Simulating Melting with the Material Point Method

Ina M. Sørensen

MSc Computer Animation and Visual Effects

22nd of August, 2016

Bournemouth University

Abstract

The Material Point Method is a technique presented by Stomakhin et. al (2013) for use in simulating deformable bodies. It was originially used to model snow for Walt Disney Animation Studio's Frozen and has since been shown to produce good results for a large range of materials behaviours including melting and solidification. The aim of this project was to implement the melting simulation as outlined by Stomakhin et. al (2014) using C++ with import and export from Houdini. The time span of the project allowed for an initial implementation of each part of the technique, including interpolation between particles and grid, deviatoric and projected velocity calculations and temperature updates resulting in a phase change. The results give the effect of a deformable solid and a viscous fluid, however, future work is required to improve the stability and accuracy of the solver.

Acknowledgements

I would like to thank Alexey Stomakhin, Jon Macey and Hammadi Nait-Charif for their help with this project. I would also like to thank everyone of the MSc CAVE students this year for helping me with all of my random coding problems. Finally I would like to thank my family and friends for their support throughout this year.

Contents

Chapter 1

Introduction

Deformable objects have been of interest to the graphics community for more than two decades, as shown by the paper on elasticity in computer graphics by Terzopoulos et. al[1]. Since then a large number of methods have been developed such as the mass-spring model for cloth, the Smooth Particle Hydrodynamics (SPH) model and the physically-based Finite Element Method (FEM), each with their own advantages and disadvantages. Whilst particle-based methods have the advantage of a simple representation of an object, they run into issues due to the need to calculate differentials and integrals without an underlying grid, leading to the use of simplified solutions which are not necessarily physically based. The grid-based FEM method on the other hand suffers from the underlying mesh getting tangled as the object deforms. Similar methods have been explored in fluid simulation and similar issues have been encountered, leading to the development of the Fluid Implicit Particle (FLIP) method. In recent years this has been extended to solids by the introduction of the Material Point Method (MPM).

MPM was first presented in 1995 by Sulsky et. al[2] where it was used for engineering purposes, but it was not until 2013 that the method made its way into graphics through its use in Walt Disney Animation Studio's Frozen. As mentioned, MPM is like a FLIP method for solid objects as it is also a hybrid model using particles to represent the object and its deformation whilst taking advantage of a stationary Eulerian grid to solve the equations determining the deformations. As such, it combines the advantages of both the particle and grid-based methods and has been shown to give good results for a large number of uses, including the simulation of melting which this thesis will focus on.

Chapter 2

Related Work

Physically-based simulations of deformable bodies have become a large field of research as computer graphics has moved from rigid bodies to more realistic materials. As mentioned, the topic was first introduced by Terzopoulos et. al[1] who simulated deformations by applying elastic theory to curves, surfaces and volumes. Since then, the aim has been to model an ever larger range of behaviours, from stiff solids to viscous fluids. Attempts have also been made to make the models more physically correct by applying theories developed in engineering. A large number of different methods have originated from these endeavours which are commonly divided into four main categories; Eulerian grid-based methods, Lagrangian mesh-based methods, mesh-free particle-based methods and particle/grid hybrids.

Simulating melting and solidification is an area within the deformable objects field which has been widely researched. It is a complicated topic as it requires a continuum between solid and fluid behaviours which are usually simulated using two different methods. The topic was first explored by Terzopoulos et. al[3] in which the material was modelled as particles with force interactions laid out in a grid-like structure, similarly to the mass-spring model. As temperature increase, the forces between the particles would relax to give a plastic or flow-like behaviour. Since then, attempts have been made using methods from each of the four categories, often starting in either the solid or liquid end of the spectra.

Goktekin et. al[4] were one of the first to use a Eulerian grid to create viscoelastic fluids which could model materials such as pudding, clay and toothpaste, that is, materials with both solid and fluid characteristics. Although Eulerian grids have been used extensively in fluid simulations, the main difficulty with using Eulerian grids for solids lies in the computation of the equation describing deformation. Hence, more recent efforts using this method have involved simulating the solid as a high viscosity fluid.[5, 6]

The mesh-based FEM method has become a popular method for simulating deformable solids in the graphics community.[7, 8] It is extensively used in engineering due to its ability to accurately solve the continuum equations which describe how a material deforms, however, the methods has limitations in graphics where it is often desired to simulate materials with large deformations. These limitations arise due to the fact that the mesh is used to represent the shape of the object, hence it often becomes entangled when the material has to undergo extensive shape changes. A great deal of work has been done to efficiently remesh objects to improve computations[8, 9, 10, 11], and Clausen et. al[12] were able to adapt these methods to simulate melting using a local, dynamical remeshing FEM. However, mesh entanglement still remains a major issue for FEM simulations.

Mesh-free particle-based methods, such as SPH[13] or Point-Based Dynamics (PBD)[14], have given very good results when applied to fluids and solids. These methods are especially good at simulating interactions between different materials due to the particles storing the material properties such as stiffness, hence, they have also provided very good simulations of melting.[15, 16, 17] The main disadvantage of these methods is that calculation of the derivatives required to solve the physically-based continuum equations is complicated, and approximated versions have to be applied instead. This can lead to accumulation of errors in the elastic and plastic response of the materials.[9]

Another approach to melting was made by Losasso et. al[18] which couples Lagrangian mesh-based methods for solids with Eulerian grid-based methods for fluids.

MPM, the particle/grid hybrid which is the focus of this project, was first presented by Sulsky et. al[2], and recently introduced into the field of computer graphics by Stomakhin et. al.[19] In the years following its introduction, a large number of papers have been published providing further improvements to the method and displaying a growing number of material properties. The most noteworthy improvements are the development of the Affine Particle-In-Cell (APIC) interpolation method[20], which greatly increases the stability of the MPM, and the work by

Klàr et. al[21], which includes a more physically-based interpretation of plasticity and yield.

Chapter 3

The Material Point Method

The MPM method is a grid/particle hybrid similar to FLIP fluids, where the material is made up of particles which store the calculation variables, like position, velocity and temperature. The governing equations are solved on a stationary Eulerian grid, and steps at the beginning and end of a calculation are taken to interpolate the particle data to the grid and to update the particle data from the grid. In this way, MPM is able to combine the advantages of the mesh-based, grid-based and particle-based methods that were outlined in the previous chapter. First of all, MPM is able to solve the same governing equations as FEM, leading to physically accurate simulations. The background Eulerian grid enables easy implementation of boundary conditions, as well as eliminating the need for inter-particle collision detection and response. This is due to particle-particle collisions not occurring when their velocities are obtained from the velocity field of the grid. Finally, the particle-based material representation provides a way to simply deform the material without the issue of mesh entanglement. Furthermore, particle representation also facilitates splitting, merging and coupling between materials with different behaviours, thus it is a good method for simulating melting.

The MPM algorithm has the following steps:

- Transfer data from particles to grid.
- Detect collisions with external objects and classify cells as interior, colliding or empty.
  - The classification is done to avoid unnecessary calculations on empty cells and to ease

    the inclusion of boundary conditions at colliding cells.

- Compute material forces.
- Update grid velocities including boundary conditions.
  - In the case of a melting simulation, this step also includes a pressure calculation to ensure the incompressibility of fluid.
- Update temperature.
  - This step is only necessary in simulations involving temperature dependent material properties.
- Update particle data from grid.
- Advect particles by calculating their new positions based on the updated particle data.

The following sections will describe the details involved in each of these steps. First, a brief introduction to continuum mechanics will be given, including the main concepts used in the implementation. Second, the interpolation steps will be described, before the governing equations which are solved by the grid are presented. The explicit and implicit integration schemes applied to the velocity update are then outlined. Finally, a brief account will be given of the calculations performed in the final step of the algorithm which not only includes advection, but also deformation update and phase transitions. The information in the following sections is primarily based on the articles by Stomakhin et. al[19, 22] and the SIGGRAPH course notes by Jiang et. al[23].

3.1 Continuum Mechanics and Material Deformations

In order to model deformable bodies, a link must be made between the forces applied to an object and its resulting change in shape. This is called a constitutive model, and the choice of model depends on the material behaviour desired. The constitutive model used in this project is based on a few key concepts from continuum mechanics, such as the deformation gradient, stress, elasticity and plasticity, which will be introduced in this section.

3.1.1 Deformation Gradient

Continuum mechanics is the study of deformations in large scale objects. In this field, an object is represented as continuous pieces of matter and its physical quantities, such as density, velocity and forces, are calculated by functions which are continuous with position. Deformation is defined as a change from the rest or undeformed position, $\mathbf{X}$, and current or deformed position, $\mathbf{x}$. The function describing the change from the former to the latter is given by ϕ:

$$\mathbf{x}(\mathbf{X}, t) = \phi(\mathbf{X}, t) \tag{3.1}$$

That is, ϕ describes the change in the positions of the points on an object, leading to the deformed shape. The differentials of this function with respect to time will give velocity and acceleration:

$$\mathbf{V}(\mathbf{X}, t) = \frac{\partial \phi}{\partial t}(\mathbf{X}, t) \tag{3.2a}$$

$$\mathbf{A}(\mathbf{X}, t) = \frac{\partial^2 \phi}{\partial t^2}(\mathbf{X}, t) \tag{3.2b}$$

Here, it is important to make a distinction between Lagrangian and Eulerian viewpoints as eq. 3.1-3.2 are defined in the Lagrangian view. In this viewpoint, a quantity is measured with regards to a specific particle or material point, as opposed to the Eulerian viewpoint, where a quantity is defined relative to a fixed point of reference. For example, in the case of a fluid, the Lagrangian view would mean that the velocity of a specific water particle is measured, whilst the Eulerian view would mean measuring the velocity relative to a set position on a grid. This distinction is important to keep in mind since MPM used both as it moves from the particles to the grid and vice versa.

As mentioned, ϕ gives the change in position of the material points. However, this function will also include rigid body rotations and translations which will not lead to shape changes. The deformation gradient, $\mathbf{F}$, is used to distinguish the deformations that do lead to shape changes and is very useful when simulating deformable bodies. It is a 3x3 matrix in 3D space given by the Jacobian of ϕ:

$$\mathbf{F}(\mathbf{X},t) = \frac{\partial \phi}{\partial \mathbf{X}}(\mathbf{X},t) \tag{3.3}$$

Its determinant, J, describes volume change. The deformation gradient will become important when relating forces to shape changes in the constitutive model.

3.1.2 Elastic and Plastic Deformations

Deformations can be either elastic or plastic. In the elastic regime, deformations are not permanent and the material will snap back to its original shape as soon as the force has been removed. In the plastic regime, the deformations are permanent and can lead to a change in the material properties, such as making it more or less susceptible to future deformations.

In the work presented by Stomakhin et. al[19, 22], an inexact but practical plasticity model is used. In this model, the deformation gradient is separated into an elastic part, $\mathbf{F}_E$, and plastic part, $\mathbf{F}_P$:

$$\mathbf{F} = \mathbf{F}_E\mathbf{F}_P \tag{3.4}$$

The grid then uses an equation which describes the deformation response of a perfectly elastic material to update the elastic part of the deformation gradient. To incorporate plasticity, a check is done to determine whether $\mathbf{F}_E$ is too large to be purely elastic. This is done by performing a Singular Value Decompositon (SVD) on $\mathbf{F}_E$:

$$\mathbf{F}_E = \mathbf{U}\hat{\Sigma}\mathbf{V}^T \tag{3.5}$$

where $\mathbf{U}$ and $\mathbf{V}$ are rotation matrices and $\hat{\Sigma}$ is a diagonal matrix with components which work like the eigenvalues of $\mathbf{F}_E$, that is, by setting the scale of the deformation. The components of $\hat{\Sigma}$ are then clamped to stretch and compression limits as a way of imposing a yield criterion, which is used to determine whether the force applied to a material will result in plastic deformations. Once this has been done, the new $\mathbf{F}_P$ has to be determined so as to place the excess elastic deformation in the plastic deformation gradient. Since the overall deformation gradient, $\mathbf{F}$, does not change in this calculation, the new $\mathbf{F}_P$ can be determined as follows:

$$\mathbf{F}_P = \mathbf{V}\Sigma^{-1}\mathbf{U}^T\mathbf{F} \tag{3.6}$$

where Σ is the clamped version of $\hat{\Sigma}$.

As mentioned, this is a very simplistic model of plasticity and yield. In reality, yield criteria tend to take the direction of the force into account since some materials are much stronger under compression than extension or are much weaker along certain axes. Some improvements on the plasticity model for the MPM have been made, such as the work presented by Klàr et. al[21] which implements a more realistic yield criterion for sand.

