UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Multi-material simulation with the Material Point Method

Masterarbeit

zur Erlangung des Grades Master of Science (M.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Alexander Maximilian Nilles

Erstgutachter: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)

Zweitgutachter: Bastian Krayer, M.Sc.
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im Mai 2020

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☒	☐

Polch, 11.05.2020

....

(Ort, Datum) (Unterschrift)

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de

UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Masterarbeit
Alexander Maximilian Nilles
(Mat. Nr. 214 200 655)

Thema: Multi-material simulation with the Material Point Method

The Material Point Method can be used to numerically simulate continuum materials such as gases, liquids and solids. It combines a lagrangian, particle-based perspective with an eulerian, grid-based one, while transferring quantities between both views, combining advantages of these approaches. It has recently gained popularity due to its usage in movies such as *Frozen* for simulating snow and other materials.

The goal of this thesis is to implement a simulation with multiple materials based on the Material Point Method and evaluate it or compare it to other methods or variations of the same method. The simulation should utilize the GPU where it is feasible and appropriate.

This thesis focuses on:

1. Learning the basics
   - Continuum mechanics
   - Material Point Method
2. Choice of a material model and conceptual design
3. Implementation
4. Evaluation
5. Documentation of the results

Koblenz, den 12.11.2019

- Alexander Maximilian Nilles -

- Prof. Dr. Stefan Müller-

Zusammenfassung

Die *Material Point Method* (MPM) hat sich in der Computergrafik als äußerst fähige Simulationsmethode erwiesen, die in der Lage ist ansonsten schwierig zu animierende Materialien zu modellieren [1, 2]. Abgesehen von der Simulation einzelner Materialien stellt die Simulation mehrerer Materialien und ihrer Interaktion weitere Herausforderungen bereit. Dies ist Thema dieser Arbeit. Es wird gezeigt, dass die MPM durch die Fähigkeit Eigenkollisionen implizit handzuhaben ebenfalls in der Lage ist Kollisionen zwischen Objekten verschiedenster Materialien zu beschreiben, selbst, wenn verschiedene Materialmodelle eingesetzt werden. Dies wird dann um die Interaktion poröser Materialien wie in [3] erweitert, was ebenfalls gut mit der MPM integriert. Außerdem wird gezeigt das MPM auf Basis eines einzelnen Gitters als Untermenge dieses Mehrgitterverfahrens betrachtet werden kann, sodass man das gleiche Verhalten auch mit mehreren Gittern modellieren kann. Die poröse Interaktion wird auf beliebige Materialien erweitert, einschließlich eines frei formulierbaren Materialinteraktionsterms. Das Resultat ist ein flexibles, benutzersteuerbares Framework das unabhängig vom Materialmodell ist. Zusätzlich wird eine einfache GPU-Implementation der MPM vorgestellt, die die Rasterisierungspipeline benutzt um Schreibkonflikte aufzulösen. Anders als andere Implementationen wie [4] ist die vorgestellte Implementation kompatibel mit einer Breite an Hardware.

Abstract

The *Material Point Method* (MPM) has proven to be a very capable simulation method in computer graphics that is able to model materials that were previously very challenging to animate [1, 2]. Apart from simulating singular materials, the simulation of multiple materials that interact with each other introduces new challenges. This is the focus of this thesis. It will be shown that the self-collision capabilities of the MPM can naturally handle multiple materials interacting in the same scene on a collision basis, even if the materials use distinct constitutive models. This is then extended by porous interaction of materials as in [3], which also integrates easily with MPM. It will furthermore be shown that regular single-grid MPM can be viewed as a subset of this multi-grid approach, meaning that its behavior can also be achieved if multiple grids are used. The porous interaction is generalized to arbitrary materials and freely changeable material interaction terms, yielding a flexible, user-controllable framework that is independent of specific constitutive models. The framework is implemented on the GPU in a straightforward and simple way and takes advantage of the rasterization pipeline to resolve write-conflicts, resulting in a portable implementation with wide hardware support, unlike other approaches such as [4].

Contents

I Introduction **1**

1 Motivation **1**

2 Related Work **1**

II Basics **4**

3 Continuum Mechanics **4**

4 Material Point Method **11**
4.1 Procedure 11
4.2 Particle-Grid Transfers 15
4.2.1 Grid Basis Functions 15
4.2.2 Particle-In-Cell Method 18
4.2.3 Fluid Implicit Particle Method 18
4.2.4 Affine Particle-In-Cell Method (APIC) 18

5 Constitutive Models **19**
5.1 Hyperelasticity 20
5.1.1 Saint Venant-Kirchhoff 20
5.1.2 Hencky Strain 20
5.1.3 Neo-Hookean 20
5.1.4 Fixed Corotated 21
5.1.5 Fluids 21
5.2 Plasticity 22
5.2.1 Snow 22
5.2.2 Sand 23

III Method and Implementation **27**

6 Method **27**
6.1 Multiple Materials 27
6.1.1 Single Constitutive Model 27
6.1.2 Multiple Constitutive Models 28
6.1.3 Porous Materials 29
6.2 MPM Details 32
6.2.1 Particle-Grid Transfer 32
6.2.2 Moving Least Squares Material Point Method 32
6.2.3 Kernels 34

7 Implementation **34**
7.1 Overview . . . 34
7.2 Libraries and Technologies . . . 36
7.3 Buffers and Textures . . . 37
7.3.1 Particle State . . . 37
7.3.2 Materials . . . 39
7.3.3 Grid . . . 41
7.4 Shader . . . 43
7.4.1 P2G Shader . . . 44
7.4.2 Grid Shader . . . 47
7.4.3 G2P Shader . . . 49
7.5 Particle Initialization . . . 49
7.5.1 Sampling Strategies . . . 51
7.5.2 Object Types . . . 52
7.6 Visualization . . . 54
7.6.1 Grid Visualization . . . 54
7.6.2 Particle Visualization . . . 54
7.7 User Interface . . . 57

IV Evaluation **60**

8 Sampling Methods **60**

9 Basis Functions **61**

10 Single Constitutive Model **63**

11 Multiple Constitutive Models **66**

12 Porous Materials **69**

13 Particle Collisions **73**

V Conclusion and Future Work **75**

Part I
Introduction

1 Motivation

Capturing and reproducing the behavior of real-world materials is a challenging task in computer graphics. Many materials are quite complex and especially phenomena such as fracturing or granular flow can be hard to reproduce in simulations. While graphics does not necessarily need physically plausible behavior of materials, failure to do so can mean that the behavior also looks implausible to a human viewer.

While materials such as water have already been researched extensively in the past, with fluid solvers being commonplace in VFX [3], many other types of materials lacked good solutions. In recent years, the *Material Point Method* (MPM) has gained a lot of momentum in the graphics community, starting with its usage for snow simulation in Disney's *Frozen* [1]. Their model can simulate many different types of snow, which has previously been very difficult to model [1]. MPM further proved very effective in the simulation of granular materials, i.e. for sand animation [2, 5], as well as viscoelastic fluids, foams and sponges [6, 7]. There even exist more advanced formulations of MPM that can be used for cloth simulations [8].

A lot of the research typically focuses on specific materials that are then simulated on their own with one-way coupling to rigid body collision objects. The MPM is however also very capable in multi-material simulations that are fully coupled, which is the topic of this thesis. Some of the related research is introduced in the following section. Section 3 provides an introduction into the field of continuum mechanics, followed by a description of the MPM in Section 4. A selection of constitutive models that were implemented in this thesis is detailed in Section 5. The multi-material framework is then introduced in Section 6, starting with material variety based on single constitutive models, then expanding this to multiple models and finally extending it with porous material interaction based on [3]. The rasterization based implementation is then described in Section 7, followed by the evaluation (Part IV) and conclusion (Part V).

2 Related Work

A lot of research has been published on the Material Point Method (MPM) in the graphics community in recent years. It has also benefitted greatly from advances to *Particle-in-Cell* (PIC) fluid simulations, which is outlined later in Section 4.2. Recent research has addressed many of the shortcomings and limitations of the method, enabling greater material variety as well as

coupling with different simulation frameworks.

