on reference geometry.
- Gas Synchronize Field - Matches the centers and sizes of the target fields to the reference field.

At the root of the FLIP solver is a Multiple Solver node, which causes a simulation object to be solved by more than one solver at each time step, each solver attached to this node is applied in order to the simulation object [SideFX (2019a)]. This allows the gas microsolver nodes and others to be merged and input into the Multiple Solver, therefore creating a custom solver. The custom MPM solver was created in a similar fashion to the Houdini FLIP solver, due to the similarities and solid performance of the FLIP solver. A large portion of the research in Houdini involved diving in to the FLIP solver to understand the intricacies of how it works, and how the user can control different aspects of it. A simplified FLIP solver was created from scratch using gas microsolvers and the Multiple Solver node to put this understanding into practice.

3 Technical Background

To fully understand the implementation of the created MPM solver within Houdini, a detailed description of the proposed ten step Material Point Method and accompanying mathematical theory is presented in this section. As stated MPM is a hybrid method that utilises both Lagrangian particles (which in this case represents the snow) and a Eulerian grid (for various calculations). Figure 1 shows an overview of the Material Point Method, where the top row represents steps applied to particles and the bottom row steps applied to the grid. The background grid is fixed during each time step and once the calculations are applied to the particles the grid is reset, this means the grid does not have to be static and can be displaced to follow the particles.

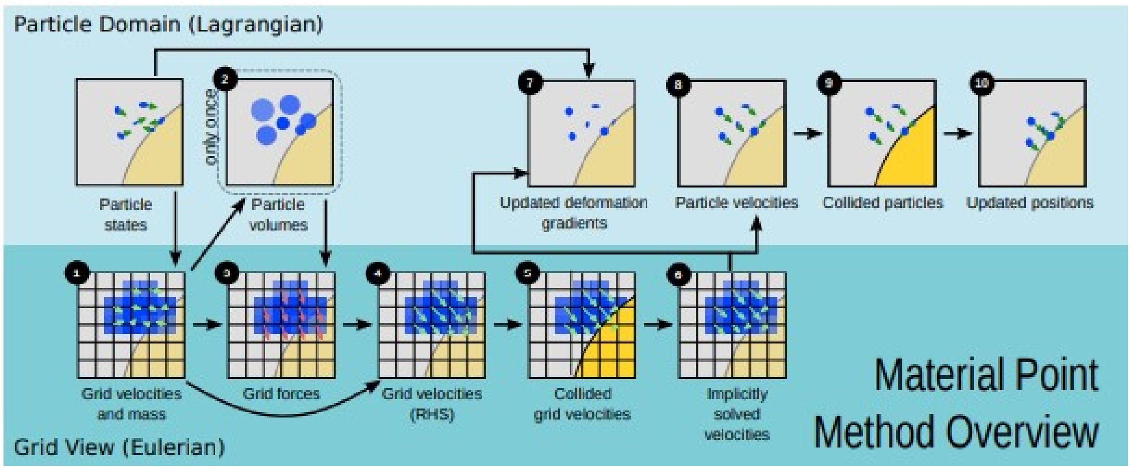


Figure 1: Overview of the Material Point Method (from Stomakhin et al. 2013)

The first step is to transfer the velocity and mass of the particles to the background grid. This is implemented with the two equations

$$m_i^n = \sum_p m_p w_{i\,p}^n \tag{1}$$

$$v_i^n = \sum_p \frac{v_p^n m_p w_{i\,p}^n}{m_i^n} \tag{2}$$

where m_i^n is the mass of grid node i at time n, v_i^n is the velocity of grid node i at time n, v_p^n is the velocity of the particle at time n, m_p the mass of the particle and $w_{i\,p}^n$ is the weighting function between particle p and grid node i at time n, calculated as the product of one dimensional cubic B-splines such that

$$w_{i\,p}^{n} = N(\frac{1}{h}(x_p - i))N(\frac{1}{h}(y_p - j))N(\frac{1}{h}(z_p - k)) \tag{3}$$

where i = (i,j,k) is the grid node position, x_p is the particle position, h is the spacing between grid nodes and

$$N(x) = \begin{cases} \frac{1}{2}|x|^3 - x^2 + \frac{2}{3}, & \text{for } 0 \le |x| < 1 \\ -\frac{1}{6}|x|^3 + x^2 - 2|x| + \frac{4}{3}, & \text{for } 1 \le |x| < 2 \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

This results in grid nodes only being influenced by particles that are within $2h$ distance in the x, y and z directions. Figure 16 shows the graphical representation of $N(x)$.

The gradient of the weighting function is used within later steps and is defined as:

$$\nabla w_{i\,p}^{n} = \begin{pmatrix} (\frac{1}{h})N'(\frac{1}{h}(x_p - i))N(\frac{1}{h}(y_p - j))N(\frac{1}{h}(z_p - k)) \\ N(\frac{1}{h}(x_p - i))(\frac{1}{h})N'(\frac{1}{h}(y_p - j))N(\frac{1}{h}(z_p - k)) \\ N(\frac{1}{h}(x_p - i))N(\frac{1}{h}(y_p - j))(\frac{1}{h})N'(\frac{1}{h}(z_p - k)) \end{pmatrix} \tag{5}$$

where

$$N'(x) = \begin{cases} \frac{|x|}{x}\frac{3}{2}x^2 - 2x, & \text{for } 0 \le |x| < 1 \\ -\frac{|x|}{x}\frac{1}{2}x^2 + 2x - 2\frac{|x|}{x}, & \text{for } 1 \le |x| < 2 \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

Figure 17 shows the graphical representation of $N'(x)$.

The second step calculates the initial volume of each particle, which will be used within a later force calculation. Therefore, it is only executed at the first time step. The density of each grid cell is estimated as $\frac{m_i}{h^3}$ and can be weighed back to the particle density as

$$\rho_p = \sum_i \frac{m_i^0 w_{i\,p}^0}{h^3}. \tag{7}$$

Therefore a particles volume can be calculated using the equation:

$$V_p = \frac{m_p}{\rho_p}. \tag{8}$$

The upcoming step involves the deformation gradient F, the fundamental measure of deformation in continuum mechanics. It is the second order tensor which maps line elements in the reference configuration into line elements (consisting of the same material particles) in the current configuration [Kelly (2012)]. Each particle has a deformation gradient attribute F, that is initialised as I the identity matrix at the beginning of the simulation, as this notes lack of deformation and rotations. It is customary to separate F into an elastic part F_e and plastic part F_p such that $F = F_eF_p$. Where elastic deformation is reversible and plastic is permanent.

The thresholds to start plastic deformation are θ_c and θ_s, the critical compression and the critical stretch. Initially all changes are assumed to belong to the elastic part of the deformation gradient such that $F = F_e$. The singular value decomposition of F is then calculated such that

$$\hat{F} = U\hat{\Sigma}V^T \tag{9}$$

where $\hat{\Sigma}$ is a diagonal matrix containing singular values. These singular values are then clamped to a permitted range

$$\Sigma = \text{clamp}(\hat{\Sigma}, 1 - \theta_c, 1 + \theta_s) \tag{10}$$

F_e and F_p are then calculated with the following formula

$$F_e = U\Sigma V^T \text{ and } F_p = V\Sigma U^T F. \tag{11}$$

The third step computes the forces at the grid nodes which is evaluated using the following formula.

$$f_i(\hat{x}) = -\sum_p V_p^n \sigma_p \nabla w_{i\,p}^n \tag{12}$$

where V_p^n is the volume of the material occupied by particle p at time n and σ_p is the Cauchy stress. $V_p^n = J_p^n V_p^0$ where J_p^n = Determinant(F_p^n).

Cauchy stress σ is derived in the accompanying technical portion of the paper as

$$\sigma = \frac{2\mu}{J}(F_e - R_e)F_e^T + \frac{2}{\lambda}(J_e - 1)J_eI \tag{13}$$

where J and J_e are the determinants of F and F_e respectively. The deformation gradient F can be written as $F = RS$ where R is the rotation matrix and S is the symmetric matrix describing the deformations, sources of stress [McGinty (2012)], these matrices are calculated through polar decomposition. Therefore R_e is the rotation matrix relating to the elastic part of the deformation gradient. The Lamé parameters μ and λ are functions of the plastic deformation gradient and calculated such that

$$\mu(F_p) = \mu_0 e^{\xi(1-J_p)} \text{ and } \lambda(F_p) = \lambda_0 e^{\xi(1-J_p)} \tag{14}$$

where ξ is the hardening coefficient. The initial The Lamé coefficients μ_0 and λ_0 are calculated with the following equation.

$$\mu_0 = \frac{E_0 v}{(1+v)(1-2v)} \text{ and } \lambda_0 = \frac{E_0}{2(1+v)} \tag{15}$$

where E_0 is the initial Young's modulus and v Poisson's ratio.