3.1.3 Constitutive Models

A constitutive model is the relation between an applied force and the resulting shape change, or, more correctly, the relation between stress and strain. Stress is given by force over area and describes a field set up inside the object as a result of the forces applied to the object. It appears as a result of the atoms moving out of their ideal positions and hence setting up attractive or repelling forces in an attempt to return to their original positions. Strain is a measure of the deformation of a material and can be exchanged for the deformation gradient.

There are a large number of constitutive models available as different material groups, such as metals, plastics and glass, tend to respond very differently to applied loads. The constitutive model used in this project is defined for hyperelastic materials, that is, materials which are perfectly elastic. This model was chosen since the MPM assumes a perfectly elastic material for the grid calculations and applies the plastic deformation later. By choosing the hyperelastic material model, the constitutive relation can be expressed through a potential strain energy density function, $\Psi(\mathbf{F})$, which increases with non-rigid deformations. This function can be related to Cauchy stress, σ, as follows:

$$\sigma = \frac{1}{J}\frac{\partial \Psi}{\partial \mathbf{F}}\mathbf{F}^T \tag{3.7}$$

where Cauchy stress is just one of many possible stress definition, and J is the determinant of **F**.

The next step is to choose a model for $\Psi(\mathbf{F})$ within the hyperelastic regime. In the work by Stomakhin et. al[19, 22], a Fixed Co-rotated Energy Density function is chosen which is a simple but widely used model that combines the linear elastic model with some non-linearity to avoid rigid body rotations affecting the energy density:

$$\Psi(\mathbf{F}_E) = \Psi_\mu(\mathbf{F}_E) + \Psi_\lambda(J_E) \tag{3.8a}$$

$$\Psi_\mu(\mathbf{F}_E) = \mu\|\mathbf{F}_E - \mathbf{R}_E\|_E^2 \tag{3.8b}$$

$$\Psi_\lambda(J_E) = \frac{\lambda}{2}(J_E - 1)^2 \tag{3.8c}$$

where $\mathbf{R}_E$ is the rotation matrix from the polar decomposition of $\mathbf{F}_E$ which can be defined as follows:

$$\mathbf{F}_E = \mathbf{R}_E\mathbf{S}_E \tag{3.9a}$$

$$\mathbf{R}_E = \mathbf{U}\mathbf{V}^T \tag{3.9b}$$

$$\mathbf{S}_E = \mathbf{V}\Sigma\mathbf{V}^T \tag{3.9c}$$

where $\mathbf{S}_E$ is a symmetric matrix, $\mathbf{R}_E$ is a rotation matrix, and **U**, **V** and Σ come from the SVD decomposition as shown in eq. 3.5. A more detailed description of the model can be found in the SIGGRAPH course notes by Jiang et. al[23].

μ and λ in eq. 3.8 are the Lame constants which can be determined from elastic modulus, E, and Poisson ratio, ν as follows:

$$\mu = \frac{E}{2(1+\nu)} \tag{3.10a}$$

$$\lambda = \frac{\nu E}{(1+\nu)(1-2\nu)} \tag{3.10b}$$

The elastic modulus, E, is a measure of how stiff a material is or how much elastic deformation will result from a given force in the direction of the force. The Poisson ratio, ν, then relates how much the material will compress along the other directions when a force is applied to extend the material in the direction of the force. Stomakhin et. al[19] used the Lame constants to include the effect of material hardening. By setting the constants above to μ_0 and λ_0, they redefined the constants as given below:

$$\mu = \mu_0 exp(\xi(1 - J_P)) \tag{3.11a}$$

$$\lambda = \lambda_0 exp(\xi(1 - J_P)) \tag{3.11b}$$

where ξ is the hardening coefficient and J_P is the determinant of the plastic deformation gradient. These definitions would allow a material to reduce stiffness when it is undergoing plastic stretch and increase stiffness when it is undergoing plastic compression.

As seen in eq. 3.7, stress is related to the partial differential of the energy density function. It will therefore be useful to derive the differential of Ψ since this will be used in the force calculation later.

$$\frac{\partial \Psi}{\partial \mathbf{F}}(\mathbf{F}_E) = 2\mu(\mathbf{F}_E - \mathbf{R}_E) + \lambda(J_E - 1)J_E\mathbf{F}_E^T \tag{3.12}$$

With all these definitions at hand, it is now possible to derive the governing equation for the change in velocity as will be done in section 3.3.

3.2 Data Interpolations

At the beginning and end of the simulation, particle data has to be transferred to and from the grid. This requires the use of interpolation weights, ω_{ip}, and an interpolation function, N(x). This section will look at how to calculate the interpolation weights for the grid based on the interpolation function, which interpolation functions should be chosen, and finally, how data is transferred to and from the grid using the interpolation weights. First, however, a naming convention will be set to enable the distinction between particle and grid variables, and the time frame of the variables.

3.2.1 Naming Convention

In this implementation, a staggered MAC grid is used with velocity and heat conductivity defined at the cell faces and all other quantities defined at the cell centres. For this reason it is important to distinguish between tensor, ie. matrices and vectors, quantities and scalars as a grid cell velocity can be defined as an overall vector and as a scalar at each face. This distinction is made by using bold letters for tensors. Furthermore, cell face variables are marked by α to say that the quantity is defined in x, y and z direction, whilst cell centre variables are marked by c. All variables that belong to a particle is given a subscript p and those belonging to cells are given subscript i. The i stands for the index of the cell in the grid; i=[i,j,k]. Finally, some quantities are given a superscript to determine the time frame of the value; n means it is the value from the previous time step, n+1 is the final value for the current time step and * is given to intermediate values.

3.2.2 Interpolation Weights

In MPM, the interpolation weight between a cell and a particle is given by:

$$\omega_{ip} = N^h_{i\alpha}(\mathbf{x}_p) = N(\frac{1}{h}(x_p - x_{i\alpha})) \cdot N(\frac{1}{h}(y_p - y_{j\alpha})) \cdot N(\frac{1}{h}(z_p - z_{k\alpha})) \tag{3.13}$$

where h is the length of the side of a cell, $\mathbf{x}_p = [x_p, y_p, z_p]$ is the position of the particle, and $\mathbf{x}_{i\alpha} = [x_{i\alpha}, y_{j\alpha}, z_{k\alpha}]$ is the position of the grid cell with index i,j,k. The weight gradient is also required and is defined as:

$$\nabla\omega_{ip} = \begin{pmatrix} \frac{1}{h}\frac{dN}{dx}(\frac{1}{h}(x_p - x_{i\alpha})) \cdot N(\frac{1}{h}(y_p - y_{j\alpha})) \cdot N(\frac{1}{h}(z_p - z_{k\alpha})) \\ N(\frac{1}{h}(x_p - x_{i\alpha})) \cdot \frac{1}{h}\frac{dN}{dy}(\frac{1}{h}(y_p - y_{j\alpha})) \cdot N(\frac{1}{h}(z_p - z_{k\alpha})) \\ N(\frac{1}{h}(x_p - x_{i\alpha})) \cdot N(\frac{1}{h}(y_p - y_{j\alpha})) \cdot \frac{1}{h}\frac{dN}{dz}(\frac{1}{h}(z_p - z_{k\alpha})) \end{pmatrix} \tag{3.14}$$

On the staggered MAC grid, the cell positions, $\mathbf{x}_{i\alpha}$, are given by:

Cell centre or face	Position vector
c	[i*h, j*h, k*h]
x	[(i - 1/2)*, j*h, k*h]
y	[i*h, (j - 1/2)*h, k*h]
z	[i*h, j*h, (k -1/2)*h]

3.2.3 Interpolation Functions

The interpolation function, N(x), is chosen based on smoothness, efficiency and width, along with two requirements set by the MPM method. Firstly, $\frac{dN(x)}{dx}$ must be continuous as it will be used in the force calculations, otherwise the grid forces will not be continuous. Secondly, $\frac{dN(x)}{dx}$ must go to zero as N(x) goes to zero to ensure that cells with few particles affecting it will also contribute small forces to the grid. For these reasons, a cubic B-Spline has been used by Stomakhin et. al[19, 22]:

$$N(x) = \begin{matrix} \frac{1}{2}|x|^3 - x^2 + \frac{2}{3} & \text{for} & 0 \leq |x| < 1 \\ -\frac{1}{6}|x|^3 + x^2 - 2|x| + \frac{4}{3} & \text{for} & 1 \leq |x| < 2 \\ 0 & & \text{otherwise} \end{matrix} \tag{3.15}$$

The effect of the cubic B Spline can be seen in figure 3.1 where it is clear that a particle will affect the cell its in, the layer of nearest neighbour cells around it, and the corner of the second nearest neighbour layer that is closest to the particle.

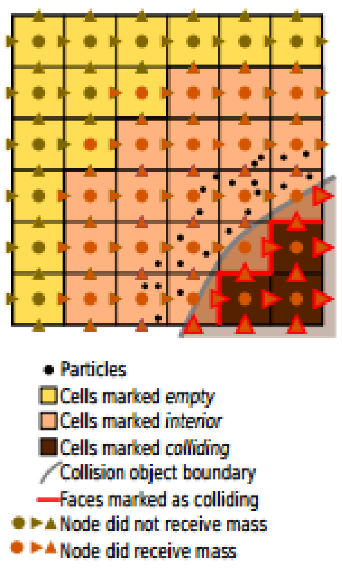


Figure 3.1: The cells affected by the cubic B-Spline are marked in orange and the empty cells are marked by brown. The figure also displays the cell classification scheme. Reproduced from Stomakhin et. al[22]

As will be seen later in this chapter, a pressure calculation will be performed to update the face velocities of the grid, however, this calculation will only be performed at cells which are not empty. Figure 3.2 shows how this causes certain cell faces that are reached by the cubic B-Spline to be uncorrected by the pressure calculation. To prevent these faces from contributing to the particle updates, a different interpolation function is used when transfering data back to the particles, namely a tighter Quadratic stencil:

$$N(x) = \begin{array}{ll} -x^2 + \frac{2}{3} & \text{for} \quad 0 \leq |x| < \frac{1}{2} \\ \frac{1}{2}x^2 - \frac{3}{2}|x| + \frac{9}{8} & \text{for} \quad \frac{1}{2} \leq |x| < \frac{3}{2} \\ 0 & \quad \text{otherwise} \end{array} \tag{3.16}$$

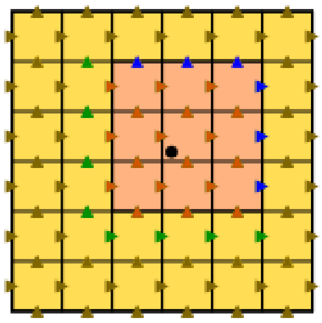

● Reference particle
🟨 Cells whose pressures not corrected
🟧 Cells whose pressures corrected
Node received masses / corrected / not used
Node received mass / not corrected / not used
Node received no mass / not corrected /not used
Node received mass / corrected / used

Figure 3.2: The cells affected by the tighter Quadratic Stencil are shown in orange, green and blue are the ones affected by the cubic B-Spline but not the tight Quadratic Stencil, and brown are the ones affected by neither. Reproduced from Stomakhin et. al[22]

3.2.4 Moving data between grid and particles

In the first implementation of the MPM method[19, 22], particle data was transferred to the grid accoring to the following equations:

$$m_\alpha^n = \sum_p m_p \omega_{\alpha p}^n \tag{3.17a}$$

$$m_c^n = \sum_p m_p \omega_{cp}^n \tag{3.17b}$$

$$A_\alpha^n = \frac{1}{m_\alpha^n} \textstyle\sum_p A_p^n m_p \omega_{\alpha p}^n \quad \text{for} \quad A \in \{v, \kappa\} \tag{3.17c}$$

$$B_c^n = \frac{1}{m_c^n} \textstyle\sum_p B_p^n m_p \omega_{cp}^n \quad \text{for} \quad B \in \{J, J_E, c, T, \lambda^{-1}\} \tag{3.17d}$$

However, it has been shown by Jiang et. al[20] that this method for transfering particle data, especially the particle velocities, will lead to instabilities in the simulation. Instead, the APIC method should be used. This method will not be described here as it was not implemented in the project, but a full outline can be found in their paper.

As outlined in the same paper, the main ways to update the particles from the grid include PIC, FLIP, PIC/FLIP mixes and APIC. Again, the APIC method has been shown to give the most stable results, as well as providing the best means of maintaining angular velocity. However, in this project, the PIC/FLIP mix method outlined in the paper by Stomakhin et. al[22] was used. In this implementation, the variables are updated according to the following equations:

$$A_p^{n+1} = \beta A_p^{FLIP} + (1-\beta)A_p^{PIC} \quad \text{for} \quad A \in \{\mathbf{v}, T\} \tag{3.18a}$$

$$A_p^{FLIP} = A_p^n + \sum_i (A_i^{n+1} - A_i^n)\omega_{ip} \tag{3.18b}$$

$$A_p^{PIC} = A_p^n + \sum_i A_i^{n+1}\omega_{ip} \tag{3.18c}$$

where β is a constant giving the degree of FLIP in the mix. Stomakhin et. al[22] uses a value of 0.95.