Stomakhin *et al.* [9] developed an augmented MPM that can handle phase-change, i.e. melting and solidifying. The approach is capable of simulating a wide range of materials in the same scene. A heat-equation solver that was adapted to MPM is used to handle temperature changes, with each material point additionally carrying temperature and phase state. The former is used to vary the Lamé parameters of the material, while phase only determines whether deviatoric stresses exist (solid phase) or not (fluid phase). This required a splitting of the constitutive model into deviatoric and dilational parts, which they achieve with a modification of the fixed corotated energy density function from [10]. Their method also treats *latent heat*, which is heat that is needed for a phase change that does not trigger a change in temperature. They achieve this with temperature buffers on material points. In order to support a larger variety of materials, they furthermore extend the method to be able to handle incompressible and nearly incompressible materials without locking. This allows incompressible fluids and highly rigid solids to be simulated with MPM. Like typical incompressible fluid solvers, they use staggered *marker and cell* grids for this and adapt MPM accordingly. The dilational stress is used to derive pressure needed for their generalized Chorin-style projection approach that facilitates incompressibility. Deviatoric stress is handled by MPM as usual.

Jiang *et al.* [11] describe a method that couples mesh-based Lagrangian simulations with MPM. In order to do this, the forces $\boldsymbol{f}_p$ on meshed particles are first calculated as usual with the given Lagrangian force model, for example mass-spring systems or finite elements. These forces are then transformed into forces $\boldsymbol{f_i}$ on the Eulerian grid used in MPM, which allows to consider them during the velocity update on the grid. Masses and velocities of the meshed particles are also transferred to the grid, which is done exactly as for MPM particles. This enables the MPM particles to be affected by the meshed particles. In order to achieve two-way coupling, the velocity of meshed particles is also subjected to the MPM transfers, meaning that velocity is transferred back from the grid onto them as well. A side-effect of this treatment is that the Lagrangian simulation can now benefit from the automatic collision handling provided by Eulerian grids, while still keeping its precise surface tracking capability. With this two-way coupling, materials such as cloth that are a bad fit for standard MPM and are instead simulated with mesh-based approaches can be introduced into MPM simulations and interact with materials that instead are harder to simulate mesh-based.

Hu *et al.* [12] developed the *Moving Least Squares Material Point Method*, which is a new discretization scheme that is described in detail in Section 6.2.2. This served as a basis for their *Compatible Particle-In-Cell* (CPIC) algorithm. CPIC adds the possibility of material point discontinuities which is not possible in regular MPM. Discontinuities make infinitely thin boundaries possible by preventing material points on either side to affect each other.

The authors achieve this with the usage of colored distance fields and a compatibility condition that specifies whether a given material point and grid node are allowed to affect each other. With CPIC, thin-shell dynamic rigid bodies achieve the desired effects of material separation and cutting of objects is made possible. Cutting previously required either very thick objects, or techniques such as deletion of material points or plastic softening that introduce artifacts. The authors furthermore developed two-way rigid body coupling, allowing proper interaction between rigid bodies and MPM. Among other things, they used this to successfully predict movement of a robot through sand.

Part II
Basics

3 Continuum Mechanics

This section will briefly introduce the basics of continuum mechanics. If not otherwise stated, the information in this section was taken from continuum-mechanics.org. For an in-depth overview on the topic, similar information can be found in literature such as [13, 14].

As the name implies, objects are considered to be a continuous lump of mass in this field of physics. The fact that they are actually made up of a discrete set of particles such as atoms is thus abstracted away, meaning that behavior of materials is described using models. A core task in continuum mechanics is to describe the deformations or *strains* in a continuum and derive *stresses* from them.

Stress is force divided by area. It describes the internal burden of an object based on outside forces. Force perpendicular to the surface results in *normal stress* (σ), while force parallel to the surface yields *shear stress* (τ). Figure 1 illustrates the concept on an imaginary cut through the object along the y-axis. The stresses

$$\sigma_{xx} = \frac{F_x}{A_x} \quad \text{and} \quad \tau_{xy} = \frac{F_y}{A_x} \tag{1}$$

can be defined here, where A_x is the area of the cut surface. The normal stresses in this example are positive because the object is under tension. In the case of compression, the force would point inwards, resulting in negative normal stresses.

The stresses that can be defined like this can be summarized in a *stress tensor* $\boldsymbol{\sigma}$. In 3D, this is

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_{yy} & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \sigma_{zz} \end{bmatrix} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix}. \tag{2}$$

Stress tensors are symmetric, i.e. $\tau_{xy} = \tau_{yx}$.

The eigenvalues of a stress tensor are the *principal stresses* σ_1, σ_2 and σ_3. They correspond to the normal stresses if the coordinate system is oriented according to principal stress space, i.e. towards the eigenvectors of $\boldsymbol{\sigma}$. In this case, there are no shear stresses, meaning that the stress tensor only contains the eigenvalues on its diagonal and is zero anywhere else. The eigenvalues also contain the maximum and minimum possible normal stresses. Even though the stress tensor changes depending on the coordinate system used

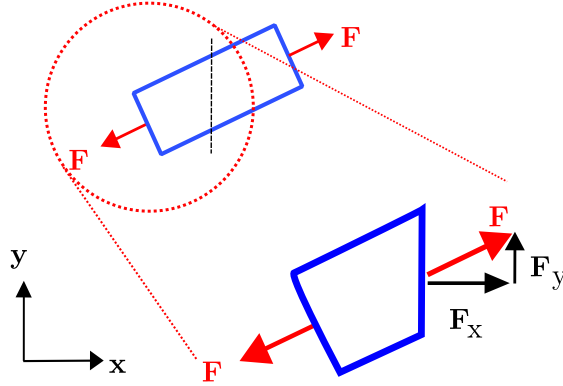


Figure 1: An imaginary cut through an object resulting in a surface. The force F is split into a normal and tangential component, resulting in normal and shear stresses. Source: continuummechanics.org [1]

to describe it, there are many invariants that do not change if the coordinate system is changed:

$$I_1(\boldsymbol{\sigma}) = \mathrm{tr}(\boldsymbol{\sigma}) \qquad = \sigma_1 + \sigma_2 + \sigma_3 \tag{3}$$

$$I_2(\boldsymbol{\sigma}) = \frac{1}{2}\left(\mathrm{tr}(\boldsymbol{\sigma}^2) - \mathrm{tr}\,(\boldsymbol{\sigma})^2\right) \qquad = -(\sigma_1\sigma_2 + \sigma_2\sigma_3 + \sigma_3\sigma_1) \tag{4}$$

$$I_3(\boldsymbol{\sigma}) = \det(\boldsymbol{\sigma}) \qquad = \sigma_1 \cdot \sigma_2 \cdot \sigma_3 \tag{5}$$

This also means that the *hydrostatic stress*

$$\sigma_{\mathrm{Hyd}} = \frac{\mathrm{tr}(\boldsymbol{\sigma})}{3} \tag{6}$$

is invariant as well.

Strain describes an objects deformation. There are *normal strains* (ϵ) and *shear strains* (γ). Given a displacement field $\boldsymbol{u}(\boldsymbol{X})$ on the undeformed positions $\boldsymbol{X}$, the normal stresses are defined as

$$\epsilon_{xx} = \frac{\partial u_x}{\partial X_x}, \quad \epsilon_{yy} = \frac{\partial u_y}{\partial X_y} \quad \text{and} \quad \epsilon_{zz} = \frac{\partial u_z}{\partial X_z}. \tag{7}$$

An object stretched by 2% along the x-axis would mean $\epsilon_{xx} = 0.02$, while compression would yield a negative number. The shear strains can be defined as

$$\gamma_{yx} = \frac{\partial u_x}{\partial X_y} + \frac{\partial u_y}{\partial X_x} = \gamma_{xy} \tag{8}$$

and analogously for the other axis combinations. Figure 2 illustrates what these definitions mean using finite differences instead of derivatives.