Step four updates the grid velocities using the equation

$$v_i^* = v_i^n + \frac{\Delta t f_i^n}{m_i} \tag{16}$$

to include gravity in the simulation, the force equation (12) is updated to

$$f_i(\hat{x}) = -\sum_p V_p^n \sigma_p \nabla w_i^n{}_p + g m_i \tag{17}$$

which cancels out the mass division from (16).

The collisions are processed twice per time step. The first is on the grid velocity v_i^* and is updated as follows. First the relative velocity between the grid node and the collision object is calculated:

$$v_{rel} = v_i^* - v_{co}. \tag{18}$$

The dot product between this and the normal of the collision object is taken.

$$v_n = n \cdot v_{rel} \tag{19}$$

If v_n is positive it means the grid node and objects are moving away from each other and the calculation can be terminated, if v_n is negative then the calculation continues. The tangential portion of the relative velocity is then calculated as

$$v_t = v_{rel} - n v_n. \tag{20}$$

Let μ be the friction coefficient. A sticking impulse is required if $(\|v_t\| < -\mu v_n)$ therefore the relative velocity $v'_{rel} = 0$. Else

$$v'_{rel} = v_t + \mu v_n \frac{v_t}{\|v_t\|}. \tag{21}$$

The final step then transforms this collision back from the relative space to real world coordinates as such

$$v_i^* = v'_{rel} + v_{co}. \tag{22}$$

Step six solves the linear system explicitly such that

$$v_i^{n+1} = v_i^*. \tag{23}$$

The deformation gradient at each particle is then updated with the following formulae

$$F^{n+1} = (I + \Delta t \nabla v_p^{n+1}) F^n, \tag{24}$$

$$\nabla v_p^{n+1} = \sum_i v_i^{n+1} (\nabla w_{i\,p}^n)^T. \tag{25}$$

This will be used to calculate F_e and F_p in the next time step as described above.

Velocities are then transferred back to particles using the equation

$$v_p^{n+1} = (1 - \alpha) v_{PICp}^{n+1} + \alpha v_{FLIPp}^{n+1}. \tag{26}$$

A combination of FLIP update, where particle velocity is incremented by the delta in grid velocity, and PIC where the particles are updated with the new grid velocity, is implemented. The FLIP approach is better at maintaining individual particle velocities but can become chaotic. α is the ratio of FLIP in the blend, an α of 0.95 is a common value. The FLIP and PIC updates are calculated using the following equations

$$v_{PICp}^{n+1} = \sum_i v_i^{n+1} w_{i\,p}^n, \tag{27}$$

$$v_{FLIPp}^{n+1} = v_p^n + \sum_i (v_i^{n+1} - v_i^n) w_{i\,p}^n. \tag{28}$$

Collisions are then processed a second time to account for any minor discrepancies. The exact same process is used as grid collisions except collisions are particle based meaning the update is applied to v_p^{n+1} as opposed to the v_i^*.

Finally the particle positions are updated in a standard manner.

$$x_p^{n+1} = x_p^n + \Delta t v_p^{n+1} \tag{29}$$

To summarise the main steps for the MPM solver are as follows:

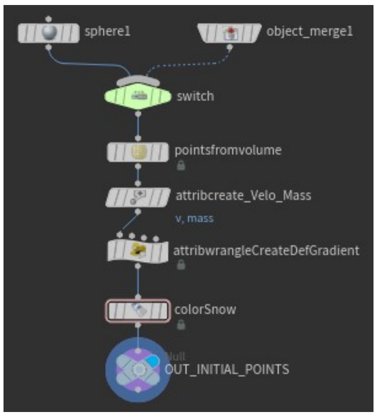
1. Transfer particle data to the background grid.
2. Calculate initial particle volumes (first timestep only).
3. Compute forces on the background grid.
4. Update grid velocities.
5. Resolve grid collisions.
6. Solve the linear system via explicit time integration.
7. Update the particle's deformation gradient F.
8. Transfer back from the grid to particles, updating their velocity using a combination of FLIP and PIC.
9. Resolve particle collisions.
10. Update particle position.

4 Implementation

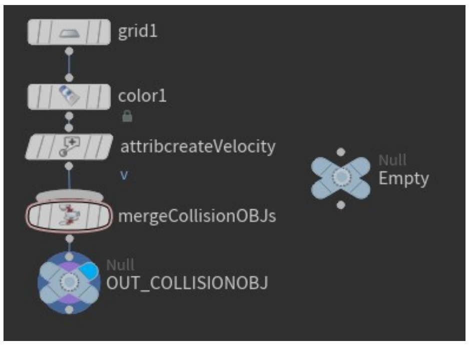
The main goal of the project is to create a Material Point Method solver from scratch in Houdini, closely following the implementation steps described in the paper 'A material point method for snow simulation'. Secondary goals are simulating realistic snow behaviour using this solver and implementing an usable tool. The created tool should allow users to define their snow objects, collision objects and the solvers parameters to change the behavior of the generated snow.

Each of the ten simulation steps outlined in the paper are implemented in Houdini and wired into the Multiple Solver node. This is achieved through a combination of wrangle nodes and gas microsolvers as detailed in Section 2.

Testing of the VEX code written in the various wrangle nodes was carried out with two main approaches. Firstly, any created functions and shorter pieces of code were tested on SOP geometry outside the DOP context, where the input values could easily be varied and the output was readily accessible. The second approach was running the DOP simulations with small particle and grid samples, to test the values of parameters for each step, with field values being accessed using the printf VEX function. Although, a simulation with just one particle means there are sixty-four voxels surrounding it, each having separate weight calculations that need to be checked, resulting in the checks taking a substantial amount of time. Values at each voxel of a field are not viewable in the geometry spreadsheet compared to particle data, this meant it was harder to access the values quickly, to test if they were as expected, so the approach to print out values in VEX had to be taken. All implemented VEX functions created for the custom solver have been tested with appropriate values, and no bugs have been found.



(a) Snow Particles



(b) Collision Geometry

Figure 2: Node Network in Houdini to create the initial geometry

The SOP geometry for the snow particles and collision objects are created by the user and imported into the DOP simulation as the initial state. For the former, any initial defined geometry has points created from it using the Points From Volume node. Attributes for velocity, mass and deformation gradient are assigned to these points using the Attribute Create and Attribute Wrangle nodes. An optional Merge node can be included for more complex simulations involving different initial geometries with varying initial conditions. These points are wired into a final Null node that is called from the DOP simulation. A similar process is used to create the collision geometry, although only the velocity attribute is created.

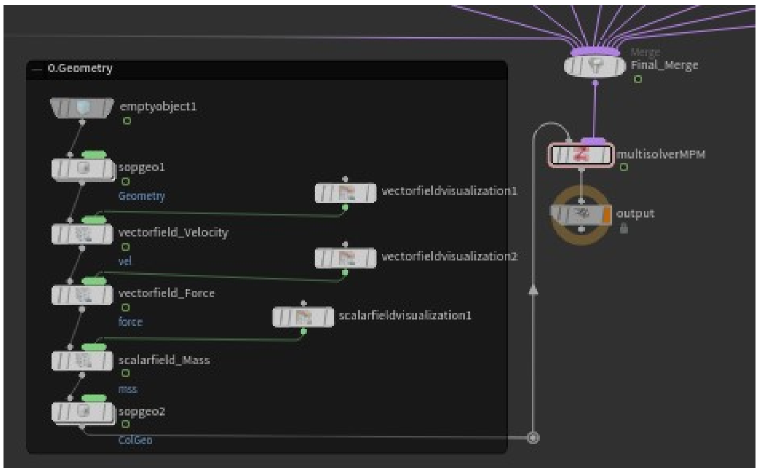
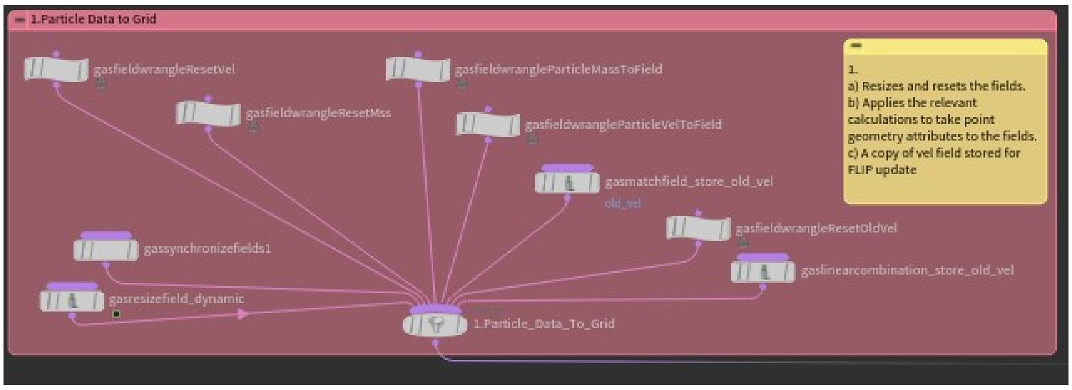


Figure 3: Initialing the Geometry and Fields within the DOP network

Within the DOP network the geometry and main fields are initialised, and plugged into the first input of the Multiple Solver node. SOP Geometry nodes are used to read the created SOP data. Vector fields for velocity and force and a scalar field for mass are also created. Fields within Houdini can either contain a vector or scalar, so multiple fields are needed to represent the single background grid, this is a similar implementation of a grid to Houdini's native FLIP solver. Visualisation nodes allow the direction and magnitude of the velocity and force to be viewed at each complete timestep, allowing for quick checks.