The velocity gradient is also required and is updated by:

$$\nabla \mathbf{v}_p^{n+1} = \sum_i v_\alpha^{n+1}\mathbf{e}_\alpha[\nabla\omega_{\alpha p}]^T \tag{3.19}$$

where $\mathbf{e}_\alpha$ is the unit vector in the direction of the face normal.

With the presentation of APIC, it was also found that when FLIP or FLIP/PIC mixes are used, the particle positions should be updated directly; $\mathbf{x}_p^{n+1} = \sum_i \mathbf{x}_i^{n+1}\omega_{ip}$. Here, $\mathbf{x}_i^{n+1}$ is the fictional new position of a grid cell since the grid does not move but its updated position is used to update the particles.

3.3 The Governing Equations

The three governing equations for the MPM melting simulation are the conservation of mass where ρ is density;

$$\frac{D\rho}{Dt} = 0 \tag{3.20}$$

the conservation of monentum, where $\mathbf{v}$ is velocity, σ is the Cauchy stress and $\mathbf{g}$ is the sum of the external forces;

$$\rho \frac{D\mathbf{v}}{Dt} = \nabla \cdot \sigma + \rho \mathbf{g} \tag{3.21}$$

and the conservation of energy:

$$\rho \frac{Du}{Dt} = -\nabla \cdot \mathbf{q} \tag{3.22}$$

where u is the internal energy of the object and q is the heat transfered to or from the object.

The details of each of these functions will be given in separate sections along with descriptions of how they are solved by the MPM.

3.3.1 Conservation of Mass

Conservation of mass is fairly simple to enforce in MPM due to the use of particles. Each particle is initially given a mass which is unaltered throughout the simulation. However, since eq. 3.20 is based on the density, it is also necessary to calculate the constant particle density. This is done in the first step of the simulation by weighting the cell density back to each particle:

$$\rho_p^0 = \sum_c \frac{m_c^0 \omega_{cp}^0}{h^3} \tag{3.23}$$

As long as m_p and ρ_p stay constant throughout the simulation, the conservation of mass is obtained.

3.3.2 Conservation of Momentum

The conservation of momentum, eq. 3.21, is used to update the velocities and the deformation gradient, and hence needs to be discretised in order to be solved on the grid. The derivation of the discretised equation uses a weak form of the force balance, similarly to how the FEM equations are determined, and the full derivation can be found in literature[2, 23]. The result is the following equation which relates the internal force in the material to the deformation gradient:

$$f_{i\alpha}(\hat{\mathbf{x}}) = -\sum_p V_p^0 \mathbf{e}_\alpha^T \frac{\partial \Psi}{\partial \mathbf{F}_E}(\hat{\mathbf{F}}_{Ep}(\hat{\mathbf{x}}))[\mathbf{F}_{Ep}^n]^T \nabla \omega_{i\alpha p}^n \tag{3.24}$$

where V_p^0 is the constant particle volume, $\mathbf{e}_\alpha$ is a unit vector in the direction of the face normal, ie $[1, 0, 0]$ for the x face, and $\hat{\mathbf{x}}$ is the deformed grid position which is a fictional variable. $\frac{\partial \Psi}{\partial \mathbf{F}_E}(\hat{\mathbf{F}}_{Ep}(\hat{\mathbf{x}}))$ is as given in eq. 3.12. The force calculated is for an individual face. How this force is used to update the grid velocity depends on the integration method as will be outlined in section 3.4.

Pressure Splitting

In the snow simulation presented by Stomakhin et. al[19] the above equation is the only one required to update the velocity. However, it will not enfore incompressibility as is required in fluids. Thus, in the paper on melting simulations[19], the velocity update is split in two. This is done by noticing that stress can be separated into deviatoric stress which is related to shape change and hydrostatic stress which is related to changes in volume.

$$\sigma = \sigma_{dev} + \sigma_{hyd} \tag{3.25}$$

In fact, hydrostatic stress can be directly related to pressure; $\sigma_{hyd} = -p\mathbf{I}$.

In a similar way, it is possible to split the consitutive model in two as seen in eq. 3.8a where Ψ_λ is only dependent on volume change and Ψ_μ is related to deviatoric deformations. The idea is to first calculate an intermediate velocity, $\mathbf{v}^*$, using deviatoric stress, or Ψ_μ, and then calculate the final velocity by including a pressure calculation based on Ψ_λ.

The main issue is that Ψ_μ is not completely orthogonal to Ψ_λ, meaning that Ψ_μ also affects the volume change and hence will not become equal to zero when calculated based on a deformation gradient for volume change. To remedy this, two steps are taken. The first is to remove any deviatoric elastic deformations in fluid, as fluids are almost perfectly plastic with regards to deviatoric deformations. Since the parts of $\mathbf{F}_E$ that relate volume change are given by $(J_E)^{1/3}\mathbf{I}$, the deviatoric components of $\mathbf{F}_E$ are simply removed by setting $\mathbf{F}_{Ep} = (J_{Ep})^{1/3}\mathbf{I}$ in the fluid particles. Secondly, to avoid calculating Ψ_μ for fluid particles, only the deviatoric components of $\mathbf{F}_E$ are calculated for by redefining $\Psi_\mu(\mathbf{F}_E)$ to $\Psi_\mu(J_E^{-1/3}\mathbf{F}_E)$ where $J_E^{-1/3}\mathbf{F}_E$ is the deviatoric component of $\mathbf{F}_E$. In addition, μ is set to zero for fluid particles.

The force equation from eq. 3.24 now becomes:

$$f_{i\alpha} = -\sum_p V_p^0 \mathbf{e}_\alpha^T [\frac{\partial \Psi_\mu}{\partial (J_E^{-1/3}\mathbf{F}_E)}(J_{Ep}^{-1/3}\mathbf{F}_{Ep}) : \frac{\partial (J_E^{-1/3}\mathbf{F}_E)}{\partial \mathbf{F}_E}][\mathbf{F}_{Ep}^n]^T \nabla \omega_{i\alpha p}^n \tag{3.26}$$

More details on how to solve this equation will be given in section 3.4.

Pressure Calculation

As described, Ψ_λ can be related to hydrostatic stress which again can be related to pressure. From this, a pressure equation can be derived following the steps outlined by Stomkahin et. al[19] which is fairly similar to the pressure equation which is used in most grid-based fluid simulations[24, 25]:

$$\frac{J_{Pc}^n}{\lambda_c^n J_{Ec}^n \Delta t} p_c^{n+1} - \Delta t \sum_\alpha \sum_{c'} \frac{1}{\rho_{\alpha c}^n} \mathbf{G}_{\alpha c} \mathbf{G}_{\alpha c'} p_{c'}^{n+1} = -\frac{J_{Ec}^n - 1}{\Delta t J_{Ec}^n} - \sum_\alpha \mathbf{G}_{\alpha c} v_\alpha^* \tag{3.27}$$

where v_α^* is the intermediate velocity which is calculated based on the deviatoric Ψ_μ. As λ_c increases when the object is compressed, it will eventually make the first term on the left hand side negligible. J_{Ec} will also become 1 for fluids, and hence the equation becomes the pressure equation for fluids.

$\mathbf{G}_{\alpha c}$ is the central gradient difference and $\mathbf{G}_{\alpha c}\mathbf{G}_{\alpha c'}$ is the second order central gradient as follows:

$$\sum_\alpha \mathbf{G}_{\alpha c} v_\alpha^* = \frac{v_{i+1/2,j,k}^* - v_{i-1/2,j,k}^*}{\Delta x} + \frac{v_{i,j+1/2,k}^* - v_{i,j-1/2,k}^*}{\Delta y} + \frac{v_{i,j,k+1/2}^* - v_{i,j,k-1/2}^*}{\Delta z} \tag{3.28a}$$

$$
\begin{aligned}
\sum_{\alpha} \frac{1}{\rho^n_{\alpha c}} \mathbf{G}_{\alpha c} \mathbf{G}_{\alpha c'} p_{c'} = {} & (\frac{p_{i,j,k} - p_{i+1,j,k}}{\rho_{i+1/2,j,k} \Delta x^2}) + (\frac{p_{i,j,k} - p_{i-1,j,k}}{\rho_{i-1/2,j,k} \Delta x^2}) \\
& + (\frac{p_{i,j,k} - p_{i,j+1,k}}{\rho_{i,j+1/2,k} \Delta y^2}) + (\frac{p_{i,j,k} - p_{i,j-1,k}}{\rho_{i,j-1/2,k} \Delta y^2}) \\
& + (\frac{p_{i,j,k} - p_{i,j,k+1}}{\rho_{i,j,k+1/2} \Delta z^2}) + (\frac{p_{i,j,k} - p_{i,j,k-1}}{\rho_{i,j,k-1/2} \Delta z^2})
\end{aligned} \tag{3.28b}
$$

where the velocities and densities are defined at the cell faces, and Δx, Δy and Δz are the lenght of the sides of a cell.

The pressure equation makes up a linear system where the matrix is symmetric positive-definite. Dirichlet boundaries are enforced at empty cells and Neumann boundaries are used at faces adjacent to colliding cells.

Once this has been solved to determine the pressures, the projected velocities can be calulated according to:

$$
v^{n+1}_{i\alpha} = v^*_{i\alpha} - \Delta t \sum_c \frac{1}{\rho^n_{\alpha c}} \mathbf{G}_{\alpha c} p_c \tag{3.29}
$$

Control volume

The density in the pressure calculation requires a control volume to be determined instead of the normal cell volume, h^3, because the volume associated with a face adjacent to a colliding celll should be smaller than the volume of an interior cell face. The control volume is determined by

$$
V^n_\alpha = \sum_c \chi_c \int_{\Omega_c} N^h_{c\alpha}(\mathbf{x}) d\mathbf{x} \tag{3.30}
$$

where χ_c is set to zero for all colliding cells and $\int_{\Omega_c} N^h_{c\alpha}(\mathbf{x}) d\mathbf{x}$ is tabulated as follows:

$$
\int_{\Omega_c} N^h_{c\alpha}(\mathbf{x}) d\mathbf{x} = \int_{\Omega_c} N^h_{c\alpha}(x) dx \cdot \int_{\Omega_c} N^h_{c\alpha}(y) dy \cdot \int_{\Omega_c} N^h_{c\alpha}(z) dz \tag{3.31a}
$$

$$\int_{\Omega_c} N^h_{c\alpha}(x)dx = \begin{array}{ccc} h\int_0^1 N(u)du & \text{for} & i=0 \\ h\int_1^2 N(u)du & \text{for} & i=1 \\ 0 & \text{for} & i=2 \\ h\int_0^1 N(u)du & for & i=-1 \\ h\int_1^2 N(u)du & for & i=-2 \end{array} \tag{3.31b}$$

$$\int_{\Omega_c} N^h_{c\alpha}(x)dx = \begin{array}{ccc} 2h\int_0^{0.5} N(u)du & \text{for} & i=0 \\ h(\int_{0.5}^1 N(u)du + \int_1^{1.5} N(u)du) & \text{for} & i=1 \\ h\int_{1.5}^2 N(u)du & \text{for} & i=2 \\ h(\int_{0.5}^1 N(u)du + \int_1^{1.5} N(u)du) & for & i=-1 \\ h\int_{1.5}^2 N(u)du & for & i=-2 \end{array} \tag{3.31c}$$

Depending on which face is being calculated for, eq. 3.31b is used for the integration over the direction of the face and eq. 3.31c is used for the other two directions. That is, for cell face in x direction, eq. 3.31b would be used for $\int_{\Omega_c} N^h_{c\alpha}(x)dx$ and eq. 3.31c would be used for $\int_{\Omega_c} N^h_{c\alpha}(y)dy$ and $\int_{\Omega_c} N^h_{c\alpha}(z)dz$. i is an increment which is different for x, y and z. For example, if the control volume of cell i,j,k is being determined and $\int_{\Omega_c} N^h_{c\alpha}(\mathbf{x})d\mathbf{x}$ is being calculated for the neighbour cell in the x direction, that is, for cell i+1,j,k, then i=1 for the x direction and i=0 for the y and z direction.

Once the control volume has been determined for all the cell faces of all the interior cells, the pressure calculation can be performed to project the velocities.