[1] `https://www.continuummechanics.org/stress.html` Last accessed on 08.04.2020

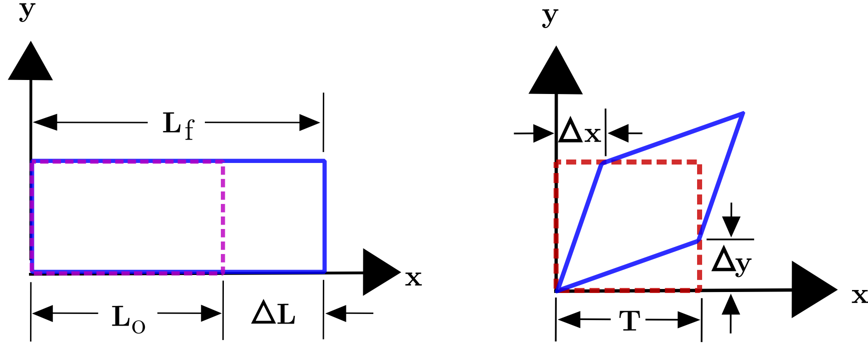


Figure 2: Normal strains (left) and shear strains (right) in terms of finite differences. The dashed red outline represents the reference configuration and the blue outline the deformed configuration. Here, $\epsilon = \frac{\Delta L}{L_0}$ and $\gamma = \frac{\Delta x + \Delta y}{T}$. These definitions are also called *engineering strain*. Source: continuum-mechanics.org [2]

The strains can be summarized in a *small strain tensor* $\boldsymbol{\epsilon}$ which is defined as

$$\boldsymbol{\epsilon} = \begin{bmatrix} \epsilon_{xx} & \gamma_{xy}/2 & \gamma_{xz}/2 \\ \gamma_{yx}/2 & \epsilon_{yy} & \gamma_{yz}/2 \\ \gamma_{zx}/2 & \gamma_{zy}/2 & \epsilon_{zz} \end{bmatrix} = \begin{bmatrix} \epsilon_{xx} & \epsilon_{xy} & \epsilon_{xz} \\ \epsilon_{yx} & \epsilon_{yy} & \epsilon_{yz} \\ \epsilon_{zx} & \epsilon_{zy} & \epsilon_{zz} \end{bmatrix}. \tag{9}$$

According to Eq. (8), this tensor is also symmetric. The problem of this strain tensor is that rigid body rotations will result in strains, even though they should not, meaning that this tensor is only applicable for small rotations. This problem will be addressed later.

The eigenvalues of a strain tensor are the *principal strains* ϵ_1, ϵ_2 and ϵ_3. They contain the maximum and minimum possible normal strains and correspond to the normal strains if the coordinate system is oriented such that there are no shear strains. This orientation is given by the eigenvectors. Strain tensors have the same invariants as stress tensors:

$$I_1(\boldsymbol{\epsilon}) = \operatorname{tr}(\boldsymbol{\epsilon}) \qquad = \epsilon_1 + \epsilon_2 + \epsilon_3 \tag{10}$$

$$I_2(\boldsymbol{\epsilon}) = \frac{1}{2}\left(\operatorname{tr}(\boldsymbol{\epsilon}^2) - \operatorname{tr}(\boldsymbol{\epsilon})^2\right) \qquad = -(\epsilon_1\epsilon_2 + \epsilon_2\epsilon_3 + \epsilon_3\epsilon_1) \tag{11}$$

$$I_3(\boldsymbol{\epsilon}) = \det(\boldsymbol{\epsilon}) \qquad = \epsilon_1 \cdot \epsilon_2 \cdot \epsilon_3 \tag{12}$$

The first invariant corresponds to the *volumetric strain* which describes the volume change that has occurred as

$$\epsilon_{\text{Vol}} = \frac{\Delta V}{V} \approx \operatorname{tr}(\boldsymbol{\epsilon}). \tag{13}$$

This is only approximate, but the error is small as long as the strains are small.

[2] `https://www.continuummechanics.org/strain.html` Last accessed on 08.04.2020

Hooke's Law describes how strain and stress are related for isotropic hyperelastic materials. It is a first order linearization, but it can still be used for nonlinear material laws as an approximation as long as the strains are small.

The model uses two parameters. The first is Young's modulus (E), also called tensile modulus. It describes the stiffness of a material. Higher values mean that higher stress is needed to achieve the same strain. E has to be positive. The second parameter is Poisson's ratio ν. It describes how much the material expands or shrinks in the direction perpendicular to compression or stretching. Positive values mean that the material reacts in opposition, i.e. that it expands in the direction perpendicular to compression and shrinks in the direction perpendicular to stretching. $\nu = 0$ means that neither lateral expansion nor lateral contraction happens, while $\nu = 0.5$ describes incompressible materials. Negative values are possible and result in a material exhibiting lateral expansion if it is stretched and vice-versa.

Hooke's Law is defined as

$$\boldsymbol{\epsilon} = \frac{1}{E}\big((1+\nu)\boldsymbol{\sigma} - \nu \boldsymbol{I}\,\mathrm{tr}(\boldsymbol{\sigma})\big) \tag{14}$$

and it can be inverted to relate strain to stress instead

$$\boldsymbol{\sigma} = \frac{E}{(1+\nu)}\left(\boldsymbol{\epsilon} + \frac{\nu}{(1-2\nu)}\boldsymbol{I}\,\mathrm{tr}(\boldsymbol{\epsilon})\right). \tag{15}$$

Based on Hooke's Law, other material properties can be defined and related to E and ν. Rearranging Eq. (15) yields

$$\begin{aligned}\boldsymbol{\sigma} &= 2 \cdot \frac{E}{2(1+\nu)}\boldsymbol{\epsilon} + \frac{E\nu}{(1+\nu)(1-2\nu)}\boldsymbol{I}\,\mathrm{tr}(\boldsymbol{\epsilon}) \\ &= 2\mu\boldsymbol{\epsilon} + \lambda\boldsymbol{I}\,\mathrm{tr}(\boldsymbol{\epsilon}).\end{aligned} \tag{16}$$

μ and λ are called the first and second Lamé parameters, respectively. They are used in place of E and ν in many constitutive models as they greatly simplify the formulae. μ is always positive and is also called *shear modulus* (G). In fluid dynamics it is instead named *dynamic viscosity.* λ is usually positive but is allowed to be negative.

Fluids are often described using the bulk modulus

$$K = -V\frac{dp}{dV}, \tag{17}$$

where p is pressure and V is volume. Pressure is the negative of hydrostatic stress, which means that the bulk modulus can be defined in terms of hydrostatic stress and volumetric strain. This allows to define K in terms of E and ν using Hooke's Law, resulting in

$$K = \frac{\sigma_{Hyd}}{\epsilon_{Vol}} = \frac{E}{3(1-2\nu)} \tag{18}$$

Deformation gradients are used to solve the problems of rigid body translations and rotations by separating them from the actual deformations. Instead of directly computing the small strain tensor, a strain tensor is instead computed from a deformation gradient $\boldsymbol{F}$. Given the reference vector $\boldsymbol{X}$ and the deformed vector $\boldsymbol{x} = \boldsymbol{X} + \boldsymbol{u}$, it is calculated as

$$\boldsymbol{F} = \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{X}} = \begin{bmatrix} \frac{\partial x_1}{\partial X_1} & \frac{\partial x_1}{\partial X_2} & \frac{\partial x_1}{\partial X_3} \\ \frac{\partial x_2}{\partial X_1} & \frac{\partial x_2}{\partial X_2} & \frac{\partial x_2}{\partial X_3} \\ \frac{\partial x_3}{\partial X_1} & \frac{\partial x_3}{\partial X_2} & \frac{\partial x_3}{\partial X_3} \end{bmatrix} = \boldsymbol{I} + \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{X}}. \tag{19}$$

The deformation gradient is a linear transformation. This means that $J = \det(\boldsymbol{F})$ describes the volume change, leading to the following relation

$$1 + \epsilon_{\text{Vol}} = \frac{V + \Delta V}{V} = \frac{JV}{V} = J. \tag{20}$$

This measure is exact, unlike Eq. (13).