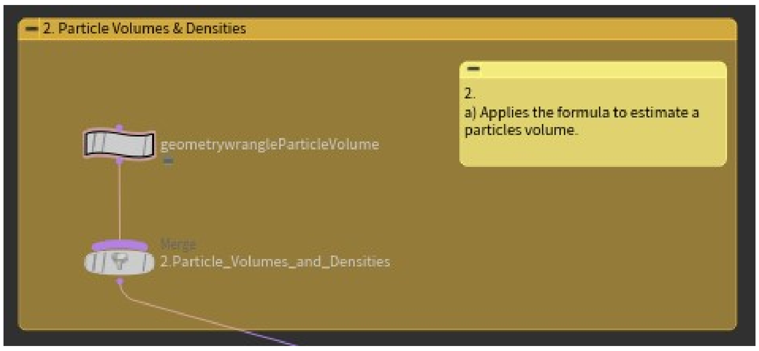
The first of the ten steps uses a Gas Resize Field node so the fields are centered on the particles and encapsulate them. A static grid is not needed for MPM as current grid node values are not dependent on their previous values, by employing a dynamic grid, the number of grid nodes are drastically reduced, resulting in fewer calculations. After resizing the fields, mass and velocity of the particles is transferred to each respective field using a Gas Field Wrangle. Inside each wrangle the transfer equations and weighting functions have been implemented using VEX. A point cloud is used at each grid voxel node to run a nearest neighbour search, the data structure of a point cloud makes it faster to run computations and takes up less space. Using the pciterate function, all the neighbour points in turn are used within the summation function, which automatically stops when there are no points remaining to iterate over. Finally, a new field oldvel is created as a duplicate of the velocity field v_i^n to be used in the FLIP update of step eight.


1.Particle Data to Grid
gasfieldwrangleResetVel
gasfieldwrangleParticleMassToField
gasfieldwrangleResetMss
gasfieldwrangleParticleVelToField
gasmatchfield_store_old_vel
old_vel
gasfieldwrangleResetOldVel
gassynchronizefields1
gaslinearcombination_store_old_vel
gasresizefield_dynamic
1.Particle_Data_To_Grid

1.
a) Resizes and resets the fields.
b) Applies the relevant calculations to take point geometry attributes to the fields.
c) A copy of vel field stored for FLIP update

Figure 4: Step 1 Node Network.

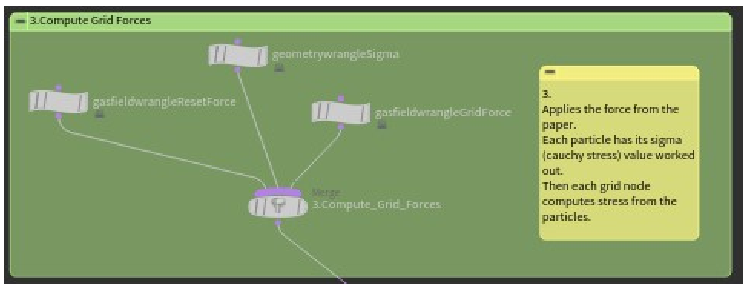
To calculate a particle's volume a Geometry Wrangle node is implemented to execute VEX over each material point. To iterate over all field voxels a point cloud cannot be used as in the previous step, this is because the voxels are not points and there are multiple fields. Therefore, three empty arrays were created, representing the x, y and z directions and the volumeres function was used to retrieve the size of the fields (which are identical) returned in a vector. The corresponding components of this vector are used to populate the three empty arrays with a custom function. This function uses a while loop and takes the size of the volume (a) and empty array, and returns an array [0,1,..,a-1]. It only returns up to a-1 because the size of the volume will start at 1 but the index values start at 0. Three nested for-loops that run over the x,y and z arrays are then used to access the index of each voxel.


2. Particle Volumes & Densities
geometrywrangleParticleVolume
Merge
2.Particle_Volumes_and_Densities

2.
a) Applies the formula to estimate a particles volume.

Figure 5: Step 2 Node Network.

The volumeindextopos and volumesample functions access the voxels position and stored value respectively, allowing the density at each grid cell and weighting function to be calculated. Finally, to guarantee this step only runs at the start of a simulation, an if statement that calls the global variable @SimTime was added.

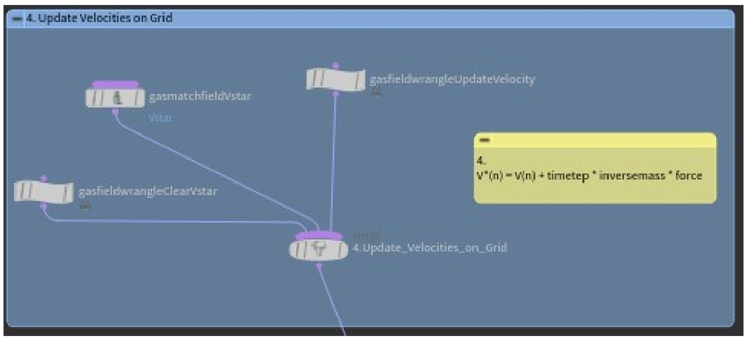


3.
Applies the force from the paper.
Each particle has its sigma (cauchy stress) value worked out.
Then each grid node computes stress from the particles.

Figure 6: Step 3 Node Network.

Force calculation is executed in two sub-steps. The first uses a Geometry Wrangle node to calculate σ (Cauchy stress) at each material point. A function was created to calculate the The Lamé parameters. The VEX functions clamp, svddecomp and polardecomp are implemented to limit the elastic deformation gradient, apply a singular value decomposition to retrieve the scaling matrix and apply a polar decomposition to retrieve the rotation matrix respectively, as detailed in Section 3. The decomposition values were tested against an external decomposition calculator and return expected results.

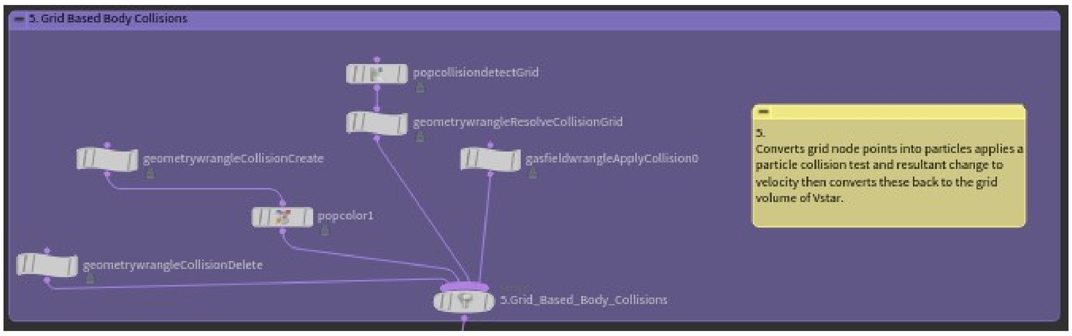
The second sub-step uses a Gas Field Wrangle node on the force field. This again implements the created weighting function, but also introduces the created function that is the gradient of the weighting function. A point cloud is opened per voxel in the same manner as step one to run over the particles. Finally multiplication between the sigma matrix and gradient weight vector to retrieve the final force for each voxel is implemented through a custom function that inputs a 3*3 matrix and a 3*1 vector and outputs the resultant 3*1 vector.