3.3.3 Conservation of Energy

The final governing equation is the conservation of energy, given by eq. 3.22. By including the definitions of heat transfer and heat capacity, an equation involving temperature can be derived as shown by Stomakhin et. al[22]:

$$T_c^{n+1} + \Delta t h^3 \sum_\alpha \sum_{c'} \frac{\kappa_\alpha^n}{m_c^n c_c^n} \mathbf{G}_{\alpha c} \mathbf{G}_{\alpha c'} T_c^{n+1} = T_c^n \tag{3.32}$$

where κ_α^n is the heat conductivity of face α and c_c^n is the heat capacity at the cell centre. This equation, like the pressure equation makes up a linear system with a symmetric positive-definite matrix. A Dirichlet boundary is enforces at the colliding cells which have fixed temperatures and Neumann boundaries are enforced at empty cells.

3.4 Explicit and Implicit Integration

The time integration step is done to calculate the intermediate velocities from the deviatoric forces as outlined in section 3.3.2. This can be done explicitly, which provides the same accuracy, or implicitily, which allows larger time steps and hence gives a reduction in calculation time.

3.4.1 Explicit Time Integration

In the case of an explicit update, the new velocity is calculated according to:

$$v^*_{i\alpha} = v^n_{i\alpha} + \frac{\Delta t}{m_{i\alpha}} f_{i\alpha} + \Delta t g_{i\alpha} \Sigma_p \omega^n_{i\alpha p} \tag{3.33}$$

where $f_{i\alpha}$ is given by eq. 3.26 and $g_{i\alpha}$ is the sum of the external forces applied to the face α of cell i. Several different boundary conditions can be used at the colliding cell faces such as sticky, separating and slipping as described by Klàr et. al[21]. In the case of explicit time integration, these are applied by setting the velocities at the colliding faces to specific values.

3.4.2 Implicit Time Integration

The implicit update is fairly complicated to implement as the equation for the force derivative is quite complex and the final system produces a symmetric semi-definite matrix, meaning that the common Conjugate Gradient solver might not give the correct solutions.

In the case of the melting simulation described by Stomakhin et. al[22], the force derivative is given by the following equations:

$$A_{ij} = \delta_{ij} + \frac{\Delta t^2}{2m_i} \Sigma_p V^0_p \mathbf{e}^T_\alpha \mathbf{A}_p [F_{Ep}]^T \nabla \omega^n_{i\alpha p} \tag{3.34a}$$

$$\mathbf{A}_p = \frac{\partial^2 \hat{\Psi}_\mu}{\partial \mathbf{F}_E^2}(\mathbf{F}_E(\hat{\mathbf{x}})) : (\mathbf{e}_\alpha [\nabla \omega_{j\alpha p}^n]^T \mathbf{F}_{Ep}^n) \tag{3.34b}$$

where A_{ij} is the matrix element in row i and column j, $\hat{\Psi}_\mu(\mathbf{F}_E) = \Psi_\mu(J_E^{-1/3}\mathbf{F}_E)$, and $\hat{\mathbf{x}}$ is the deformed grid position. $\mathbf{A}_p$ can be further derived based on the technical report accompanying the melting simulation paper[22]:

$$\mathbf{A}_p = (C : (B : Z)) : B + a(H : Z)A : B + aJ^a(A : Z)H - aJ^a(A : F)HZ^T H \tag{3.35}$$

where a = -1/3, A is $\frac{\partial \Psi_\mu}{\partial \mathbf{F}_E}$ as given in eq. 3.12, B is the differential of $J_E^{-1/3}\mathbf{F}_E$ which will be given below, F is $\mathbf{F}_E$, H is $\mathbf{F}_E^{-T}$, J is J_E and Z is the term on the right hand side of : in eq. 3.34b. The B component arises due to the fact that, with pressure splitting, $\Psi(\mathbf{F}_E)$ is now redefined to $\Psi(J_E^{-1/3}\mathbf{F}_E)$, thus $\frac{\partial \Psi}{\partial \mathbf{F}_E}(\mathbf{F}_E)$ becomes $\frac{\partial \Psi}{\partial \mathbf{F}_E}(J_E^{-1/3}\mathbf{F}_E) = \frac{\partial \Psi}{\partial J_E^{-1/3}\mathbf{F}_E}(J_E^{-1/3}\mathbf{F}_E) : \frac{\partial J_E^{-1/3}\mathbf{F}_E}{\partial \mathbf{F}_E}$.

In the same way, C:(B:Z), where C is $\frac{\partial^2 \Psi}{\partial \mathbf{F}_E^2}(\mathbf{F}_E)$, can be given by

$$\begin{aligned} C : (B : Z) &= \frac{\partial^2 \Psi}{\partial (J_E^{-1/3}\mathbf{F}_E)^2}(J_E^{-1/3}\mathbf{F}_E) : \delta(J_E^{-1/3}\mathbf{F}_E) \\ &= 2\mu\delta(J_E^{-1/3}\mathbf{F}_E) - 2\mu\delta\mathbf{R} \end{aligned} \tag{3.36a}$$

$$\delta(J_E^{-1/3}\mathbf{F}_E) = B : \delta\mathbf{F}_E \tag{3.36b}$$

$$\delta\mathbf{F}_E = \mathbf{e}_\alpha [\nabla \omega_{j\alpha p}^n]^T \mathbf{F}_{Ep}^n \tag{3.36c}$$

Here, $\delta\mathbf{R}_E$ is the rotational part of the polar decomposition of $\Psi(J_E^{-1/3}\mathbf{F}_E)$. Due to the fact that $\mathbf{R}_E^T\delta\mathbf{R}_E$ is skew-symmetric, ie. $[\mathbf{R}_E^T\delta\mathbf{R}_E]^T = -\mathbf{R}_E^T\delta\mathbf{R}_E$, it can be written as:

$$\mathbf{R}_E^T\delta\mathbf{R}_E = \begin{pmatrix} 0 & x & y \\ -x & 0 & z \\ -y & -z & 0 \end{pmatrix} \tag{3.37}$$

and determined using the following definition:

$$\mathbf{R}_E^T \delta \mathbf{F}_E - \delta \mathbf{F}_E^T \mathbf{R}_E = (\mathbf{R}_E^T \delta \mathbf{R}_E)\mathbf{S}_E + \mathbf{S}_E^T \delta \mathbf{R}_E^T \mathbf{R}_E \tag{3.38}$$

where $\mathbf{S}_E$ is the the symmetric matrix from the polar decomposition of $J_E^{-1/3}\mathbf{F}_E$. This results in 6 linear equations for 3 unknowns and is an overdetermined system. Taking 3 of them, a linear system, **Ax**=**B**, is set up and solved to determine $\mathbf{R}_E^T \delta \mathbf{R}_E$:

$$\mathbf{A} = \begin{pmatrix} (S_{11}+S_{22}) & S_{23} & -S_{13} \\ S_{23} & (S_{11}+S_{33}) & S_{12} \\ -S_{13} & S_{12} & (S_{22}+S_{33}) \end{pmatrix} \tag{3.39a}$$

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{3.39b}$$

$$\mathbf{B} = \begin{pmatrix} [\mathbf{R}_E^T \delta \mathbf{F}_E - \delta \mathbf{F}_E^T \mathbf{R}_E]_{12} \\ [\mathbf{R}_E^T \delta \mathbf{F}_E - \delta \mathbf{F}_E^T \mathbf{R}_E]_{13} \\ [\mathbf{R}_E^T \delta \mathbf{F}_E - \delta \mathbf{F}_E^T \mathbf{R}_E]_{23} \end{pmatrix} \tag{3.39c}$$

$\delta \mathbf{R}_E$ can then be found from $\delta \mathbf{R}_E = \mathbf{R}_E(\mathbf{R}_E^T \delta \mathbf{R}_E)$.

In order to solve eq. 3.35, two final equations must be given, namely the solutions to **Z:B** and **B:Z** where $\mathbf{B} = \frac{\partial J_E^{-1/3}\mathbf{F}_E}{\partial \mathbf{F}_E}$ is a 9x9 matrix and **Z** is an arbitrary 3x3 matrix:

$$\mathbf{B} : \mathbf{Z} = J^a(\mathbf{Z} + a(\mathbf{H} : \mathbf{Z})\mathbf{F})) \tag{3.40a}$$

$$\mathbf{Z} : \mathbf{B} = J^a(\mathbf{Z} + a(\mathbf{F} : \mathbf{Z})\mathbf{H})) \tag{3.40b}$$

With the above definitions, it is possible to determine the A_{ij} components of the A matrix for the implicit integration. The B components are given by the right hand-side of eq. 3.33 and the boundary conditions are enforced by altering the values of the B components similar to a Dirichlet condition. Finally, the intermediate velocities can be found by solving the system using a linear solver for a symmetric semi-definite system like the Conjugate Residual or MINRES methods.

3.5 Update Particles

In this final step of the MPM algorithm, the particles are advected by updating their positions based on the velocity field on the grid. In addition, the deformation gradient and temperature must be updated. The steps taken to update these variables from the grid were outlined in section 4.3.2, however, there are some important details to the calculation of the particle deformation gradient and temperature which will be described here.

3.5.1 Updating the Deformation Gradient

The elastic deformation gradient is updated according to the following equation:

$$\mathbf{F}_{Ep}^{n+1} = (\mathbf{I} + \Delta t \nabla \mathbf{v}_p^{n+1})\mathbf{F}_{Ep}^{n} \tag{3.41}$$

where $\nabla \mathbf{v}_p^{n+1}$ is updated from the grid according to eq. 3.19. However, in certain cases, this will lead to $J_{Ep}^{n+1} \leq 0$ due to numerical errors. As the determinant describes the deformed volume, values of zero and below are unphysical, and will also give division by zero in some of the governing equations. To avoid this, an exponential computation of $\mathbf{F}_{Ep}^{n+1}$ should be used instead. This computation is expensive, so Stomakhin et. al[22] implemented a method which updates the deformation gradient using the geometric series of the exponential function; $exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$, including just enough terms to ensure that J_{Ep}^{n+1} is positive.

3.5.2 Temperature Update and Phase Transition

When simulating phase changes, it is important to determine the point when the material changes phase. A simple method would be to set all particles with temperatures above a certain transition temperature to fluid and those below to solid, however, real phase transitions are not immediate. To model a more physically correct phase transition, an energy buffer is used. When a particle is in the solid phase, its buffer will be empty and when it is in the fluid phase, the buffer is full. In the example of melting, the temperature of a solid particle increases until the transition temperature, T_{trans}, is reached. Any temperature increases that take place after this will be removed from the temperature of the particle, and instead converted to heat according to $q = c_p m_p (T_p^{n+1} - T_{trans})$, where c_p is the heat capacity of the particle and q is the heat added by the temperature increase. The heat will be added to the energy buffer and the particle temperature will be clamped to the transition temperature. This process is repeated for each time step until the energy of the buffer reaches a value specificed as the latent heat of the material. At this point, the phase of the particle will be changed to fluid and the temperature will again be allowed to increase. The same process take place in solidification, only the buffer is then originally full and each time step reduces the amount of energy in it until it is empty.

Chapter 4

Implementation of Program

The implementation followed the algorithm described by Stomakhin et. al[22] as outlined in the previous chapter. The program was written in C++11 with an import and export to Houdini providing a simplified interface for the user. OpenGL was also used to visualise the result during implementation. The majority of the code was written specifically for the project except for the MINRES solver which was written by U. Villa[26] and the Alembic Exporter which was originally written for a previous project.[27] A few libraries were also used including OpenMP for simple parallelisation, and Eigen for vector and matrix calculations as well as the Conjugate Gradient and Singular Value Decomposition solvers. The following sections outline the design of the program and the Houdini interface, as well as giving a detailed description of the implementation of the most important simulation steps.

4.1 Design

The implementation of the Material Point Method involved a large number of steps as described in the previous chapter; hence a great deal of thought was necessary when designing the program. The main consideration in the implementation was to produce a correct simulation and create a program which would be fairly easy to understand, thus allowing for future improvements. Optimisation of speed and memory usage was a secondary consideration; although the initial design gave some thought to this, a great deal has been left for future work. Another goal was to make the code extendable so other functions could be added later, for example the collision detection section which was written to allow for level sets to be easily included in the future.

The implementation was originally separated into 11 classes, see fig. 4.1, of which, ReadGeo and AlembicExport were used for the framework around the simulation to import and export simulation data to Houdini, OpenGLWindow was used to visualise the simulation using OpenGL, and MathFunctions was used to contain extendable mathematical calculations or solvers such as the cubic B-Spline calculations and the Conjugate Gradient solver. The remaining classes were directly involved in the actual simulation.