The small scale strain from Eq. (9) can be calculated from $\boldsymbol{F}$ as

$$\boldsymbol{\epsilon} = \frac{1}{2}\left(\boldsymbol{F} + \boldsymbol{F}^T\right) - \boldsymbol{I}. \tag{21}$$

This suffers from the previously mentioned problem of rigid body rotations, which is why strain is defined differently in practice. One possible definition uses the polar decomposition $\boldsymbol{F} = \boldsymbol{R} \cdot \boldsymbol{U}$. This separates the rotation $\boldsymbol{R}$, leaving a symmetric matrix $\boldsymbol{U}$. A strain tensor can be defined from this as

$$\boldsymbol{\epsilon}' = \boldsymbol{U} - \boldsymbol{I} \tag{22}$$

which is equivalent to the small scale strain in the absence of rotation.

$\boldsymbol{U}$ can be computed using the singular value decomposition $\boldsymbol{F} = \boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{W}^T$ and setting

$$\boldsymbol{R} = \boldsymbol{V}\boldsymbol{W}^T, \quad \boldsymbol{U} = \boldsymbol{W}\boldsymbol{\Sigma}\boldsymbol{W}^T. \tag{23}$$

This is an expensive operation. An alternative uses the fact that

$$\boldsymbol{F}^T \cdot \boldsymbol{F} = (\boldsymbol{R} \cdot \boldsymbol{U})^T \cdot (\boldsymbol{R} \cdot \boldsymbol{U}) = \boldsymbol{U}^T \cdot \boldsymbol{R}^T \cdot \boldsymbol{R} \cdot \boldsymbol{U} = \boldsymbol{U}^T \cdot \boldsymbol{U}, \tag{24}$$

which also eliminates rotation. This is called the *right Cauchy-Green deformation tensor*. The *left Cauchy-Green deformation tensor* is instead defined as $\boldsymbol{F} \cdot \boldsymbol{F}^T$, which also eliminates rotation. Based on this, strains can be defined. A common choice is the *Green-Lagrangian strain tensor*

$$\boldsymbol{E} = \frac{1}{2}(\boldsymbol{F}^T\boldsymbol{F} - \boldsymbol{I}). \tag{25}$$

This strain tensor is not equivalent to the small strain tensor, but it is a reasonable approximation for small strains.

Another strain tensor is the *true strain* which was introduced by Hencky [15]. It is also called *logarithmic strain* or *Hencky strain*, calculated as $\frac{1}{2}\ln(\boldsymbol{F}\boldsymbol{F}^T)$ or $\frac{1}{2}\ln(\boldsymbol{F}^T\boldsymbol{F})$ [16]. The true strain is often the preferred measure of strain because it is a more physical measure of strain and has better properties in large deformation scenarios compared to the Green-Lagrangian strain [16].

In conjunction with Hooke's Law, different strains give rise to different stresses. The Green-Lagrangian strain results in the *second Piola–Kirchhoff stress* $\boldsymbol{\sigma}^{\mathrm{PK2}}$, while true strain gives rise to *Cauchy stress* $\boldsymbol{\sigma}$ [16]. They are related as

$$\boldsymbol{\sigma}^{\mathrm{PK2}} = J\boldsymbol{F}^{-1}\boldsymbol{\sigma}\boldsymbol{F}^{-T}. \tag{26}$$

A key difference between them is that $\boldsymbol{\sigma}^{\mathrm{PK2}}$ is force over area in the reference configuration $\boldsymbol{X}$, while $\boldsymbol{\sigma}$ is given in the deformed configuration $\boldsymbol{x}$.

Another definition of stress is the *first Piola-Kirchhoff stress*, which is related to the Cauchy stress as

$$\boldsymbol{\sigma}^{\mathrm{PK1}} = J\boldsymbol{\sigma}\boldsymbol{F}^{-T}. \tag{27}$$

In contrast to the other stresses, $\boldsymbol{\sigma}^{\mathrm{PK1}}$ is not symmetric.

So far, only hyperelastic materials are accounted for. For elasto-plastic materials, $\boldsymbol{F}$ is separated into an elastic part $\boldsymbol{F}_E$ and a plastic part $\boldsymbol{F}_P$ such that

$$\boldsymbol{F} = \boldsymbol{F}_E\boldsymbol{F}_P, \tag{28}$$

which is called *multiplicative plasticity theory* [1]. Exactly how $\boldsymbol{F}$ is split depends on the material's *yield criterion*. Only $\boldsymbol{F}_E$ is used to calculate stresses and the corresponding elastic response of the material, while $\boldsymbol{F}_P$ is „forgotten“, but may still have an impact on the material's behavior, i.e. by changing its stiffness [1].

Material derivatives describe the rate at which a property of a piece of material that moves with a velocity $\boldsymbol{v}$ changes over time. Given a function $\phi(\boldsymbol{x})$, its material derivative is

$$\frac{d\phi}{dt} = \frac{\partial\phi}{\partial t} + \boldsymbol{v}\cdot\nabla\phi. \tag{29}$$

The material derivative of position is simply the velocity, while the material derivative of velocity is acceleration. For the deformation gradient, it is

$$\frac{d\boldsymbol{F}}{dt} = \frac{d}{dt}\left(\frac{\partial\boldsymbol{x}}{\partial\boldsymbol{X}}\right) = \frac{\partial}{\partial\boldsymbol{X}}\left(\frac{d\boldsymbol{x}}{dt}\right) = \frac{\partial\boldsymbol{v}}{\partial\boldsymbol{X}}. \tag{30}$$

This is the partial derivative of velocity with respect to the reference configuration, which can be computed using the chain rule:

$$\frac{\partial\boldsymbol{v}}{\partial\boldsymbol{X}} = \frac{\partial\boldsymbol{v}}{\partial\boldsymbol{x}}\cdot\frac{\partial\boldsymbol{x}}{\partial\boldsymbol{X}} = (\nabla\boldsymbol{v})\boldsymbol{F} = \frac{d\boldsymbol{F}}{dt}. \tag{31}$$

This means that deformation gradients can be evolved using only the current deformation gradient and the velocity gradient.

Conservation laws are a core concept in continuum mechanics. Conservation of mass is necessary for an Eulerian analysis that defines density ρ and velocity $\boldsymbol{v}$ at fixed points in space that do not move with the material flow. Lagrangian analysis greatly simplifies conservation of mass, as it uses particles that are advected with the flow and have their mass associated with them. Conservation of mass ensures that the rate at which mass is flowing in and out of a volume matches the rate at which mass changes within the volume. It is expressed using the *continuity equation*

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{v}) = 0. \tag{32}$$

By applying the product rule to the divergence term, the continuity equation can be reformulated:

$$\begin{aligned} & \frac{\partial \rho}{\partial t} + \boldsymbol{v} \cdot \nabla \rho + \rho(\nabla \cdot \boldsymbol{v}) = 0 \\ \iff & \frac{d\rho}{dt} + \rho(\nabla \cdot \boldsymbol{v}) = 0. \end{aligned} \tag{33}$$

This can be further simplified for incompressible materials, as they require that the material derivative of density is 0:

$$\begin{aligned} & \frac{d\rho}{dt} + \rho(\nabla \cdot \boldsymbol{v}) = 0 \\ \iff & \rho(\nabla \cdot \boldsymbol{v}) = 0 \\ \iff & \nabla \cdot \boldsymbol{v} = 0. \end{aligned} \tag{34}$$

The Lagrangian form is simply

$$\rho_0 = \rho J, \tag{35}$$

meaning that the initial density in reference configuration has to equal the current density in the deformed configuration multiplied by the determinant of the deformation gradient.

Conservation of momentum ensures that the sum of all forces acting on an element is equal to its mass multiplied by its acceleration. This is expressed using the *equilibrium equation*:

$$\rho \frac{d\boldsymbol{v}}{dt} = \nabla \cdot \boldsymbol{\sigma} + \rho \boldsymbol{f}. \tag{36}$$

Here, $\boldsymbol{f}$ is the acceleration due to external forces such as gravity, while $\nabla \cdot \boldsymbol{\sigma}$ is the acceleration due to internal stresses (multiplied by density). The

equilibrium equation is very similar to the Navier-Stokes equation for fluid flow,

$$\rho \frac{d\boldsymbol{v}}{dt} = -\nabla p + \mu \nabla^2 \boldsymbol{v} + \rho \boldsymbol{f}. \tag{37}$$

The only difference is that $\nabla \cdot \boldsymbol{\sigma}$ is instead replaced with its definition in terms of pressure p and viscosity μ. The continuity equation is also used in the process, meaning that the Navier-Stokes equation can be considered a combination of both.