4. Update Velocities on Grid
gasmatchfieldVstar
Vstar
gasfieldwrangleUpdateVelocity
gasfieldwrangleClearVstar
Merge
4.Update_Velocities_on_Grid

4.
V*(n) = V(n) + timetep * inversemass * force

Figure 7: Step 4 Node Network.

Step four is implemented in a relatively quick fashion, following the paper a new field V^* is created as a copy of the velocity field using a Gas Match Field node. This new field is then updated by implementing equation (16) within a Gas Field Wrangle node, the values of the mass and force fields are accessed using the volumesamplev and volumesample VEX functions.

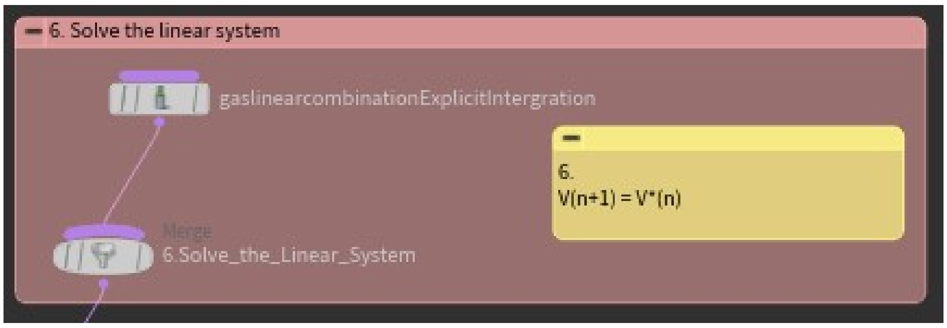

5. Grid Based Body Collisions
popcollisiondetectGrid
geometrywrangleResolveCollisionGrid
geometrywrangleCollisionCreate
gasfieldwrangleApplyCollision0
popcolor1
geometrywrangleCollisionDelete
5.Grid_Based_Body_Collisions

5.
Converts grid node points into particles applies a particle collision test and resultant change to velocity then converts these back to the grid volume of Vstar.

Figure 8: Step 5 Node Network.

Grid based body collisions were initially going to be implemented using a collision field and response in a similar fashion to the Houdini FLIP solver, however because of the custom collision response presented in the paper this proved challenging. Therefore, a new approach was utilised. At each timestep a new set of point geometry is created, this point geometry is representative of the grid nodes. A Geometry Wrangle node accesses each voxel using the same array and nested for loop process used in Step two. The position and velocity (obtained from from the field V^*) of each voxel is saved to the new geometry using the addpoint and setpointatrrib VEX functions. A POP Collision Detect node is used to return if the newly formed grid point geometry is colliding with the collision objects and that object's: position, velocity and normal are saved as attributes.

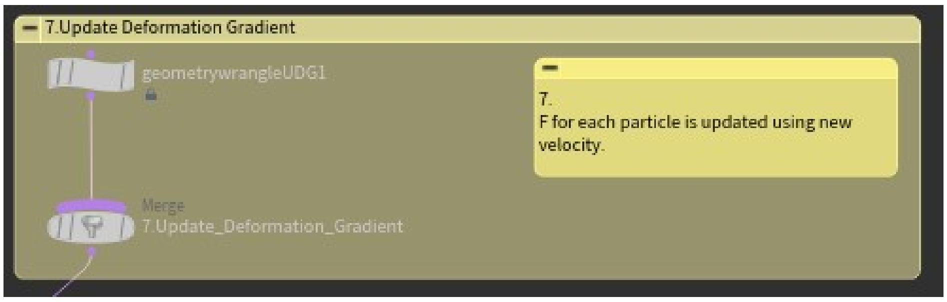
A further geometry wrangle then applies the collision response to the velocity of the newly formed grid point geometry as described in Section 3. This velocity is then transferred back to the field V^* through the means of a Gas Field Wrangle node. Making sure the points matched back to the field voxels was very important and a at first there was a minor issue when creating additional attributes to do this, they did not match in certain cases. However after various tests on @ptnum of the created geometry, the correct formula to transfer back to the voxels was found. The index of each voxel and size of the overall field is used to create a unique id for the voxel, as shown in figure 18. This id is the same as the particle number that matches the voxel, therefore, the velocity of that particle number equal to the unique id is assigned to the voxel. Additionally, the new point geometry representing the grid nodes is deleted at the start of this step.


6. Solve the linear system
gaslinearcombinationExplicitIntergration
Merge
6.Solve_the_Linear_System

6.
V(n+1) = V*(n)

Figure 9: Step 6 Node Network.

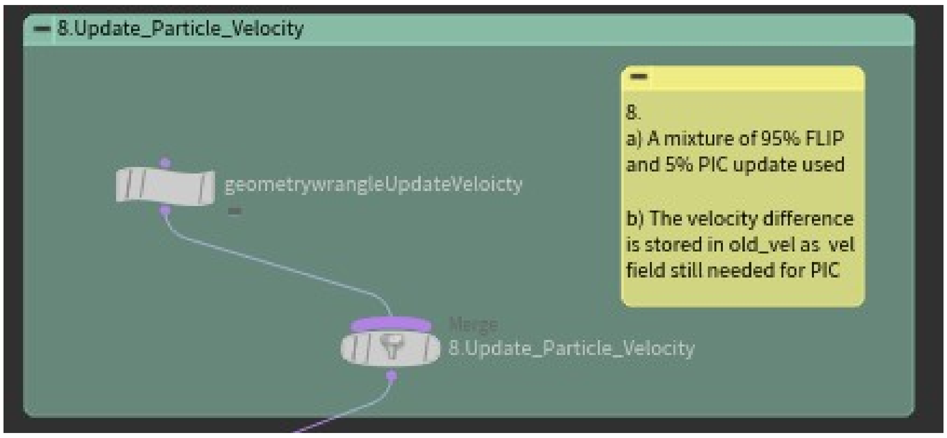
Explicit time integration is solved simply through a Gas Linear Combination node. The velocity field value is replace with value of the field in $V*$. To save space in the future and not follow the paper precisely, the $V*$ field could be omitted with changes from steps four and five made to the initial vector field.



7.
F for each particle is updated using new velocity.

Figure 10: Step 7 Node Network.

The deformation gradient update for each particle is carried out in a Geometry Wrangle node. It again cycles through each voxel by using nested for loops and arrays based on the grid size. The weighting functions are again called in this node, and a new function is created to carry out matrix multiplication. An issue was noticed at this step where accessing the position data of the voxel with the volumeindextopos function returned a very small floating point error, due to the spacing between voxels being a derived quality, since grid transforms are stored as a size and origin. This floating point error of the position of each voxel meant the symmetry around each particle was lost, and values for the gradient of the weighting function had minor errors. Instead of values of 0 the deformation gradient had values close to e^{-10}. A fix was put in place to round up any errors in position but when comparing the small floating errors had very minimal effect on the outcome so the fix was not included.

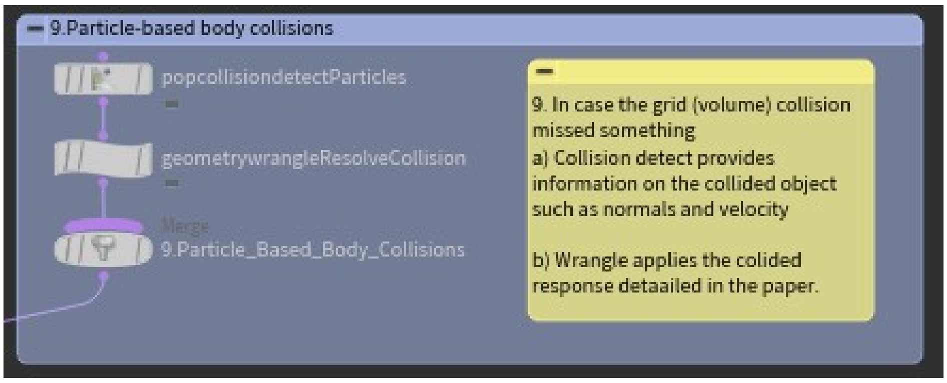

8.Update_Particle_Velocity
geometrywrangleUpdateVeloicty
Merge
8.Update_Particle_Velocity

8.
a) A mixture of 95% FLIP and 5% PIC update used

b) The velocity difference is stored in old_vel as vel field still needed for PIC

Figure 11: Step 8 Node Network.

Step eight, the velocity update step, uses a Geometry Wrangle node and accesses the voxel data in the same fashion as the previous steps. The FLIP update portion uses the velocity difference and accesses the field oldvel saved in step one.