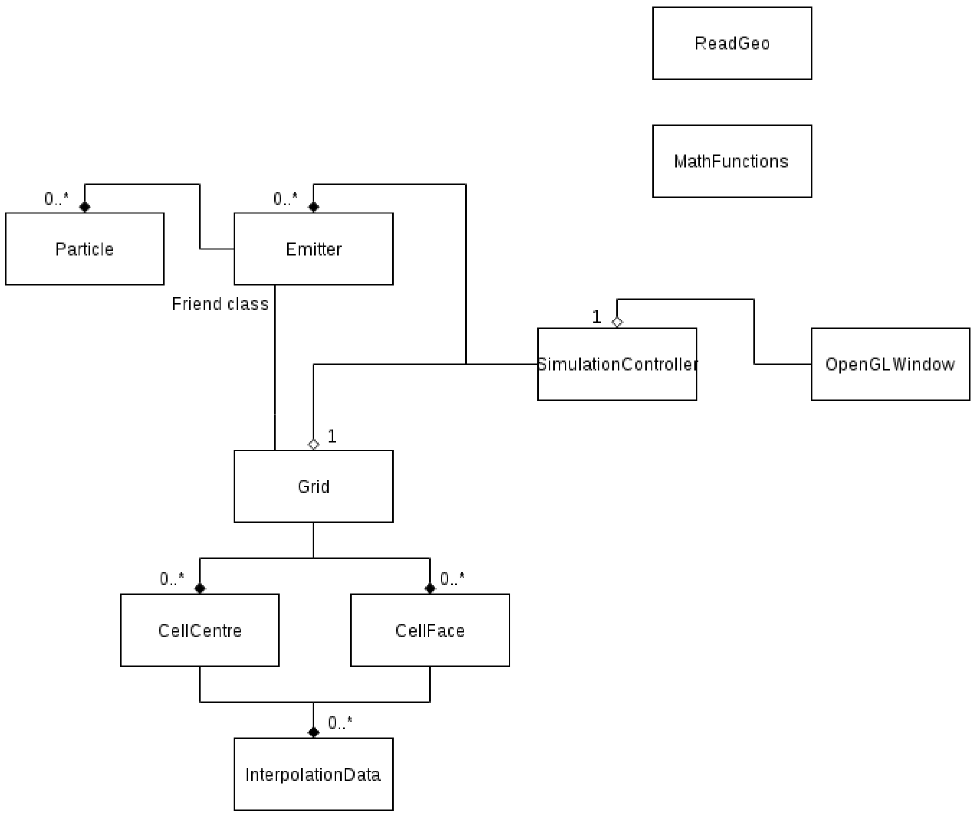


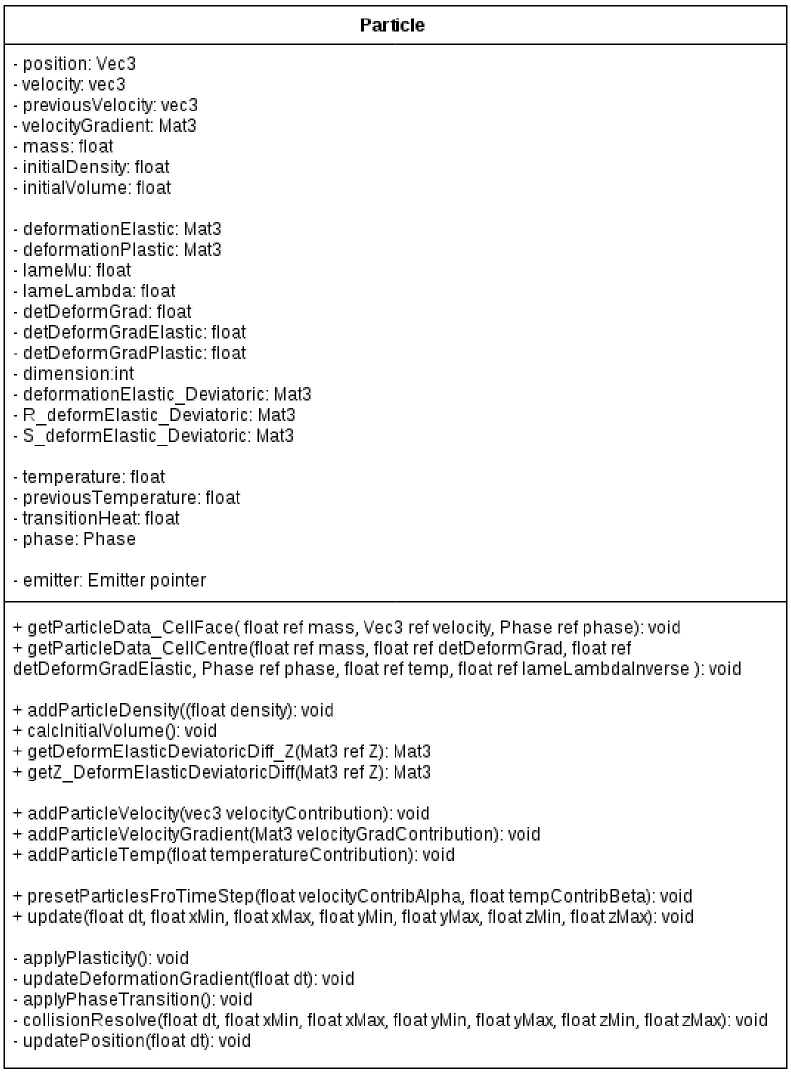
Figure 4.1: Original class diagram

4.1.1 Simulation Controller

This class controls the simulation by communicating with the emitter and grid in turn to preset the particles, perform calculations on the grid and update the particles. It also controls the read in of the simulation parameters from file and allows the results to either be rendered out using the OpenGLWindow or by exporting using Alembic.

4.1.2 Particle

The particle class, fig. 4.2, stores the history dependent simulation data, such as positon, velocity and deformation gradients. This way they represent the deformed material for rendering.

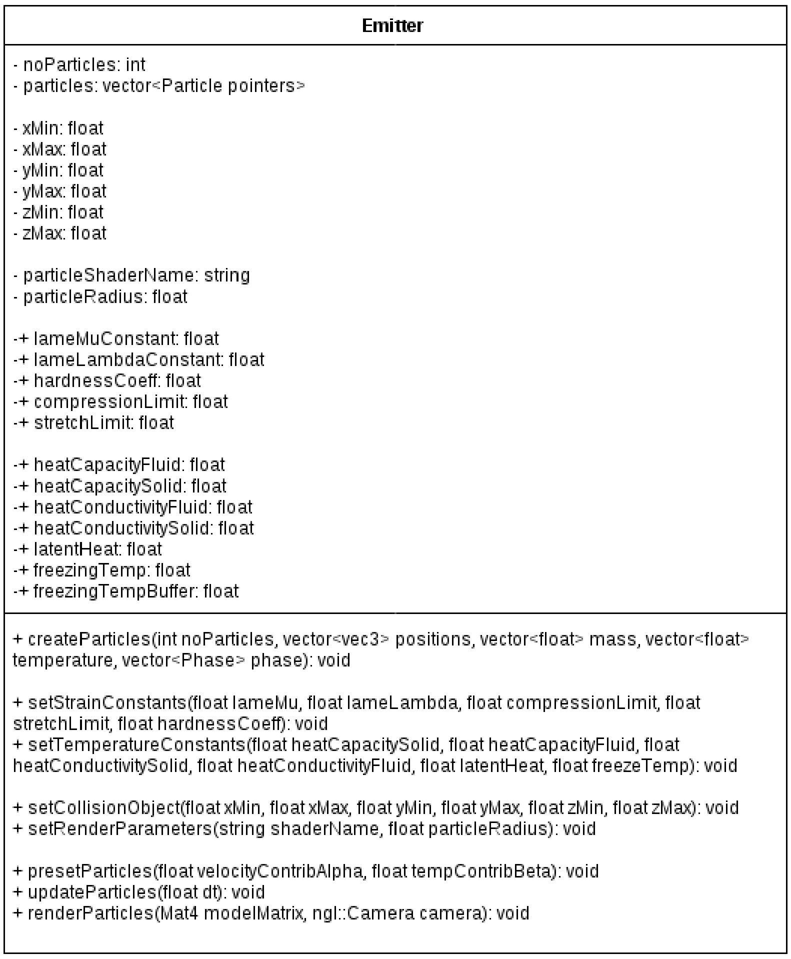


```
Particle

- position: Vec3
- velocity: vec3
- previousVelocity: vec3
- velocityGradient: Mat3
- mass: float
- initialDensity: float
- initialVolume: float

- deformationElastic: Mat3
- deformationPlastic: Mat3
- lameMu: float
- lameLambda: float
- detDeformGrad: float
- detDeformGradElastic: float
- detDeformGradPlastic: float
- dimension:int
- deformationElastic_Deviatoric: Mat3
- R_deformElastic_Deviatoric: Mat3
- S_deformElastic_Deviatoric: Mat3

- temperature: float
- previousTemperature: float
- transitionHeat: float
- phase: Phase

- emitter: Emitter pointer

+ getParticleData_CellFace( float ref mass, Vec3 ref velocity, Phase ref phase): void
+ getParticleData_CellCentre(float ref mass, float ref detDeformGrad, float ref
detDeformGradElastic, Phase ref phase, float ref temp, float ref lameLambdaInverse ): void

+ addParticleDensity((float density): void
+ calcInitialVolume(): void
+ getDeformElasticDeviatoricDiff_Z(Mat3 ref Z): Mat3
+ getZ_DeformElasticDeviatoricDiff(Mat3 ref Z): Mat3

+ addParticleVelocity(vec3 velocityContribution): void
+ addParticleVelocityGradient(Mat3 velocityGradContribution): void
+ addParticleTemp(float temperatureContribution): void

+ presetParticlesFroTimeStep(float velocityContribAlpha, float tempContribBeta): void
+ update(float dt, float xMin, float xMax, float yMin, float yMax, float zMin, float zMax): void

- applyPlasticity(): void
- updateDeformationGradient(float dt): void
- applyPhaseTransition(): void
- collisionResolve(float dt, float xMin, float xMax, float yMin, float yMax, float zMin, float zMax): void
- updatePosition(float dt): void
```

Figure 4.2: Variables and functions in the particle class

4.1.3 Emitter

This class, fig. 4.3, was included to simplify the extension of the program to simulate more than one material in a scene. Hence, it contains the material parameters such as the Lamé constants and the heat constants, as well as a list of all the particles making up the material.

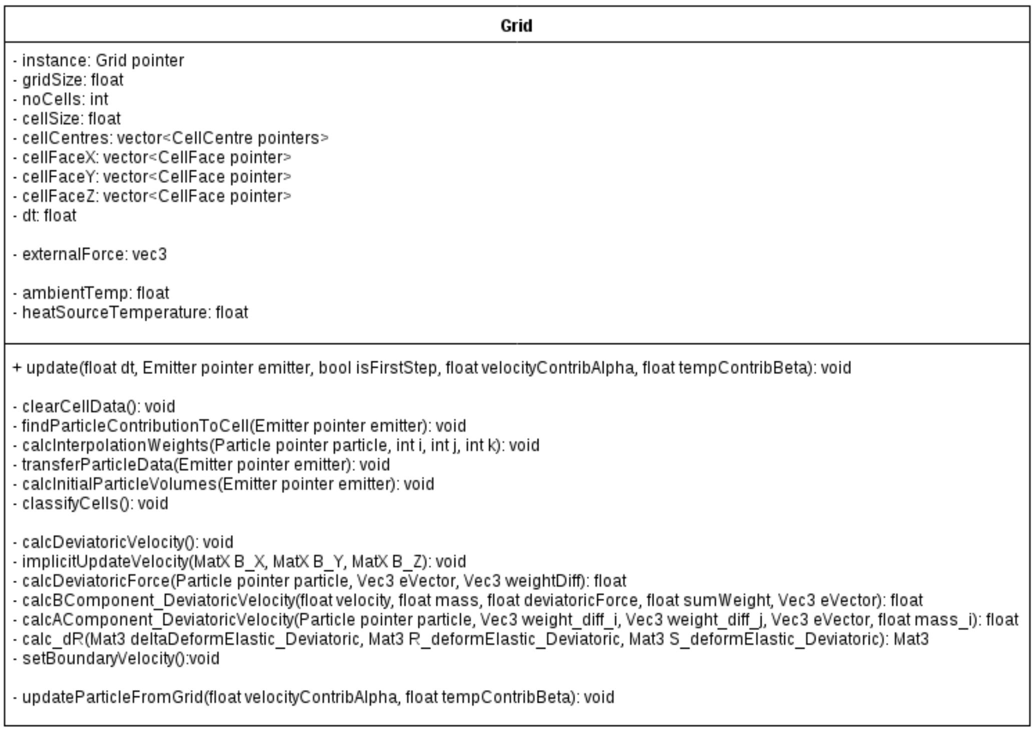


```
Emitter

- noParticles: int
- particles: vector<Particle pointers>

- xMin: float
- xMax: float
- yMin: float
- yMax: float
- zMin: float
- zMax: float

- particleShaderName: string
- particleRadius: float

-+ lameMuConstant: float
-+ lameLambdaConstant: float
-+ hardnessCoeff: float
-+ compressionLimit: float
-+ stretchLimit: float

-+ heatCapacityFluid: float
-+ heatCapacitySolid: float
-+ heatConductivityFluid: float
-+ heatConductivitySolid: float
-+ latentHeat: float
-+ freezingTemp: float
-+ freezingTempBuffer: float

+ createParticles(int noParticles, vector<vec3> positions, vector<float> mass, vector<float>
temperature, vector<Phase> phase): void

+ setStrainConstants(float lameMu, float lameLambda, float compressionLimit, float
stretchLimit, float hardnessCoeff): void
+ setTemperatureConstants(float heatCapacitySolid, float heatCapacityFluid, float
heatConductivitySolid, float heatConductivityFluid, float latentHeat, float freezeTemp): void

+ setCollisionObject(float xMin, float xMax, float yMin, float yMax, float zMin, float zMax): void
+ setRenderParameters(string shaderName, float particleRadius): void

+ presetParticles(float velocityContribAlpha, float tempContribBeta): void
+ updateParticles(float dt): void
+ renderParticles(Mat4 modelMatrix, ngl::Camera camera): void
```