4 Material Point Method

The Material Point Method is a continuum based hybrid method. It combines Lagrangian particles (the *material points*) with an Eulerian Cartesian background grid. Unlike some Lagrangian methods, the material points do not need to be connected by a mesh, just like in *Smoothed Particle Hydrodynamics* (SPH). MPM was proposed by Sulsky *et al.* [17] as an extension of the Particle-In-Cell Method to solid mechanics, meaning that it shares the same hybrid approach. Thus, improvements of PIC such as the *Fluid Implicit Particle Method* (FLIP) can be applied to MPM as well. The Lagrangian nature of MPM simplifies conservation of mass as well as the discretization of the equilibrium equation's left-hand side (Eq. (36)), while the Eulerian grid ensures proper self-collision as well as fracturing in an implicit way. Because there is no mesh connecting the material points, the discretization of stress derivatives is complicated. These are necessary in order to compute forces. The Eulerian grid addresses this problem. It is used along with basis functions to discretize $\nabla \cdot \boldsymbol{\sigma}$, similar to the *finite element method* (FEM) [1].

According to Stomakhin *et al.* [1], MPM was not used in graphics before their publication. They used MPM to simulate snow, which was utilized in the movie *Frozen* by Disney. Since then, MPM has been intensively used for graphics applications such as sand animation [2], multi-species simulations [3] or phase-change and nearly incompressible solids and fluids [9]. The method was also topic of SIGGRAPH courses [18, 19] and it was implemented using the GPU [4]. This thesis will focus on MPM from a graphics viewpoint instead of an engineering one.

4.1 Procedure

In order to differ between quantities associated with particles and those associated with grid nodes, subscripts will be used in the following. For example, $m_{\boldsymbol{i}}^n$ would be the mass associated with node $\boldsymbol{i}$ at timestep n and m_p the mass associated with particle p. The grid nodes are equally spaced and typically store all quantities at cell centers, although other grids such as *marker and cell* (MAC) grids that store information at centers and faces are possible too [9].

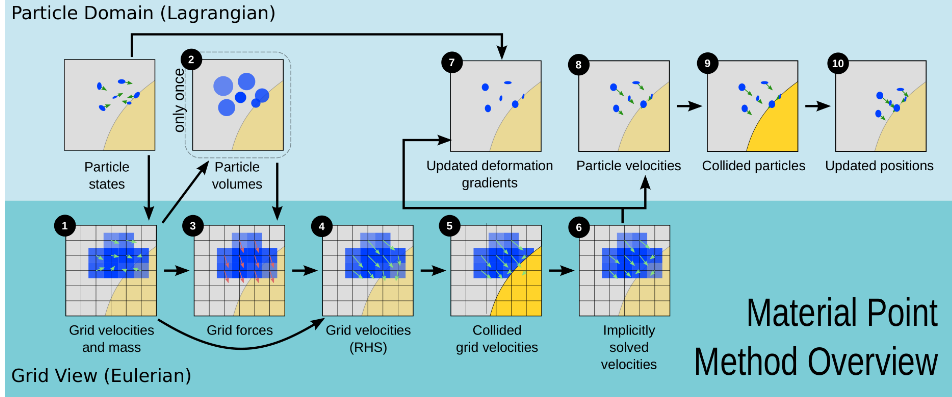


Figure 3: Overview of the 10 steps of the MPM according to Stomakhin *et al.* [1]. Image source: [1].

Many variations of MPM exist. Figure 3 shows the basic procedure according to Stomakhin *et al.* [1], which is divided into 10 steps:

1. **Rasterize particle data to the grid.** This step transfers mass and velocity from particles to grid nodes. Mass is transferred as

$$m_{\boldsymbol{i}}^n = \sum_p m_p \omega_{\boldsymbol{i}p}^n. \tag{38}$$

   Here, $\omega_{\boldsymbol{i}p}^n$ is the weight between grid node and particle, dictated by the chosen grid basis function (see Section 4.2.1). There are numerous options for velocity transfer, three of which are detailed in Sections 4.2.2 to 4.2.4.

2. **Compute particle volumes and densities.** This is only necessary for particle initialization and is thus only done at the first timestep. Grid cell density is estimated as $\rho_{\boldsymbol{i}}^0 = m_{\boldsymbol{i}}^0/h^3$, where h is the grid spacing. Particle density can be estimated from this as $\rho_p^0 = \sum_{\boldsymbol{i}} \rho_{\boldsymbol{i}}^0 \omega_{\boldsymbol{i}p}^0$, and volume is $V_p^0 = m_p/\rho_p^0$.

3. **Compute grid forces.** This step requires computing the Cauchy stress $\boldsymbol{\sigma}_p$, which in turn depends upon the deformation gradient $\boldsymbol{F}_p^n$ and the chosen constitutive model. Deformation gradients can be initialized as $\boldsymbol{F}_p^0 = \boldsymbol{I}$ and are evolved in a later step. Constitutive models are often expressed using an elasto-plastic energy density function $\Psi(\boldsymbol{F}_{Ep}, \boldsymbol{F}_{Pp})$. The derivative of this with respect to the deformation gradient is the first Piola-Kirchhoff stress,

$$\boldsymbol{\sigma}_p^{PK1} = \frac{\partial \Psi}{\partial \boldsymbol{F}_p}. \tag{39}$$

This means that

$$\boldsymbol{\sigma}_p = \frac{1}{\det(\boldsymbol{F}_{Ep}^n)} \frac{\partial \Psi}{\partial \boldsymbol{F}_E} (\boldsymbol{F}_{Ep}^n)^T = \frac{1}{J_{Ep}} \frac{\partial \Psi}{\partial \boldsymbol{F}_E} (\boldsymbol{F}_{Ep}^n)^T \tag{40}$$

The forces can now be calculated as

$$\boldsymbol{f}_{\boldsymbol{i}} = -\sum_p V_p^n \boldsymbol{\sigma}_p \nabla \omega_{\boldsymbol{i}p}^n. \tag{41}$$

Because $V_p^n = J_{Ep} V_p^0$, this can also be written as

$$\boldsymbol{f}_{\boldsymbol{i}}^n = -\sum_p V_p^0 \boldsymbol{\sigma}_p^{PK1} (\boldsymbol{F}_{Ep}^n)^T \nabla \omega_{\boldsymbol{i}p}^n = -\sum_p V_p^0 \frac{\partial \Psi}{\partial \boldsymbol{F}_E} (\boldsymbol{F}_{Ep}^n)^T \nabla \omega_{\boldsymbol{i}p}^n. \tag{42}$$

Other external forces such as gravity can then be added to $\boldsymbol{f}_{\boldsymbol{i}}$.

4. **Update velocities on grid** as

$$\boldsymbol{v}_{\boldsymbol{i}}^\star = \boldsymbol{v}_{\boldsymbol{i}}^n + \frac{\Delta t}{m_{\boldsymbol{i}}^n} \boldsymbol{f}_{\boldsymbol{i}}^n \tag{43}$$

5. **Grid-based body collisions.** Velocities $\boldsymbol{v}_{\boldsymbol{i}}^\star$ are updated based on collisions. This update is one-way in basic MPM, but two-way coupling is possible to implement [12].