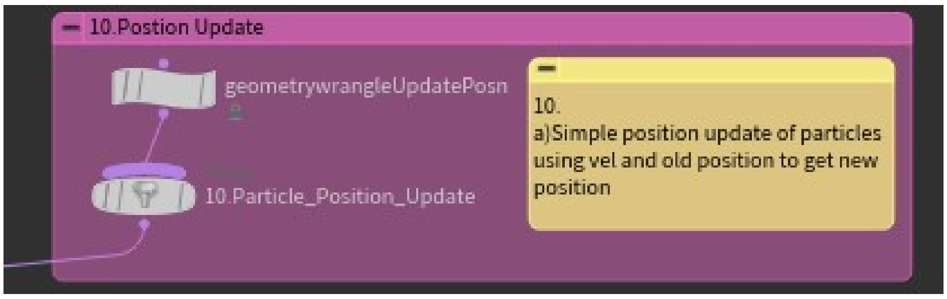

9.Particle-based body collisions
popcollisiondetectParticles
geometrywrangleResolveCollision
Merge
9.Particle_Based_Body_Collisions

9. In case the grid (volume) collision missed something
a) Collision detect provides information on the collided object such as normals and velocity

b) Wrangle applies the colided response detaailed in the paper.

Figure 12: Step 9 Node Network.

The particle collision step is applied using the exact same approach and VEX code as step five, grid based body collisions. Although this time the pop collision detect and resultant VEX code is applied to the particle data. When running the simulation with only this collision step the particles clearly collide and respond in a clean manner, with a change in the friction coefficient leading to a direct change in the speed of particles moving down a slope.

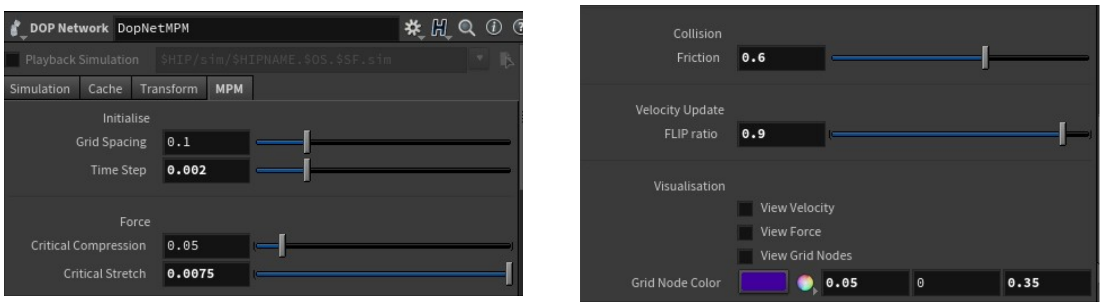


10.
a)Simple position update of particles using vel and old position to get new position

Figure 13: Step 10 Node Network.

Each particle's position is updated in the final step through VEX code in a Geometry Wrangle node. This step was fairly quick to implement and test.

Figure 14 shows the implementation of user controls for the MPM solver. The grid spacing and time step parameters control how quickly the simulation runs, however having these at too high values can cause an unstable result. The friction coefficient changes parameters in steps five and nine and successfully controls how easily the snow particles slide on different surfaces. Visualisation is very important for a user of the tool, being able to see how the velocity and force is behaving will allow users to understand what is happening with the solver. Therefore, being able to toggle these on and off easily is desirable for the final tool. Figures 28-30 shows these controls turned on.



DOP Network DopNetMPM

Initialise
Grid Spacing 0.1
Time Step 0.002

Force
Critical Compression 0.05
Critical Stretch 0.0075

Collision
Friction 0.6

Velocity Update
FLIP ratio 0.9

Visualisation
View Velocity
View Force
View Grid Nodes
Grid Node Color 0.05 0 0.35

Figure 14: User Controls

The VEX code works effectively and implements the ten steps properly. The custom created functions are easy to follow and efficiently calculate the needed result, the functions generated to support the solver steps are included in the appendix. Having to include the created functions in multiple wrangle blocks is a little messy, but the overall wrangles are still easy to follow. The final solver is efficient in regards to the goal of closely implementing the steps from the paper. Further development can be taken to have a faster MPM simulation in Houdini, that deviates from the paper.

5 Conclusion

The main objective to implement a Material Point Method solver from scratch within Houdini has been achieved successfully. The implemented solver works well, and behaves as expected, when given simple force examples - such as gravity only. The transfer of particle data to fields and back again using VEX is accomplished, and all tests for the solver pass successfully, the particles change in velocity is correct after each timestep. Furthermore, the custom collisions calculated with these simple forces behave appropriately and realistically.

Achieving realistic snow like behavior, a secondary goal, has been partially achieved. Compared to real snowballs smashing, there are some similarities but also some differences. The overall shape of the snow breaking up and separating out towards the side (figure 25) is a noticeable similarity, and shows the overall behavior as successful. However, the response of the snow moving up (figure 24) acts more like a bouncy fluid and less like a solid. A more detailed understanding of the force calculation (Step 4) needs to be undertaken, as the reasons are currently unknown as to why the snow is not behaving exactly as expected. There may be an error in understanding and implementing the force calculation that is causing this, or another calculation for the forces may need to be used. Figure 32 shows another error where the bottom block of snow does respond to the ball by separating outwards, but the crashing snowball bounces on top of the block and they do not mix.

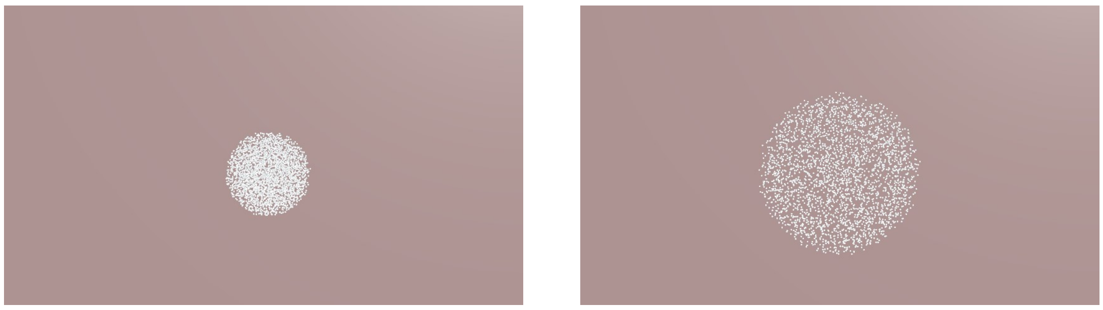


Figure 15: Particle formation before and after colliding with the ground

There are many different initial conditions and parameters that affect the final outcome of the implemented force calculation, including: grid spacing, mass, Young's modulus and hardening coefficient, a deeper understanding on how these relate and affect the final outcome is needed to achieve the desired accurate snow behaviour and look, sadly this research could not be undertaken within the time frame of the project. Small changes in some of these values can cause the simulation to become unstable and this needs to be investigated. Figure 27 shows the snow avoiding collision on the slope due to the grid collision response, small dents can be seen spaced along the bottom of the snow where the grid nodes are located and pushing the snow away. This would be less obvious with lower grid spacing, however the solver had to run in a realistic time. Smaller grid spacing and more points would likely give more realistic results as this affects the calculations, this will be tested when solver efficiency is updated. The high grid spacing may be the reason the that the separate objects in the collision example (figures 31 32) affected each other at a large distance and did not mix. Finally, the fracturing of the snowball needs a level of randomness as it breaks up into a somewhat even circle, as shown in figure 15, the paper implements a noise function for a more grouped fracture, this would be a reasonable next step to achieve a more natural result and would be evident with more particles.

The final secondary goal of providing a usable tool for snow simulation implemented with the MPM method is mostly complete. The key parameters to change the simulated behaviour of snow are given and any more could complicate the tool for the user. Giving the option of grid sizing and timestep allows the user to control how detailed they want the simulation to be. A more complex tool would allow the user to define multiple collision objects with varying friction coefficients and would be a next step in upgrading it. A further improvement would be to turn the tool into a digital asset as opposed to having changeable parameters on the DOP node, the inclusion of a help card would dramatically increase the understanding for the user.

Only the semi-implicit integration step (a more detailed optional part of step six) is missing out of the ten steps. Although not compulsory the inclusion of this would result in more accurate simulations. A next step in the project would be to implement this and compare the outcome of the solver against the current explicit time integration, to evaluate if the complexity of the computation is worth the benefit.