Figure 4.3: Variables and functions in the emitter class

4.1.4 Grid

The grid class, fig. 4.4, makes up the Eulerian grid, in this case a staggered MAC grid. Since this is a background grid used to perform the calculations, it only contains a small number of variables, mainly, lists of cell centres and cell faces. Instead, it has a large number of functions which perform the calculations assigned to the grid.

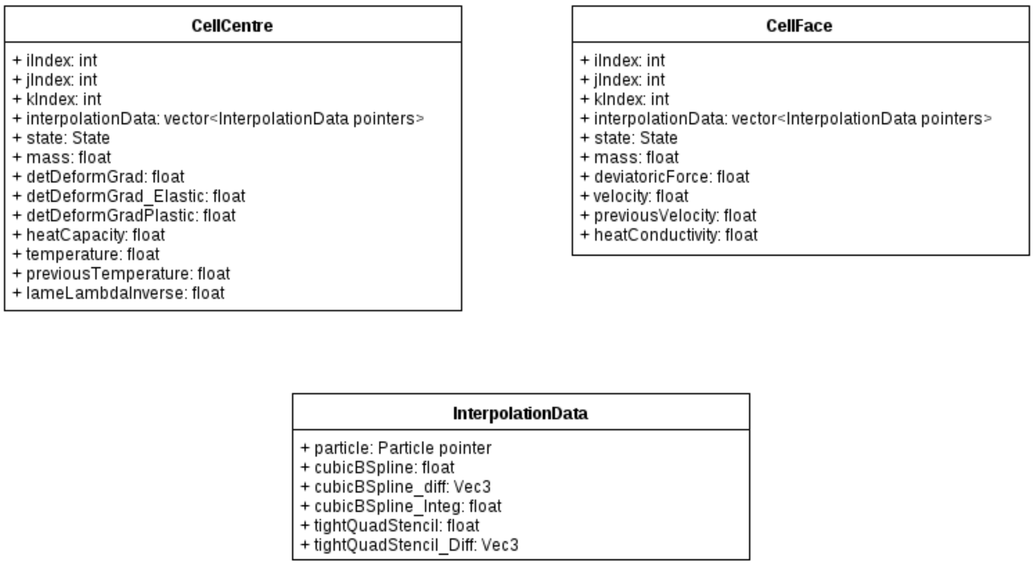


```
Grid

- instance: Grid pointer
- gridSize: float
- noCells: int
- cellSize: float
- cellCentres: vector<CellCentre pointers>
- cellFaceX: vector<CellFace pointer>
- cellFaceY: vector<CellFace pointer>
- cellFaceZ: vector<CellFace pointer>
- dt: float

- externalForce: vec3

- ambientTemp: float
- heatSourceTemperature: float

+ update(float dt, Emitter pointer emitter, bool isFirstStep, float velocityContribAlpha, float tempContribBeta): void

- clearCellData(): void
- findParticleContributionToCell(Emitter pointer emitter): void
- calcInterpolationWeights(Particle pointer particle, int i, int j, int k): void
- transferParticleData(Emitter pointer emitter): void
- calcInitialParticleVolumes(Emitter pointer emitter): void
- classifyCells(): void

- calcDeviatoricVelocity(): void
- implicitUpdateVelocity(MatX B_X, MatX B_Y, MatX B_Z): void
- calcDeviatoricForce(Particle pointer particle, Vec3 eVector, Vec3 weightDiff): float
- calcBComponent_DeviatoricVelocity(float velocity, float mass, float deviatoricForce, float sumWeight, Vec3 eVector): float
- calcAComponent_DeviatoricVelocity(Particle pointer particle, Vec3 weight_diff_i, Vec3 weight_diff_j, Vec3 eVector, float mass_i): float
- calc_dR(Mat3 deltaDeformElastic_Deviatoric, Mat3 R_deformElastic_Deviatoric, Mat3 S_deformElastic_Deviatoric): Mat3
- setBoundaryVelocity():void

- updateParticleFromGrid(float velocityContribAlpha, float tempContribBeta): void
```

Figure 4.4: Variables and functions in the grid class

4.1.5 Cell Centre and Cell Face

The cell centre and cell faces, fig. 4.5, store some data to facilitate the calculations, however, all values are reinitialised at the beginning of a time step such that the grid is only used for calculations. In the implementation, it was chosen to reinitialise the values of the cell variables instead of deleting and regenerating each cell as this would require memory reallocation which might slow down the simulation.



```
CellCentre
+ iIndex: int
+ jIndex: int
+ kIndex: int
+ interpolationData: vector<InterpolationData pointers>
+ state: State
+ mass: float
+ detDeformGrad: float
+ detDeformGrad_Elastic: float
+ detDeformGradPlastic: float
+ heatCapacity: float
+ temperature: float
+ previousTemperature: float
+ lameLambdaInverse: float
```

```
CellFace
+ iIndex: int
+ jIndex: int
+ kIndex: int
+ interpolationData: vector<InterpolationData pointers>
+ state: State
+ mass: float
+ deviatoricForce: float
+ velocity: float
+ previousVelocity: float
+ heatConductivity: float
```

```
InterpolationData
+ particle: Particle pointer
+ cubicBSpline: float
+ cubicBSpline_diff: Vec3
+ cubicBSpline_Integ: float
+ tightQuadStencil: float
+ tightQuadStencil_Diff: Vec3
```

Figure 4.5: Variables stored in the cell classes

4.1.6 Interpolation Data

This class, fig. 4.5, was created in an attempt to increase the efficiency of interpolating data to and from the grid. Through the use of this class, a grid cell knows which particles are affecting it and the respective interpolation weights.

Later in the project it was realised that it would be much faster to calculate the interpolation weights on the go and to do all interpolations of data between the particles and the grid by looping over the particles. This new setup was implemented in the last couple of weeks of the project and lead to the introduction of several grid functions with a name ending in New, however, the overall class diagram, fig. 4.1, did not change.

4.2 Houdini Interface

The Houdini interface was intended to provide a simple user interface for setting the various parameters necessary for the simulation and to set up the particles which make up the material. It should also read in the Alembic file containing the final simulation.

The user interface was set up through the use of a Digital Asset (HDA) which contained a selection of nodes which would add detail and point attributes to a set of particles. The particles were initially set up by using a $GrainSourceSOP$, however, at a later stage in the project it was realised that for a more accurate performance the material should be sampled with at least 8 particles at random positions within each non-empty grid cell. This was done by creating a box representing the bounding box for the simulation which the user can move and scale. The box is then split into cells based on the number of grid cells provided by the user and particles are generated at the centre of each cell using VEX code. It is important to note that the number of cells given in the HDA is the number of cells along one edge and that two cells will be added to this to produce a single layer of cells outside the bounding box for collisions. Hence the total number of cells in the grid will be as follows

$$N_{tot} = (N_{user} + 2)^3 \tag{4.1}$$

where N_{tot} is the total number of cells in the grid and N_{user} is the number of cells given by the user in the HDA. Once the grid particles have been generated and an object to be turned into particles is provided, the HDA performs a level set calculation giving each grid particle a signed value depending on whether it is inside or outside the bounds of the object. For each cell centre particle that is found to be inside the object, a set number of particles are generated at random position within the bounds of that grid cell. The number of particles inside each cell can also be set by the user to give denser objects.

Another important user parameter is the $VoxelSize$ which tells the level set node how finely it should sample the geometry. This parameter and the number of grid cells will have to be altered to give the resolution required by the user; a larger number of grid cells and smaller $VoxelSize$ will give higher resolution.

Once the particles are satisfactory and the simulation parameters have been set, the file must be saved out and the name of the simulation file in the C++ program has to be modified before the simulation can run. As a future improvement a function should be scripted which will automatically save to file and run the program by the push of a button in Houdini, or, ultimately, the entire C++ program should be turned into a plug-in solver.

Once the simulation is running and the export is turned on, the AlembicExport class stores the particle position to file. At this point, it has not been possible to store user-defined parameters such as particle phase or temperature so these can be used in Houdini.

4.3 Implementation of the Main Algorithm Steps

4.3.1 Preset Particles

There are several steps taken to preset the particles at the beginning of a time step.

The velocity is set to previous velocity and the new velocity is set to $\alpha\mathbf{v}_p$, where $\alpha = 0.95$, so that the velocity can be updated from the grid as a mixture of PIC and FLIP contributions, see section 4.3.8. The velocity interpolated to the grid is then technically the previous velocity. The same is done for temperature.

In case the particle is fluid, the deviatoric component of the elastic deformation gradient is removed by setting $\mathbf{F}_{Ep} = J_E^{-1/3}\mathbf{I}$ as outlined in section 3.3.2.

The plasticity contribution is calculated following the steps given in section 3.1.3.

The Lamé coefficients are updated according to eq. 3.11 where the hardening constant ξ is a user-defined parameter. To avoid these values getting too big or too small, they are clamped to threshold values of 0.1 and 10. These thresholds are arbitrary and might need to be changed to improve the simulation. In case the particle is fluid, μ is also set to zero.

The splitting correction is then applied to $\mathbf{F}_E = J_P^{1/3}\mathbf{F}_E$ and $\mathbf{F}_P = J_P^{-1/3}\mathbf{F}_P$ such that $\mathbf{F}_P$ is purely deviatoric.

Finally, the values of $J_E^{-1/d}\mathbf{F}_E$, $\mathbf{R}_E$, $\mathbf{S}_E$, and $\frac{d\hat{\Psi}_\mu}{d\mathbf{F}_E}$ are calculated to be used in the calculation of the deviatoric forces.

4.3.2 Interpolation from Particle to Grid

In the old setup, this step was done in two sections. First $findParticleContributionToCell$ loops over all the particles, determines which cell the particle is in and then loops over the nearest and second nearest neighbour cells. That is, it loops over i,j and k increments of -2 to +2. For each of these cells, the cubic B-Spline is calculated, and if unlike zero, calculates ω_{ip}

for the tight Quadratic Spline and $\nabla\omega_{ip}$ for both interpolation functions. These values, along with a pointer to the particle, is then stored in the cell centre and faces. Once all particles have been checked, the $transferParticleData$ function loops over all cells in the grid and, for each cell, loops through the InterpolationData and adds the contribution of that particle to the cell according to eq. 3.17.

In the new setup, this is performed by one function; $interpolateParticleToGrid$ which loops over the particles 3 times; first, it calculates the mass of the cell centre and faces, second, it transfers the cell density back to the particles, and third, it interpolates the particle data following eq. 3.17. The second step is only done for the first time step of the simulation as the particle density remains constant. In the current setup, the third loop also calculates the deviatoric forces and the components of the **A** and **B** tensors for the intermediate velocity calculation. Although this reduces the total number of loops and the number of times the weights have to be calculated, this setup is fairly convoluted and should possibly be separated out as its own function.

4.3.3 Classification of Cells

In this step, the state of the cell centres and faces is set to be either Colliding, Empty or Interior. It is done by $classifyCells$ or $classifyCells\_New$ and separated into two parts; first determining which cell faces are colliding and then setting the state of the cell centres and remaining cell faces.