   Stomakhin *et al.* [1] implement collision based on *level sets* ϕ. Level sets are negative inside objects, positive outside objects and 0 on the surface of an object. This means that a position is colliding if $\phi(\boldsymbol{x}) \leq 0$. In this case, the surface normal $\boldsymbol{n} = \frac{\nabla \phi(\boldsymbol{x})}{\|\nabla \phi(\boldsymbol{x})\|}$ and collision object velocity $\boldsymbol{v}_{\text{co}}$ are computed. Based on the relative velocity $\boldsymbol{v}_{\text{rel}} = \boldsymbol{v} - \boldsymbol{v}_{\text{co}}$ it is determined if the objects are already moving away from each other ($v_n = \boldsymbol{v}_{\text{rel}} \cdot \boldsymbol{n} \geq 0$), in which case nothing has to be done. Otherwise, the relative tangential velocity $\boldsymbol{v}_t = \boldsymbol{v}_{\text{rel}} - \boldsymbol{n} v_n$ is computed. Given the coefficient of friction μ_{fr}, if $\|\boldsymbol{v}_t\| \leq -\mu_{\text{fr}} v_n$ or if the surface is flagged as *sticky*, the relative velocity is updated as $\boldsymbol{v}'_{\text{rel}} = \boldsymbol{0}$. In the other cases, dynamic friction is applied as $\boldsymbol{v}'_{\text{rel}} = \boldsymbol{v}_t + \mu_{\text{fr}} v_n \frac{\boldsymbol{v}_t}{\|\boldsymbol{v}_t\|}$. Lastly, the velocity is updated as $\boldsymbol{v}' = \boldsymbol{v}'_{\text{rel}} + \boldsymbol{v}_{\text{co}}$.

   This procedure is used on grid node velocities $\boldsymbol{v}_{\boldsymbol{i}}^\star$ given their position $\boldsymbol{x}_{\boldsymbol{i}}$. In Step 9, the same algorithm is used to collide and update particle velocities $\boldsymbol{v}_p^{n+1}$ based on positions $\boldsymbol{x}_p$.

   Level sets are not limited to rigid bodies. Stomakhin *et al.* [1] use them for stationary and dynamic rigid bodies as well as deforming objects. They implemented the latter by defining level sets for key frames and interpolating between them.

6. **Solve the linear system.** Stomakhin *et al.* [1] proposed semi-implicit integration. This involves solving a system of linear equations to get $\hat{\boldsymbol{v}}_{\boldsymbol{i}}^{n+1}$ from $\boldsymbol{v}_{\boldsymbol{i}}^{\star}$, which requires the Hessian $\frac{\partial^2 \Psi}{\partial \boldsymbol{F}_E \partial \boldsymbol{F}_E}$, a fourth-rank tensor. The system can be solved using iterative methods such as GMRES [3]. This approach allows for timesteps that are one to three magnitudes smaller compared to explicit integration [1, 3]. However, Tampubolon *et al.* [3] note that it introduces severe artificial cohesion in their use-case.

   Semi-implicit or implicit integration approaches are not explored further in the following because they would introduce a lot of complexity and potential problems for the multi-material approach introduced in Part III. In an explicit scheme the velocity is updated as

   $$\hat{\boldsymbol{v}}_{\boldsymbol{i}}^{n+1} = \boldsymbol{v}_{\boldsymbol{i}}^{\star}. \tag{44}$$

   Note that $\hat{\boldsymbol{v}}_{\boldsymbol{i}}^{n+1}$ differs from $\boldsymbol{v}_{\boldsymbol{i}}^{n+1}$ which is the velocity transferred from particles in the next timestep.

7. **Update deformation gradient.** This step updates the elastic and plastic portions of the deformation gradient. All additional deformations that happened in this timestep are first attributed to $\boldsymbol{F}_E$ using Eq. (31):

   $$\hat{\boldsymbol{F}}_{Ep}^{n+1} = (\boldsymbol{I} + \Delta t \nabla \boldsymbol{v}_p^{n+1}) \boldsymbol{F}_{Ep}^{n} \tag{45}$$

   The plastic portion is set as $\hat{\boldsymbol{F}}_{Pp}^{n+1} = \boldsymbol{F}_{Pp}^{n}$. Together, they combine to the full deformation gradient as

   $$\boldsymbol{F}_p^{n+1} = \hat{\boldsymbol{F}}_{Ep}^{n+1} \hat{\boldsymbol{F}}_{Pp}^{n+1}. \tag{46}$$

   A grid transfer is used to evaluate the velocity gradient as

   $$\nabla \boldsymbol{v}_p^{n+1} = \sum_{\boldsymbol{i}} \hat{\boldsymbol{v}}_{\boldsymbol{i}}^{n+1} (\nabla \omega_{\boldsymbol{i}p}^{n})^T. \tag{47}$$

   If the constitutive model is purely elastic, then there is no plastic deformation, i.e. the plastic portion of the deformation gradient is always the identity. This means that the deformation gradient update is already done and $\hat{\boldsymbol{F}}_{Ep}^{n+1} = \boldsymbol{F}_{Ep}^{n+1} = \boldsymbol{F}_p^{n+1}$. Otherwise, $\hat{\boldsymbol{F}}_{Ep}^{n+1}$ is subjected to the yield criterion of the constitutive model and split into an elastic and plastic portion:

   $$\hat{\boldsymbol{F}}_{Ep}^{n+1} = \boldsymbol{F}_{Ep}^{n+1} \boldsymbol{F}_{Pp}^{\star}. \tag{48}$$

   The plastic portion that was split off is then attributed to the total plastic deformation gradient

   $$\boldsymbol{F}_{Pp}^{n+1} = \boldsymbol{F}_{Pp}^{\star} \hat{\boldsymbol{F}}_{Pp}^{n+1}. \tag{49}$$

8. **Update particle velocities.** Grid velocities are transferred back onto the particles as $\boldsymbol{v}_p^{n+1}$. This depends on the chosen grid transfer scheme, just like the opposite direction in Step 1 does. The different options are described in Sections 4.2.2 to 4.2.4.

9. **Particle-based body collisions.** This step optionally applies collision on $\boldsymbol{v}_p^{n+1}$. The same procedure from Step 5 is used. If this step is skipped, particles can slightly penetrate into collision bodies. The problem is reduced at higher grid resolutions. However, Klár *et al.* [2] note that applying collisions on particles directly can cause their positions to become out of sync with deformation gradients, which is a reason to skip this step. The artifacts caused by this can range from barely noticeable to major differences in simulations (see Section 13).

10. **Update particle positions** as

$$\boldsymbol{x}_p^{n+1} = \boldsymbol{x}_p^n + \Delta t \boldsymbol{v}_p^{n+1}. \tag{50}$$

4.2 Particle-Grid Transfers

Transferring quantities from particles onto grid nodes and back is a core task in MPM. It is used to estimate grid mass (Eq. (38)) and grid velocities, during the stress-based force evaluation (Eq. (41)), to update deformation gradients (Eqs. (45) and (47)) and to transfer velocities back onto particles.

This section will describe the grid basis functions used in the process and what choices exist, as well as different approaches as to how velocities are transferred.

4.2.1 Grid Basis Functions

An arbitrary 1D basis function $N(x)$ centered on $\mathbf{0}$ can be used to form 3D grid basis functions using the tensor product as

$$N_{\boldsymbol{i}}^h(\boldsymbol{x}) = N(\frac{1}{h}(x - x_{\boldsymbol{i}}))N(\frac{1}{h}(y - y_{\boldsymbol{i}}))N(\frac{1}{h}(z - z_{\boldsymbol{i}})) \quad [18] \tag{51}$$

where $\boldsymbol{x} = (x, y, z)^T$, h is the grid spacing and $\boldsymbol{x_i} = (x_{\boldsymbol{i}}, y_{\boldsymbol{i}}, z_{\boldsymbol{i}})^T$ is the position of node $\boldsymbol{i}$. Based on this, the weights between nodes and particles are defined as

$$\omega_{\boldsymbol{i}p}^n = N_{\boldsymbol{i}}^h(\boldsymbol{x}_p^n) \quad [1] \tag{52}$$

for particle positions $\boldsymbol{x}_p^n = (x_p^n, y_p^n, z_p^n)^T$ at timestep n. The gradient is defined as

$$\nabla\omega_{\boldsymbol{i}p}^n = \begin{pmatrix} \frac{1}{h}\nabla N(\frac{1}{h}(x_p^n - x_{\boldsymbol{i}})) \cdot N(\frac{1}{h}(y_p^n - y_{\boldsymbol{i}})) \cdot N(\frac{1}{h}(z_p^n - z_{\boldsymbol{i}})) \\ N(\frac{1}{h}(x_p^n - x_{\boldsymbol{i}})) \cdot \frac{1}{h}\nabla N(\frac{1}{h}(y_p^n - y_{\boldsymbol{i}})) \cdot N(\frac{1}{h}(z_p^n - z_{\boldsymbol{i}})) \\ N(\frac{1}{h}(x_p^n - x_{\boldsymbol{i}})) \cdot N(\frac{1}{h}(y_p^n - y_{\boldsymbol{i}})) \cdot \frac{1}{h}\nabla N(\frac{1}{h}(z_p^n - z_{\boldsymbol{i}})) \end{pmatrix} \quad [18] \tag{53}$$