The overall solver was created in a similar fashion to the Houdini FLIP solver where multiple fields are used alongside geometry particles. It is noted that some of the gas microsolvers used in the FLIP solver such as the Gas Field to Particle and Gas Particle to Field nodes are not included in the MPM solver. These steps have been undertaken in the Gas Wrangle nodes instead to apply a custom weighting using VEX, some of the microsolvers were used in FLIP as VEX was not available in DOPS at the time. It would be interesting to make an entire MPM solver using only wrangles as a next step as, this lets the math be easily controlled and followed, as opposed to the microsolvers who's readily available documentation does not go fully into the math being implemented.

Following the approach of the FLIP solver gets complicated in Step five (grid based body collision) when particle geometry is created and deleted for each of the grid nodes (voxels) at every time step. There are two directions that this solver could be taken in to avoid having to continuously create and destroy this geometry in addition to the fields. The first would be to implement the solver with a collison field that provides a signed distance field, and use this to implement the custom response, this was the initial idea for the solver, but needed more time to implement as it was not working as expected. The detection was not finding the SOP data to collide with and implementing the custom response was more complex than initially thought. Further research would need to be undertaken to implement this approach, and if the custom response is not possible a compromise of collision response would need to be determined. The second approach would be to not use any fields at all and create the background grid at each time step from particles, all the field values could be saved to one particle and point clouds would be used for all particle searches. When it comes to very large grids with small spacing there would be a lot of particles that take up a substantial amount of memory.

Both of these approaches would then be compared thoroughly to investigate which is the most efficient. A final note on the efficiency of the solver is that $w_i^n{}_p$ and $\nabla w_i^n{}_p$ are calculated between a particle and grid node four and two times throughout each simulation step, investigating a way to save these values would greatly increase simulation time, if the memory pay off was worth it.

The main goal in developing this project further would be to create a tool that solves a wide variety of simulations within Houdini using the created MPM solver, such as sand and baked objects as mentioned in Section 2. A solid basis of an MPM solver has been provided, therefore it should be relatively easy to appended and adjusted for simulating these materials, as well as the outcome from future research.

6 Bibliography

Claes P., 2009. *Controlling Fluid Simulations with Custom Fields in Houdini*, [online]. Available from: `https://nccastaff.bournemouth.ac.uk/jmacey/MastersProjects/MSc09/Claes/thesis/PeterClaesThesis.pdf` [Accessed: 18 August 2020] .

Ding M., Han X., Wang S., Gast T. F. and Teran J. M., November 2019. A thermomechanical material point method for baking and cooking. *ACM Trans. Graph.*, 38(6).

Guo Q., Han X., Fu C., Gast T., Tamstorf R. and Teran J., July 2018. A material point method for thin shells with frictional contact. *ACM Trans. Graph.*

Jiang C., Schroeder C., Teran J., Stomakhin A. and Selle A., 2016. The material point method for simulating continuum materials. In *ACM SIGGRAPH 2016 Courses*, SIGGRAPH '16, New York, NY, USA. Association for Computing Machinery.

Kelly P., 2012. *Solid Mechanics Part 3*, [online]. Available from: `http://homepages.engineering.auckland.ac.nz/~pkel015/SolidMechanicsBooks/Part_III/Chapter_2_Kinematics/Kinematics_of_CM_02_Deformation_Strain.pdf` [Accessed: 18 August 2020] .

Klár G., Gast T., Pradhana A., Fu C., Schroeder C., Jiang C. and Teran J., July 2016. Drucker-prager elastoplasticity for sand animation. *ACM Trans. Graph.*, 35(4).

McGinty B., 2012. *Polar Decomposition*, [online]. Available from: `https://www.continuummechanics.org/polardecomposition.html` [Accessed: 18 August 2020] .

Ram D., Gast T., Jiang C., Schroeder C., Stomakhin A., Teran J. and Kavehpour P., 08 2015. Material point method for viscoelastic fluids, foams and sponges.

Schreck C. and Wojtan C., 2020. A practical method for animating anisotropic elastoplastic materials. *Computer Graphics Forum*, 39(2), 89–99.

SideFX , 2019a. *Multiple Solver dynamics node*, [online]. Available from: `https://www.sidefx.com/docs/houdini/nodes/dop/multisolver.html` [Accessed: 18 August 2020] .

SideFX , 2019b. *Volumes*, [online]. Available from: `https://www.sidefx.com/docs/houdini/model/volumes.html` [Accessed: 18 August 2020] .

Stomakhin A., Schroeder C., Chai L., Teran J. and Selle A., July 2013. A material point method for snow simulation. *ACM Trans. Graph.*, 32(4).

Stomakhin A., Schroeder C., Jiang C., Chai L., Teran J. and Selle A., July 2014. Augmented mpm for phase-change and varied materials. *ACM Trans. Graph.*, 33(4).

Sulsky D., Zhou S.-J. and Schreyer H. L., 1995. Application of a particle-in-cell method to solid mechanics. *Computer Physics Communications*, 87(1), 236 – 252. Particle Simulation Methods.

Wretborn J., Armiento R. and Museth K., December 2017. Animation of crack propagation by means of an extended multi-body solver for the material point method. *Comput. Graph.*, 69(C), 131–139.

Zhu Y. and Bridson R., July 2005. Animating sand as a fluid. *ACM Trans. Graph.*, 24(3), 965–972.

A Graphs

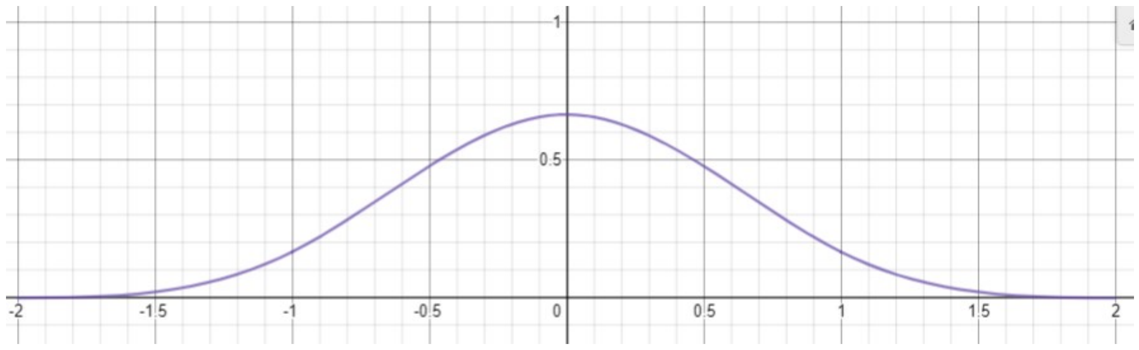


Figure 16: y = N(x)

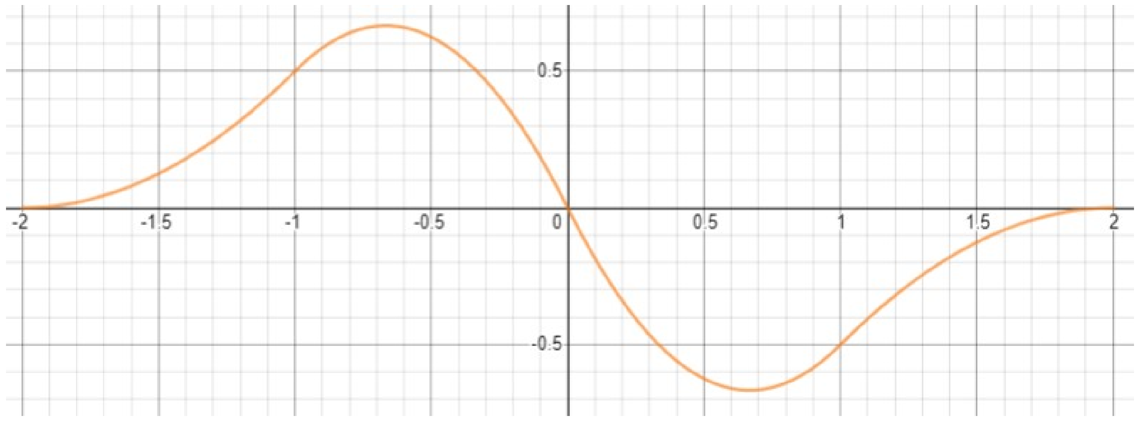
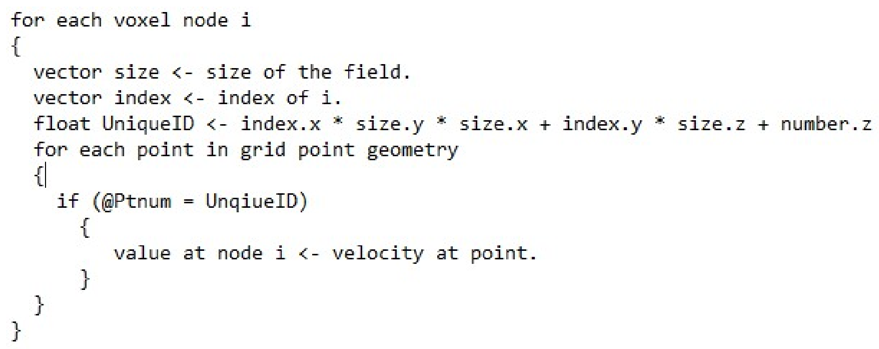