In the current implementation, only the cells making up the bounding box are set as colliding by checking the indices of the cell faces, however, in future, this section of the code could easily be exchanged for a function which performs level set collisions.

To determine the state of the cell centres, the function loops over all cells and checks whether the surrounding 6 faces of a cell are colliding. If one of the faces is not colliding, then the cell centre is set to interior if it and all its surrounding faces have mass, and empty if not. The remaining faces are also checked and if they are not colliding, they are set to interior or empty depending on whether they have mass. The function then skips to the next cell in the loop.

As seen in the previous chapter, the velocity, pressure and temperature calculations involve division by either density or mass. It was found that this could lead to inaccuracies when the cells were affected by a small number of particles. Hence, in an effort to prevent this, a noParticlesThreshold variable was introduced such that a cell centre will be set to interior if and only if it and all its surrounding faces are affected by a total number of particles greater than a threshold number.

The temperature of the empty and colliding cells is also set in this step. In the program, there is a heating element at the bottom of the bounding box. Hence, the temperature of these cells is set to the heatSourceTemperature, whilst all other colliding and empty cells are set to the ambientTemperature. As a future improvement, the velocity at the colliding cell faces should probably also be set in this step to reduce the number of loops done in the simulation.

4.3.4 Deviatoric Force Calculation

This step involves calculating the deviatoric force based on eq. 3.26. To prevent the need to calculate $\frac{d\hat{\Psi}_\mu}{dF_E}$ several times for the same particle, the calculation of $\frac{d\hat{\Psi}_\mu}{dF_E}$ is included in the $presetParticleForTimeStep$ function. That way, the grid cell only has to retrieve this value and multiply it with the remaining components, such as $\mathbf{e}_\alpha$ and $\nabla\omega_{i\alpha p}$, to calculate $f_{i\alpha}$ which is done by $calcDeviatoricForce$.

The deviatoric force is then used to calculate the components of the **B** vector for the explicit or implicit integration. This is done by $calcBComponent\_DeviatoricVelocity$ which solves eq. 3.33. In the new setup, the **A** and **B** tensors for the deviatoric velocity calculation are stored on the grid, one for each of the face directions and the B components are calculated directly in the interpolation step.

4.3.5 Explicit and Implicit Time Integration

The time integration is performed by either $explicitUpdateVelocity$ or $implicitUpdateVelocity$ depending on which integration method is chosen. For both methods, a stick collision is enforced by setting the updated velocity to zero at colliding cell faces.

Explicit Integration

The explicit time integration loops over all cells, and for each non-empty cell face, sets the face velocity equal to the **B** component for that cell.

Implicit Integration

The implicit integration has been implemented for both setups but have not been tested enough to be used at the moment. This is because the calculation of the **A** matrices in the old setup was too slow, hence the new system first had to be implemented to achieve a speed-up which would allow testing.

In the old setup, two nested loops over the cells in the grid are used to go over the i and j indices of **A**. The interpolation data of cell i and j are then compared to see if they have any particles in common. This was first done by comparing every particle pointer in the first cell with every particle pointer in the second cell, however, as this would be slow for cells containing a large number of particles, a search function; $searchCellsForCommonParticle$, was implemented to speed this up. Ultimately, this setup was scrapped as the calculation times were too long.

In the new setup, the calculation of the **A** matrix components is included in the loop over particles in the interpolation step. For each particle, the nearest and second nearest neighbours to the cell containing the particle are looped over. For each of these cells, j, $\mathbf{A}_p$ is calculated according to eq. 3.34b and a nested loop over the same cells is set up to give cells i. For each of these cells,i, that are non-empty, the variables belonging to this cell is used to calculate A_{ijp} according to eq. 3.34a. This is then added to **A**(i,j). This way, $\mathbf{A}_p$ only has to be calculated once for each column. The calculations are done for each of the face directions to create **A_X**, **A_Y** and **A_Z**.

As the **A** matrices stored on the grid are set up to contain the total number of cells, there will be rows of all zeros as empty cells are not calculated for. This means that the matrix is singular and the solutions can be inaccurate. To prevent this, the matrices are looped over before they are solved and for each diagonal that is zero, a value of 1 is inserted. However, this might still give inaccurate solutions, and in future, these matrices should be reduced to only contain non-empty cells.

The linear solver used for the implicit integration is a MINRES solver written by U. Villa[26] which can solve symmetric semi-definite systems and has proven to give accurate results against test systems. A slow but general linear solver from the Eigen library is also used to solve the linear system in eq. 3.39 to determine $\delta\mathbf{R}_E$. This linear solver was chosen because the characteristics of the linear system was not known, however, since it is small, a slow but general solver seemed to be a good choice.

4.3.6 Pressure Calculation

The steps of the pressure solver were implemented in $projectVelocity$.

First, the face control volumes of the interior cells is calculated to determine the cell face densities. In case the cells adjacent to the upper faces of the cell are empty or colliding, the densities of these faces are also calculated to be used by the interior cell.

The **A** matrix and **B** vector are then set up. In the current implementation, these are set to contain the total number of cells, but should in future be reduced to only contain the interior cells.

The function then loops over the interior cells to calculate the **B** components, using the cell faces on either side of the cell in question, and the **A** components of the row belonging to that cell. The **A** components are calculated by finding the indices of the cells adjacent to the 6 faces of the cell. Then, depending on the state of these cells, their **A** components are set to $-\frac{\Delta t}{h^2 \rho_{i\alpha}}$ if interior, and zero otherwise. Unless the neighbour cell is colliding, $\frac{\Delta t}{h^2 \rho_{i\alpha}}$ should also be added to the diagonal component for this row. This way the Dirichlet and Neumann boundary conditions are enforced.

As for the implicit integration, ones have to be inserted at the zero diagonals such that the **A** matrices are not singular. The linear system is then solved using the Conjugate Gradient solver from the Eigen library.

Finally, the projected velocities are calculated according to eq. 3.29 where the pressures of interior cells are taken from the solution of the pressure solve, the pressure for empty cells are set to zero and the pressure of colliding cells is set to be the same as the interior cell so the central gradient of pressure is zero at that face.

4.3.7 Temperature Calculation

The temperature calculation is implemented almost the same way as the pressure calculation. The only differences are that the face densities are not required and that the Dirichlet condition is used for colliding cells and the Neumann condition for empty cells. That is, the diagonal **A** component is reduced for each neighbour cell that is empty, not colliding. The system is again solved using the Conjugate Gradient solver, but the solutions can be directly inserted as the whole calculation is done for cell centres only and does not involve cell faces apart from the heat conductivity constant which, in the current implementation, is the same in all face directions.

4.3.8 Interpolate from Grid to Particle

The interpolation from grid to particle follows the PIC/FLIP mix in eq. 3.18-3.19. In the old setup, the cells are looped over, and for each cell the particles contained within it are updated using their corresponding tight Quadratic stencil values. In the new setup on the other hand, the particles are looped over, and for each particle, the nearest and second nearest neighbour cells are looped over. For each cell and particle, the interpolation weights are calculated and the particle values updated. Due to the tighter stencil, it should not be necessary to loop over more than the nearest neighbours as seen in fig. 3.2, however, time did not allow for this to be verified so at the moment it loops over the same number as the cubic B-Spline.

4.3.9 Update Particles

The final step is called by the emitter and lets all particles update independently. In this step, the elastic deformation gradient is updated from the velocity gradient following the geometric series of the exponential as described in section 3.5.1.

Phase transformations are also included by checking whether the current and previous temperature lie either side of the transition temperature. If so, the current temperature is set to the transition temperature. Otherwise, if the previous temperature is equal to the transition temperature, then the difference between the previous and current temperatures is used to calculate the heat which is added to the buffer (or subtracted if temperature is decreasing). Once this is done, the value of the buffer is checked. If equal to the latent heat, the particle is set to fluid and the current temperature is left as is. If the buffer is empty, the particle is set to solid and the current temperature is again left as is. Finally, if the buffer is inbetween, the current temperature is set to the transition temperature.

The new position of the particle is then calculated and checked against the bounding box. If inside, the new position is stored, if outside, the position is clamped to the closest point on the bounding box. The particles are now ready for the next time step.

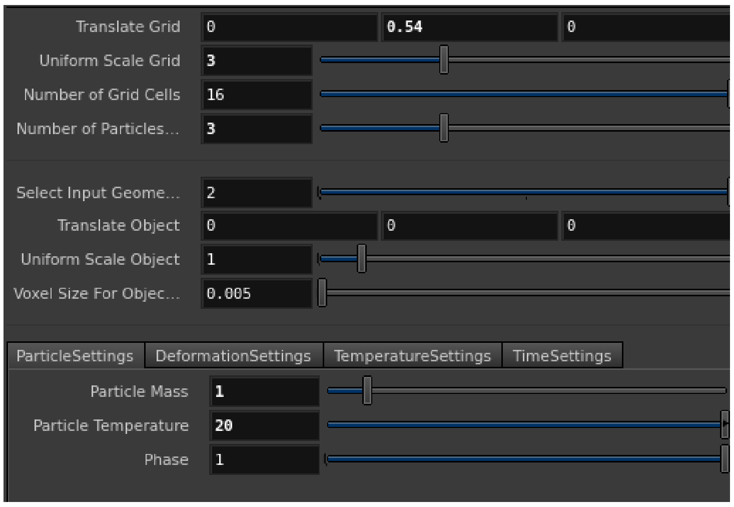
Chapter 5

Results and Discussion

Due to the large scope of the project and the fact that it was difficult to limit the project and still get good visual results, the final results are not as was hoped at the beginning of the project. In hindsight, the initial expectations might have been too great and the project too big for the time given. However, the final results show some of the wanted effects even if stability was not acheived by the deadline.

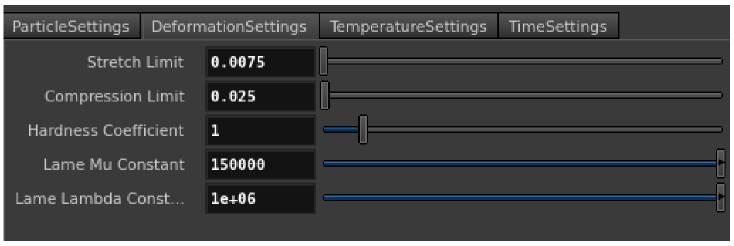
5.1 Final Results

The final render, vid. finalRender, gives the impression of a piece of melting chocolate as the object slowly collapses and spreads to the sides of the bounding box. The user-variables were set as displayed in fig. **??**. The corresponding temperature update, vid. tempUpdate, made it seem that this was taking place due to melting, however, on closer inspection it turned out that a similar effect takes place even when the object is completely solid. Still, there is some difference in the viscosity as seen in vid. solidVsFluid. To prevent the solid from collapsing, the Lame coefficients were increased to $\mu = 1 \cdot 10^6$ and $\lambda = 1 \cdot 10^9$ to make the object stiffer. This altered the behaviour as expected, see vid. stiffSolid, but the simulation became more unstable with certain particles "flying off". With larger increases, the simulation broke down completely. However, the results show that the force calculations are somewhat correct as the overall behaviour is as expected.



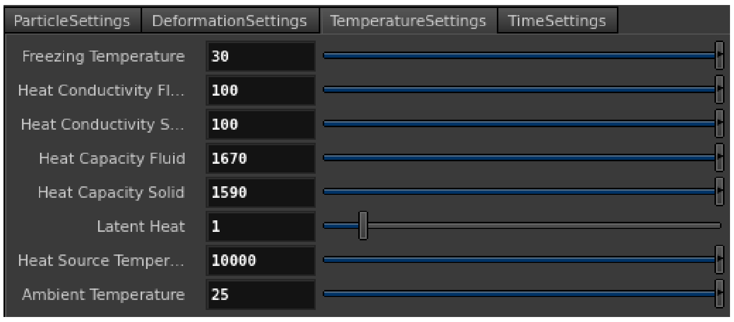
Translate Grid	0	0.54	0
Uniform Scale Grid	3		
Number of Grid Cells	16		
Number of Particles...	3		
Select Input Geome...	2		
Translate Object	0	0	0
Uniform Scale Object	1		
Voxel Size For Objec...	0.005		
Particle Mass	1		
Particle Temperature	20		
Phase	1		

(a) Grid and object.