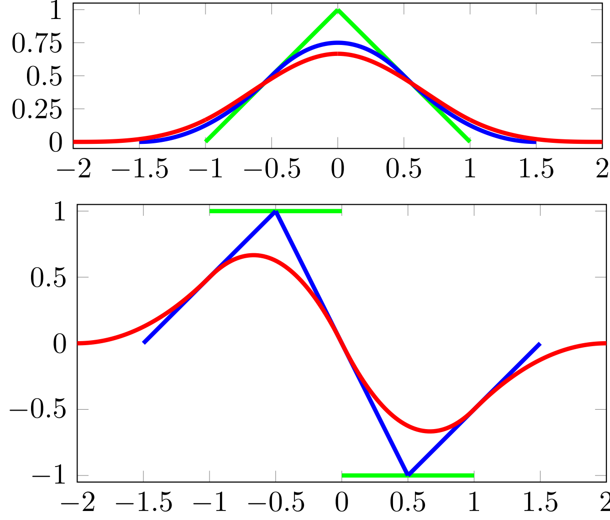


Figure 4: Plots of the piecewise linear (green), quadratic (blue) and cubic (red) basis functions (top) and their derivatives (bottom).

In order for the mass transfer in Eq. (38) to be correct, the total particle mass has to be equivalent to the total node mass:

$$\sum_{\boldsymbol{i}} m_{\boldsymbol{i}} = \sum_{p} m_p. \tag{54}$$

This is true if the basis functions have the *partition of unity* property [18]

$$\forall \boldsymbol{x} : \sum_{\boldsymbol{i}} N_{\boldsymbol{i}}^h(\boldsymbol{x}) = 1. \tag{55}$$

Using this, Eq. (54) holds because

$$\sum_{\boldsymbol{i}} m_{\boldsymbol{i}} = \sum_{\boldsymbol{i}} \sum_{p} m_p \omega_{\boldsymbol{i}p}^n = \sum_{p} m_p \sum_{\boldsymbol{i}} \omega_{\boldsymbol{i}p}^n = \sum_{p} m_p. \tag{56}$$

Piecewise linear basis functions are a common choice in FEM because they are cheap to compute and have a compact support [20]. They can be defined using

$$N(x) = \max(1 - |x|, 0) \quad \text{and} \tag{57}$$

$$\nabla N(x) = \begin{cases} 1 & -1 \leq x < 0 \\ -1 & 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}. \tag{58}$$

This is only C_0 continuous. While these basis functions work for FEM, they are considered unstable with respect to MPM [18]. The reason for this is that

the integration points in MPM are particles which are not fixed. Instead, they move across discontinuities in the derivative of the basis functions due to being advected, causing large errors. This is known as *cell-crossing instability* [20].

C_1-continuity can be achieved using quadratic B-splines defined by

$$N(x) = \begin{cases} \frac{3}{4} - x^2 & 0 \leq |x| < \frac{1}{2} \\ \frac{1}{2}(\frac{3}{2} - |x|)^2 & \frac{1}{2} \leq |x| < \frac{3}{2} \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \tag{59}$$

$$\nabla N(x) = \begin{cases} -2x & 0 \leq |x| < \frac{1}{2} \\ x + \frac{3}{2} & -\frac{1}{2} \geq x > -\frac{3}{2} \\ x - \frac{3}{2} & \frac{1}{2} \leq x < \frac{3}{2} \\ 0 & \text{otherwise} \end{cases} . \tag{60}$$

Given the particle spacing Δx, this leads to errors in $O(\Delta x^2)$ compared to $O(\Delta x)$ for linear basis functions [20]. If C_2-continuity is desired, cubic B-splines can be defined using

$$N(x) = \begin{cases} \frac{1}{2}|x|^3 - x^2 + \frac{2}{3} & 0 \leq |x| < 1 \\ \frac{1}{6}(2 - |x|)^3 & 1 \leq |x| < 2 \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \tag{61}$$

$$\nabla N(x) = \begin{cases} -\frac{3}{2}x^2 - 2x & 0 \geq x > -1 \\ \frac{3}{2}x^2 - 2x & 0 \leq x < 1 \\ \frac{1}{2}(2 + x)^2 & -1 \geq x > -2 \\ -\frac{1}{2}(2 - x)^2 & 1 \leq x < 2 \\ 0 & \text{otherwise} \end{cases} . \tag{62}$$

Figure 4 shows plots of all three basis functions as well as their derivatives.

Cubic B-splines reduce errors to $O(\Delta x^3)$ if the particles are spaced globally uniform. In less ideal particle distributions they instead behave as $O(\Delta x^2)$ just like quadratic B-splines [20].

Quadratic and cubic B-splines are the most popular choices for MPM. The former are more efficient to compute and have a smaller support which increases efficiency further, but lead to higher numerical errors. Cubic B-splines make sense for engineering applications that need low numerical error, while graphic applications may prefer the performance benefit of quadratic B-splines as long as the visual result is not impacted too much [18].

It is possible to use different basis functions for different transfer operations. For example, Stomakhin *et al.* [9] use cubic B-splines when transferring to the grid and quadratic B-splines when updating particle velocities.

4.2.2 Particle-In-Cell Method

PIC has been used as a solver for hydrodynamics and other problems since 1955 [21]. If PIC is used to resolve transfers between particles and grid, velocities are calculated in Step 1 as

$$m_{\boldsymbol{i}}^n \boldsymbol{v}_{\boldsymbol{i}}^n = \sum_p m_p \omega_{\boldsymbol{i}p}^n \boldsymbol{v}_p^n \quad [1] \tag{63}$$

and then normalized by mass. After force application and collision on the grid, velocities are transferred back to particles in Step 8 using

$$\boldsymbol{v}_{\mathrm{PIC}p}^{n+1} = \sum_{\boldsymbol{i}} \omega_{\boldsymbol{i}p}^n \hat{\boldsymbol{v}}_{\boldsymbol{i}}^{n+1} \quad [11]. \tag{64}$$

This approach is stable, but leads to severe dampening, especially for angular momentum. The reason for this is that the transfer causes information loss, as the particles have more degrees of freedom than the cells [11]. Additionally, there are usually multiple particles per each cell. This causes aliasing and is the reason for the *ringing instability* described by Brackbill [22], resulting in artifacts on particle positions.

4.2.3 Fluid Implicit Particle Method

FLIP was developed by Brackbill *et al.* [23] in order to address the strong dissipative properties of PIC. The method later became the norm for fluid simulations in graphics [11]. Transferring velocities from particles to the grid is done in FLIP the same way it is done in PIC using Eq. (63). The other direction is defined as

$$\boldsymbol{v}_{\mathrm{FLIP}p}^{n+1} = \boldsymbol{v}_p^n + \sum_{\boldsymbol{i}} \omega_{\boldsymbol{i}p}^n (\hat{\boldsymbol{v}}_{\boldsymbol{i}}^{n+1} - \boldsymbol{v}_{\boldsymbol{i}}^n) \quad [1]. \tag{65}$$

This preserves the particle velocities and only applies the velocity change of the grid on top of them. FLIP is however very noisy as a result of this change and the ringing instability is amplified, which is why FLIP is considered unstable [11, 24].

In order to increase FLIP stability while not introducing too much dampening, FLIP and PIC are often combined as

$$\boldsymbol{v}_p^{n+1} = \alpha \boldsymbol{v}_{\mathrm{FLIP}p}^{n+1} + (1-\alpha) \boldsymbol{v}_{\mathrm{PIC}p}^{n+1}. \tag{66}$$

Stomakhin *et al.* [1] use $\alpha = 0.95$.