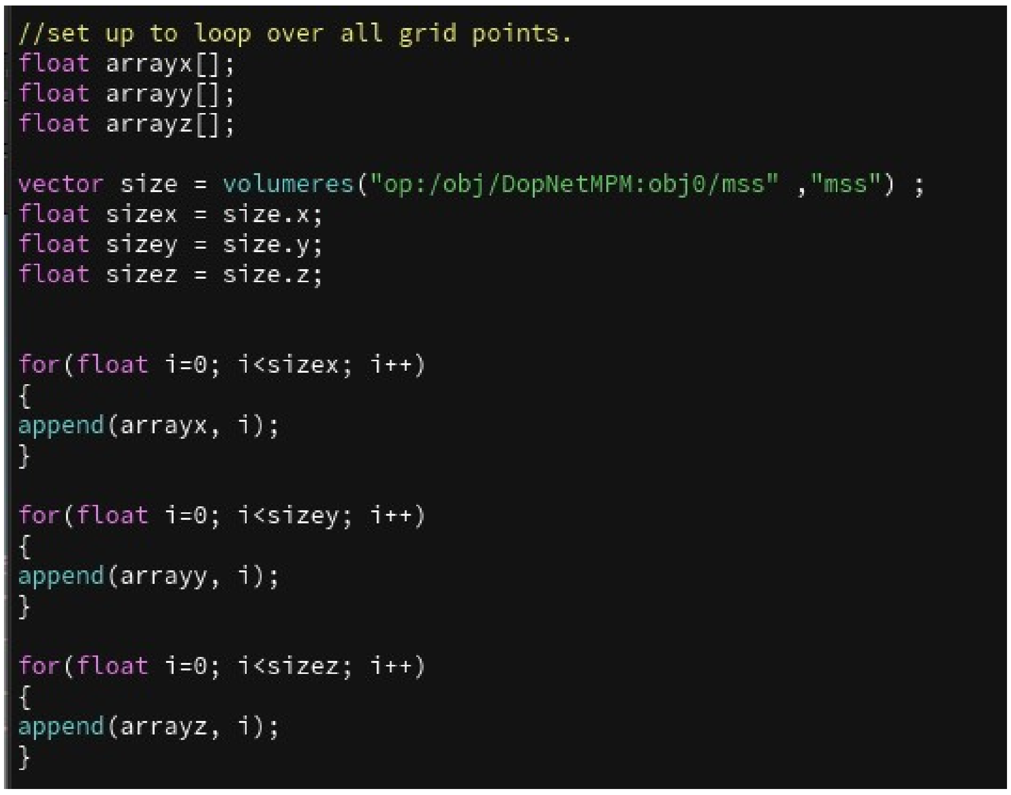
Figure 17: y = N'(x)

B Code



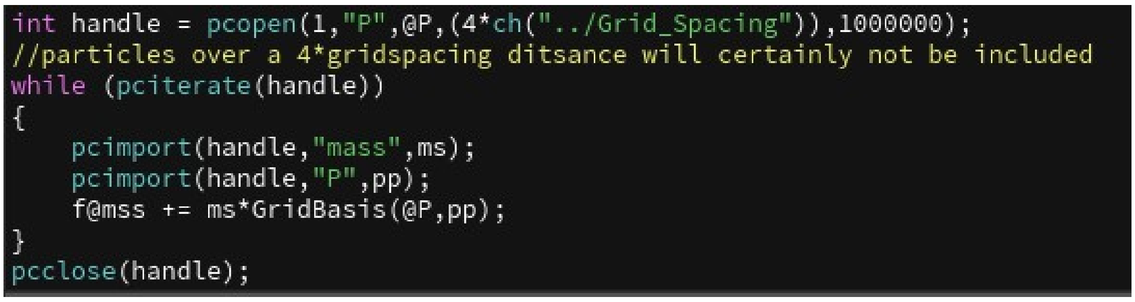
```
for each voxel node i
{
  vector size <- size of the field.
  vector index <- index of i.
  float UniqueID <- index.x * size.y * size.x + index.y * size.z + number.z
  for each point in grid point geometry
  {
    if (@Ptnum = UnqiueID)
      {
         value at node i <- velocity at point.
      }
  }
}
```

Figure 18: Pseudocode for velocity transfer back to grid voxels.



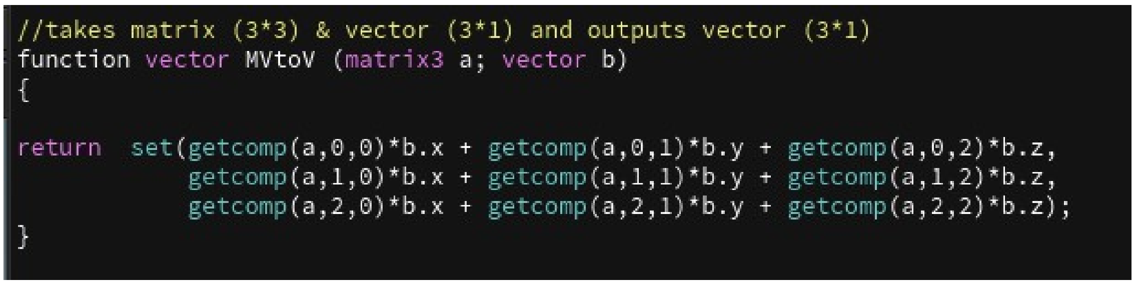
```
//set up to loop over all grid points.
float arrayx[];
float arrayy[];
float arrayz[];

vector size = volumeres("op:/obj/DopNetMPM:obj0/mss" ,"mss") ;
float sizex = size.x;
float sizey = size.y;
float sizez = size.z;


for(float i=0; i<sizex; i++)
{
append(arrayx, i);
}

for(float i=0; i<sizey; i++)
{
append(arrayy, i);
}

for(float i=0; i<sizez; i++)
{
append(arrayz, i);
}
```

Figure 19: Code to access all voxels.



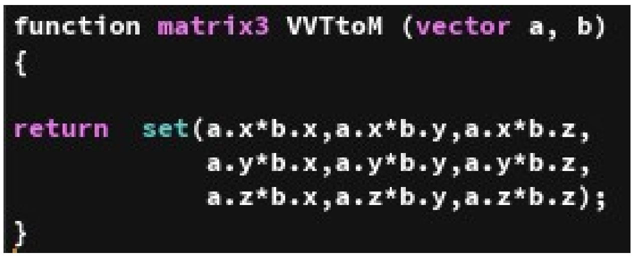
```
int handle = pcopen(1,"P",@P,(4*ch("../Grid_Spacing")),1000000);
//particles over a 4*gridspacing ditsance will certainly not be included
while (pciterate(handle))
{
    pcimport(handle,"mass",ms);
    pcimport(handle,"P",pp);
    f@mss += ms*GridBasis(@P,pp);
}
pcclose(handle);
```

Figure 20: Code to iterate over all points within a 4h spacing.



```
//takes matrix (3*3) & vector (3*1) and outputs vector (3*1)
function vector MVtoV (matrix3 a; vector b)
{

return  set(getcomp(a,0,0)*b.x + getcomp(a,0,1)*b.y + getcomp(a,0,2)*b.z,
            getcomp(a,1,0)*b.x + getcomp(a,1,1)*b.y + getcomp(a,1,2)*b.z,
            getcomp(a,2,0)*b.x + getcomp(a,2,1)*b.y + getcomp(a,2,2)*b.z);
}
```

Figure 21: Matrix and vector multiplication.



```
function matrix3 VVTtoM (vector a, b)
{

return  set(a.x*b.x,a.x*b.y,a.x*b.z,
            a.y*b.x,a.y*b.y,a.y*b.z,
            a.z*b.x,a.z*b.y,a.z*b.z);
}
```

Figure 22: Vector and vector multiplication.