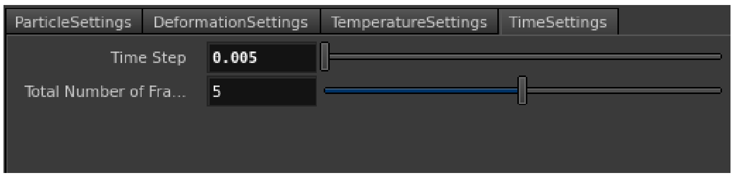
Stretch Limit	0.0075
Compression Limit	0.025
Hardness Coefficient	1
Lame Mu Constant	150000
Lame Lambda Const...	1e+06

(b) Deformation settings



Freezing Temperature	30
Heat Conductivity Fl...	100
Heat Conductivity S...	100
Heat Capacity Fluid	1670
Heat Capacity Solid	1590
Latent Heat	1
Heat Source Temper...	10000
Ambient Temperature	25

(c) Temperature settings



Time Step	0.005
Total Number of Fra...	5

(d) Time settings

Figure 5.1: Simulation paramterers for final render.

Another part that seemed to give correct results was the temperature calculation and phase change. As seen in vid. tempUpdate and phaseUpdate, the heated ground slowly increases the temperature of the object and causes the phase to change.

Some of the main issues that were not resolved within the time span of the project includes the pressure calculation update, the errors that occur in the new setup and the overall stability of the final simulation. These problems will be discussed in the following sections.

5.2 Old and New Interpolation Setup

As mentioned in the previous chapter, a new setup for the interpolation steps was implemented in the last two weeks of the project. It was initially done to allow the implicit integration step to be included without too much increase in computation time. Due to the large number of changes in the overall code, a great deal of time was spent trying to gain the same results as with the old setup. This lead to the discovery of errors in the calculation of the interpolation weights, especially $\nabla\omega_{i\alpha p}$ which did not approach zero for the same values as $\omega_{i\alpha p}$. However, time did not allow for the pressure step to be properly reviewed and hence there are still errors leading to an overall increase in grid velocities with time, and ultimately causing the simulation to break down.

The old setup was the one used to produce the results seen in the accompanying videos. However, a couple of tricks were used to achieve these results. First, the pressure values are clamped to zero to avoid the inclusion of negative pressures. Secondly, a lower threshold value was set for interior cells such that only cells affected by a number of particles greater than the threshold would be set as interior. Thirdly, ones were not included at the zero diagonals of the A matrices for the pressure and temperature calculations. These tricks were removed in the new setup in an effort to produce a more accurate code.

5.3 Instabilities

The main problem encountered in this project was instability. Often a change in the simulation variables would result in the simulation crashing or displaying unexpected results. In fact, the new setup did not even reach a level where a given set of simulation conditions worked. There may have been various reasons for this, ranging from small coding errors to problems with the stability of the implemented equations.

As mentioned, the pressure step seems to be one of the main points of instability. This could be due to time constraints not allowing for this section to be thoroughly debugged. However, some of the errors occurring seemed to be based on the mass, and hence density values, ranging from very large (10^3) to very small (10^{-10}). Initially, this was thought to be due to the particle set up in Houdini, which is why the new HDA was developed. However, when tried with the new HDA, it was found that whilst the HDA can ensure that there is at least 8 particles in each cell, it does not ensure a minimum number of particles affecting each cell. Hence, for the current HDA, a number of 2-3 particles per cell gave better stability than 8. In future, this should probably be improved upon by including a larger number of particles at the surface than in the centre of the object.

Another attempt included altering the pressure and temperature equations, eq. 3.27 and 3.32, to be multiplied by density or mass, as opposed to divided by. This gave a reasonable improvement to the temperature calculation by reducing the number of iterations made by the Conjugate Gradient solver and making the temperature increase more uniform. However, no noticable effect was seen on the pressure calculation. In fact, it might be that the multiplication was not mathematically correct due to the densities being sampled at each face and not having an overall value. Hence, the original equation was used in the new setup. In addition, A. Stomakhin suggested that the central gradients used would not be correct for the MPM setup and that a variational projection[28] should be used instead. Some investigation was done into this but in the end it was decided that there would not be enough time to implement this correctly.

The overall stability might also be improved by reducing the time step or completing the implementation of the semi-implicit time integration. In fact, Klar et. al[21] used time steps as low as 10^{-4} for explicit integration schemes and 10^{-3} for semi-implicit.

Finally, a possible cause of instability could be the interpolation steps as the APIC method was not included. In the article by Klar et. al[21] the difference between APIC and FLIP is shown, as reproduced in fig. 5.2. From this it is clear that the FLIP interpolation is fairly unstable for the MPM technique and it could be that this is causing some of the instabilities in this project as well. Additionally, it was suggested by A. Stomakhin that in the case of the current interpolation setup, the particle positions should be updated directly from the grid velocities. Again, time did unfortunately not allow for this to be attempted.

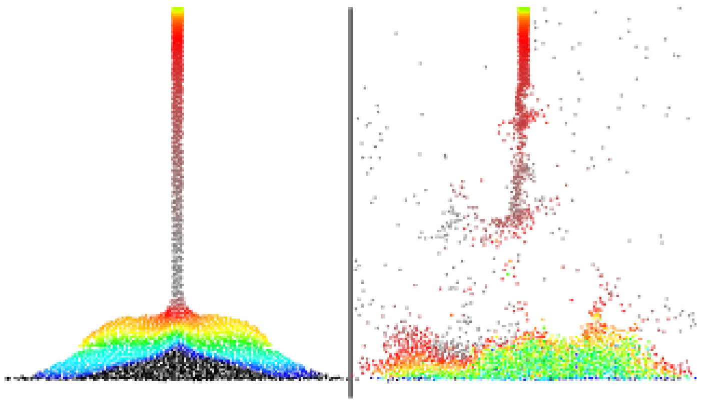


Figure 5.2: Sand simulations with APIC on the left and FLIP on the right. The figure clearly shows that APIC provides much higher stability than FLIP. Reproduced from Klar et. al[21]

5.4 Problems and Questions

Throughout the project, a large number of problems were encountered due to the lack of literature on this subject. The fact that the MPM method was first used in computer graphics in 2013 meant that only a few papers excist. Hence, in cases where the article on melting simulation[22] failed to fully explain an equation or implementation step there were few other resources available. Some of the information detailed here was in fact not presented until SIGGRAPH 2016, implying that it was not available until half way through the project. This especially concerns the importance of the APIC method, the requirements for the interpolation weights and the issues in the pressure calculation. A large amount of help was obtained from the MPM course at SIGGRAPH[23] and an email from A. Stomakhin. Had the help from A. Stomakhin been available at the beginning of the project, it might be that the final result would have been better. However, the project did provide an insight into a new and exciting field within physically-based simulation and could possibly produce a stable and accurate simulation with time.

Chapter 6

Conclusion and Further Work

The Material Point Method has been introduced to the field of computer graphics over the recent years and could become a very important method for simulating deformable bodies. Due to the physical accuracy of the model, it is extensive and not the simplest to implement, hence the project did not succeed in aquiring a stable melting simulation. However, certain steps were completed and showed promise, such as the temperature update and the deformation taking place as the object meets the ground. In future, I would probably start the project from scratch as the project has allowed me to deepen my understanding of the field and the method, hence some of the code now seems convoluted and possibly inaccurate. Still, with further debugging of the current implementation, a modification of the particle setup in Houdini and by using APIC for the data transfer, I believe the melting simulation would be achievable.

Bibliography

[1] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer. Elastically Deformable Models. *SIGGRAPH Comput. Graph.*, 21(4):205–214, 1987.

[2] D. Sulsky, S. Zhou, and H. L. Schreyer. Particle Simulation Methods Application of a Particle-In-Cell Method to Solid Mechanics. *Computer Physics Communications*, 87(1):236 – 252, 1995.

[3] D. Terzopoulos, J. Platt, and K. Fleischer. Heating and Melting Deformable Models. *The Journal of Visualization and Computer Animation*, 2(2):68–73, 1991.

[4] T. G. Goktekin, A. W. Bargteil, and J. F. O'Brien. A Method for Animating Viscoelastic Fluids. *ACM Transactions on Graphics (Proc. of ACM SIGGRAPH 2004)*, 23(3):463–468, 2004.

[5] M. Carlson, P. J. Mucha, R. Brooks Van Horn, III, and G. Turk. Melting and Flowing. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '02, pages 167–174. ACM, 2002.

[6] Y. Zhao, L. Wang, F. Qiu, A. Kaufman, and K. Mueller. Melting and Flowing in Multiphase Environment. *Computers and Graphics*, 30:2006, 2006.

[7] J. F. O'Brien and J. K. Hodgins. Graphical Modeling and Animation of Brittle Fracture. In *Proceedings of ACM SIGGRAPH 1999*, pages 137–146. ACM Press/Addison-Wesley Publishing Co., 1999.

[8] M. Teschner, B. Heidelberger, M. Muller, and M. Gross. A versatile and robust model for geometrically complex deformable solids. In *Computer Graphics International, 2004. Proceedings*, pages 312–319, 2004.

[9] A. W. Bargteil, C. Wojtan, J. K. Hodgins, and G. Turk.

[10] C. Wojtan, N. Thürey, M. Gross, and G. Turk. Deforming Meshes That Split and Merge. *ACM Trans. Graph.*, 28(3):76:1–76:10, 2009.

[11] M. Wicke, D. Ritchie, B. M. Klingner, S. Burke, J. R. Shewchuk, and J. F. O'Brien. Dynamic Local Remeshing for Elastoplastic Simulation. *ACM Transactions on Graphics*, 29(4):49:1–11, 2010.

[12] P. Clausen, M. Wicke, J. R. Shewchuk, and J. F. O'Brien. Simulating Liquids and Solid-Liquid Interactions with Lagrangian Meshes. *ACM Transactions on Graphics*, 32(2):17:1–15, 2013.

[13] M. Desbrun and M. Gascuel. Smoothed Particles: A New Paradigm for Animating Highly Deformable Bodies. In *Proceedings of the Eurographics Workshop on Computer Animation and Simulation '96*, pages 61–76. Springer-Verlag New York, Inc., 1996.

[14] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff. Position Based Dynamics. *J. Vis. Comun. Image Represent.*, 18(2):109–118, 2007.

[15] B. Solenthaler, J. Schläfli, and R. Pajarola. A Unified Particle Model for Fluid–Solid Interactions. *Computer Animation and Virtual Worlds*, 18(1):69–82, 2007.

[16] A. Paiva, F. Petronetto, T. Lewiner, and G. Tavares. Particle-based Non-Newtonian Fluid Animation for Melting Objects. In *2006 19th Brazilian Symposium on Computer Graphics and Image Processing*, pages 78–85, 2006.

[17] M. Müller, R. Keiser, A. Nealen, M. Pauly, M. Gross, and M. Alexa. Point Based Animation of Elastic, Plastic and Melting Objects. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '04, pages 141–151. Eurographics Association, 2004.

[18] F. Losasso, G. Irving, E. Guendelman, and R. Fedkiw. Melting and Burning Solids into Liquids and Gases. *IEEE Transactions on Visualization and Computer Graphics*, 12(3):343–352, 2006.

[19] A. Stomakhin, C. Schroeder, L. Chai, J. Teran, and A. Selle. A Material Point Method for Snow Simulation. *ACM Trans. Graph.*, 32(4):102:1–102:10, 2013.

[20] C. Jiang, C. Schroeder, A. Selle, J. Teran, and A. Stomakhin. The Affine Particle-in-cell Method. *ACM Trans. Graph.*, 34(4):51:1–51:10, 2015.

[21] G. Klár, T. Gast, A. Pradhana, C. Fu, C. Schroeder, C. Jiang, and J. Teran. Drucker-prager Elastoplasticity for Sand Animation. *ACM Trans. Graph.*, 35(4):103:1–103:12, 2016.

[22] A. Stomakhin, C. Schroeder, C. Jiang, L. Chai, J. Teran, and A. Selle. Augmented MPM for Phase-Change and Varied Materials. *ACM Trans. Graph.*, 33(4), 2014.

[23] C. Jiang, C. Schroeder, J. Teran, A. Stomakhin, and A. Selle. The Material Point Method for Simulating Continuum Materials. In *ACM SIGGRAPH 2016 Courses*, SIGGRAPH '16, 2016.

[24] R. Bridson and M. Müller-Fischer. Fluid Simulation: SIGGRAPH 2007 Course notesVideo Files Associated with This Course Are Available from the Citation Page. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, pages 1–81. ACM, 2007.

[25] J. Stam. Stable Fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 121–128. ACM Press/Addison-Wesley Publishing Co., 1999.

[26] Umberto Villa. TMINRES. `https://code.google.com/archive/p/tminres/`, 2012.

[27] I. M. Sorensen. Explosion. `https://github.com/Inamsorensen/Explosion`, 2016.

[28] C. Batty, F. Bertails, and R. Bridson. A Fast Variational Framework for Accurate Solid-fluid Coupling. *ACM Trans. Graph.*, 26(3), 2007.