C Simulation Images

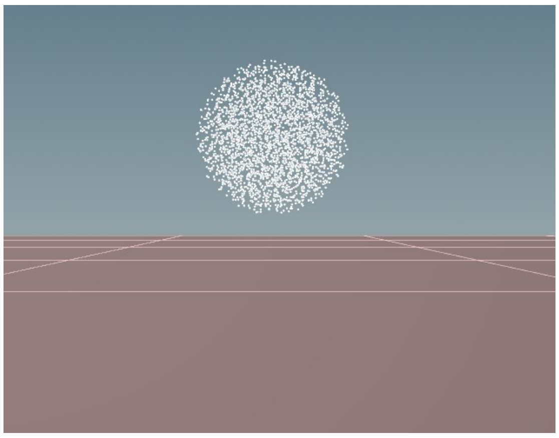


Figure 23: Snowball at the start of a simulation above horizontal plane.

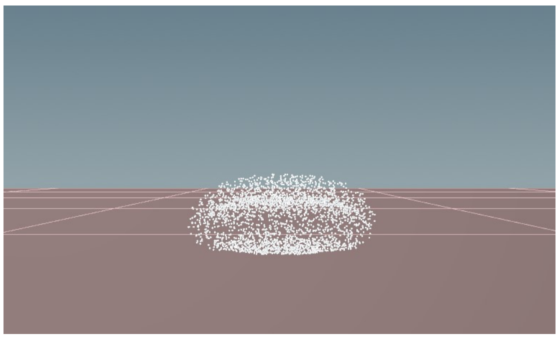


Figure 24: Impact of snowball with the flat plane.

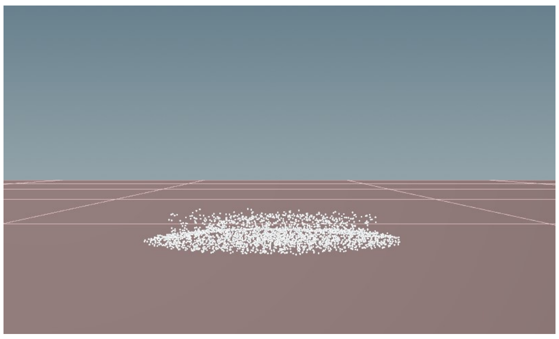


Figure 25: Rest state of the snowball

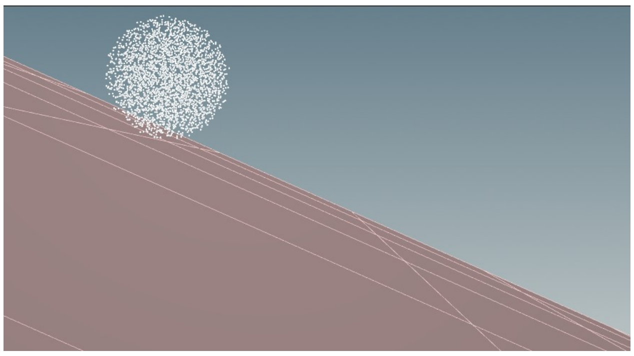


Figure 26: Snowball at the start of a simulation above a slope.

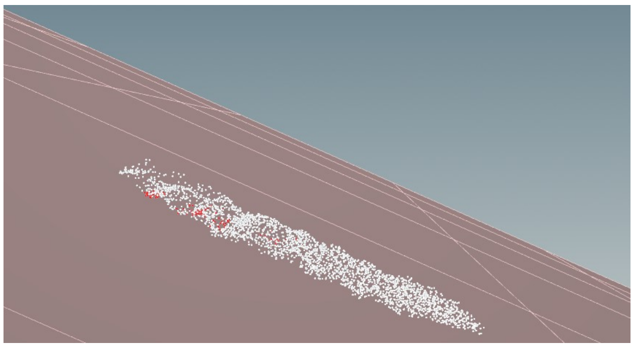


Figure 27: Snowball rest state on the slope.

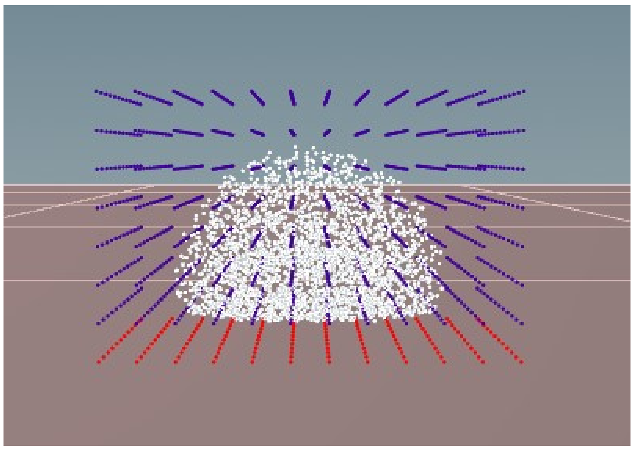


Figure 28: Grid node visualisation turned on.

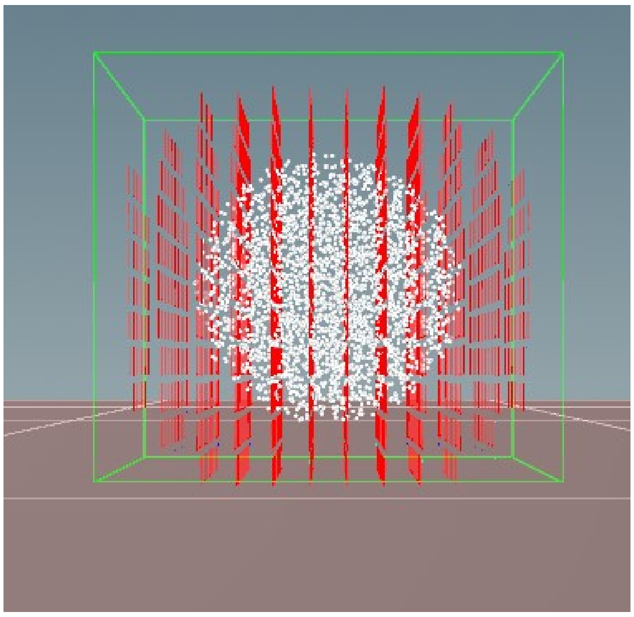


Figure 29: Velocity field visualisation turned on.

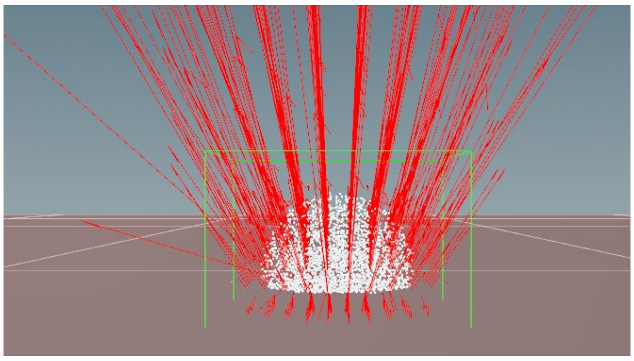


Figure 30: Force field visualisation turned on.

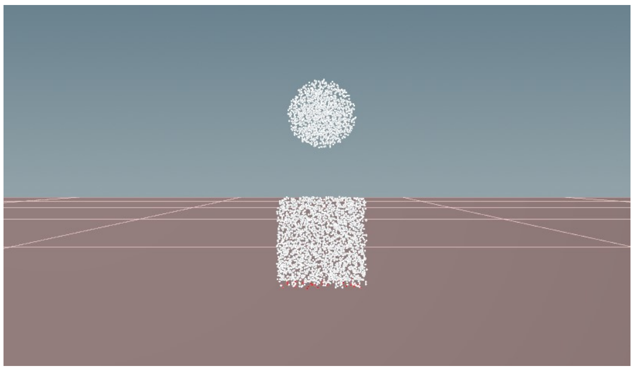


Figure 31: Snow collision start state.

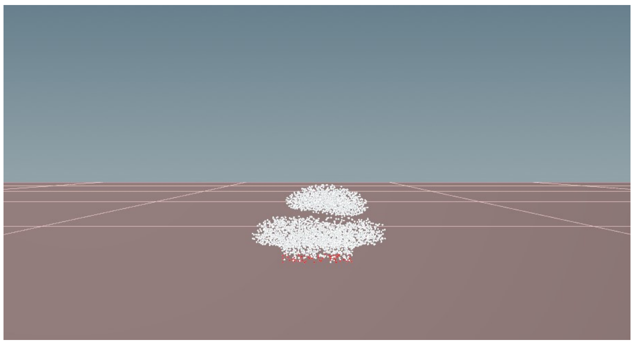


Figure 32: Snow collision end state.