4.2.4 Affine Particle-In-Cell Method (APIC)

Jiang *et al.* [11] developed APIC in order to alleviate the problems FLIP has. They initially developed the Rigid Particle-In-Cell Method (RPIC)

which preserved angular momentum of particles. This was insufficient and they subsequently extended it to preserve shearing modes as well. APIC describes the velocity as locally affine and stores this information on particles in addition to their velocity in a matrix $\boldsymbol{C}_p^n$. The grid velocity in Step 1 is then calculated using both:

$$m_{\boldsymbol{i}}^n \boldsymbol{v}_{\boldsymbol{i}}^n = \sum_p m_p \omega_{\boldsymbol{i}p}^n (\boldsymbol{v}_p^n + \boldsymbol{C}_p^n (\boldsymbol{x}_{\boldsymbol{i}} - \boldsymbol{x}_p^n)) \tag{67}$$

The affine matrix is defined as $\boldsymbol{C}_p^n = \boldsymbol{B}_p^n (\boldsymbol{D}_p^n)^{-1}$. $\boldsymbol{B}_p^0$ is initialized to $\boldsymbol{I}$ and $\boldsymbol{D}_p^n$ is defined as

$$\boldsymbol{D}_p^n = \sum_{\boldsymbol{i}} \omega_{\boldsymbol{i}p}^n (\boldsymbol{x}_{\boldsymbol{i}} - \boldsymbol{x}_p^n)(\boldsymbol{x}_{\boldsymbol{i}} - \boldsymbol{x}_p^n)^T, \tag{68}$$

which is similar to an inertia tensor. If quadratic or cubic B-spline basis functions are used, $\boldsymbol{D}_p^n$ is a constant, namely $\frac{1}{4}h^2\boldsymbol{I}$ and $\frac{1}{3}h^2\boldsymbol{I}$, respectively. $\boldsymbol{B}_p^n$ is evolved in Step 8 using

$$\boldsymbol{B}_p^{n+1} = \sum_{\boldsymbol{i}} \omega_{\boldsymbol{i}p}^n \hat{\boldsymbol{v}}_{\boldsymbol{i}}^{n+1} (\boldsymbol{x}_{\boldsymbol{i}} - \boldsymbol{x}_p^n)^T. \tag{69}$$

Velocities are transferred from grid to particle the same way they are in PIC using Eq. (64).

APIC preserves angular momentum far better than PIC and FLIP and does not suffer from noise like FLIP does. It also does not suffer from the ringing instability, unlike FLIP and FLIP/PIC blends. FLIP preserves more momentum in the general case, with the exception being the aforementioned angular momentum. Experiments by Jiang *et al.* [11] show APIC's energy conservation roughly around the FLIP/PIC blends with $\alpha = 0.95$ and $\alpha = 0.99$, with APIC ahead of PIC, FLIP, RPIC and the FLIP/PIC blends for pure rotational scenarios. The improvements of APIC come at the cost of increased computational complexity and memory.

5 Constitutive Models

Material behavior is described based on constitutive models. Purely hyperelastic models attribute all deformations to stresses. Elasto-plastic constitutive models on the other hand define a yield criterion, which means that stresses that exceed the criterion result in plastic deformation that cannot be recovered from. The plastic deformation can further be used to modify how the material behaves elastically, i.e. by hardening via increased stiffness. This section will introduce a few common hyperelastic constitutive models as well as elasto-plastic models for snow and sand.

5.1 Hyperelasticity

Idealized elastic materials can be described using an energy density function Ψ, or alternatively by directly defining a notion of stress because of the relation $\boldsymbol{\sigma}^{\text{PK1}} = \frac{\partial \Psi}{\partial \boldsymbol{F}}$.

5.1.1 Saint Venant-Kirchhoff

The Saint Venant-Kirchoff model is one of the simplest constitutive models. It can be formed by using the Green-Lagrangian strain tensor $\boldsymbol{E}$ (Eq. (25)) inside Hooke's Law (Eq. (16)):

$$\boldsymbol{\sigma}^{\text{PK2}} = 2\mu \boldsymbol{E} + \lambda \operatorname{tr}(\boldsymbol{E})\boldsymbol{I} \tag{70}$$

This model is typically not used for large deformation scenarios because the Green-Lagrangian strain tensor is unsuitable for these cases [16].

5.1.2 Hencky Strain

An alternative to Saint Venant-Kirchhoff for large deformation scenarios can be derived from the Hencky strain:

$$\boldsymbol{\sigma} = \mu \ln(\boldsymbol{F}\boldsymbol{F}^T) + \frac{1}{2}\lambda \operatorname{tr}\big(\ln(\boldsymbol{F}\boldsymbol{F}^T)\big)\boldsymbol{I}. \tag{71}$$

The corresponding energy density function is usually written using the SVD $\boldsymbol{F} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$:

$$\Psi(\boldsymbol{F}) = \mu \operatorname{tr}\big(\ln(\boldsymbol{\Sigma})^2\big) + \frac{1}{2}\lambda \operatorname{tr}\big(\ln(\boldsymbol{\Sigma})\big)^2 \quad [2]. \tag{72}$$

As $\boldsymbol{\Sigma}$ is a diagonal matrix, $\ln(\boldsymbol{\Sigma})$ refers to applying the logarithm to each element. The derivative is

$$\boldsymbol{\sigma}^{\text{PK1}} = \frac{\partial \Psi}{\partial \boldsymbol{F}}(\boldsymbol{F}) = \boldsymbol{U}\Big(2\mu\boldsymbol{\Sigma}^{-1}\ln(\boldsymbol{\Sigma}) + \lambda \operatorname{tr}\big(\ln(\boldsymbol{\Sigma})\big)\boldsymbol{\Sigma}^{-1}\Big)\boldsymbol{V}^T \quad [2]. \tag{73}$$

5.1.3 Neo-Hookean

The Neo-Hookean constitutive model is commonly used in large deformation scenarios [18], and it is the simplest one for this use case [25]. Just like the other models introduced so far, it is nonlinear. The energy density function is

$$\Psi(\boldsymbol{F}) = \frac{\mu}{2}\big(\operatorname{tr}(\boldsymbol{F}^T\boldsymbol{F}) - d\big) - \mu \ln(J) + \frac{\lambda}{2}\big(\ln(J)\big)^2 \tag{74}$$

where d is the spatial dimension. The derivative is

$$\boldsymbol{\sigma}^{\text{PK1}} = \frac{\partial \Psi}{\partial \boldsymbol{F}}(\boldsymbol{F}) = \mu(\boldsymbol{F} - \boldsymbol{F}^{-T}) + \lambda \ln(J)\boldsymbol{F}^{-T}. \tag{75}$$

5.1.4 Fixed Corotated

While $J = \det(\boldsymbol{F})$ can never be zero or negative in reality, it can happen in simulations. In these circumstances, $\ln(J)$ becomes problematic. Stomakhin *et al.* [10] proposed the fixed corotated model as a solution that still behaves valid if configurations are inverted. The energy density is defined as

$$\Psi(\boldsymbol{F}) = \mu\|\boldsymbol{F} - \boldsymbol{R}\|_F^2 + \frac{\lambda}{2}(J-1)^2. \tag{76}$$

Here, $\boldsymbol{F} = \boldsymbol{R}\boldsymbol{U}$ is the polar decomposition and $\|\cdot\|_F$ is the Frobenius matrix norm. An alternative but equivalent definition uses the SVD,

$$\hat{\Psi}(\boldsymbol{\Sigma}(\boldsymbol{F})) = \mu \operatorname{tr}\left((\boldsymbol{\Sigma} - \boldsymbol{I})^2\right) + \frac{\lambda}{2}(J-1)^2. \tag{77}$$

The derivate is

$$\boldsymbol{\sigma}^{\text{PK1}} = \frac{\partial \Psi}{\partial \boldsymbol{F}}(\boldsymbol{F}) = 2\mu(\boldsymbol{F} - \boldsymbol{R}) + \lambda(J-1)J\boldsymbol{F}^{-T}. \tag{78}$$

5.1.5 Fluids

A simple constitutive model for fluids based on deformation gradients can be derived from the Navier-Stokes equation. According to Eqs. (36) and (37)

$$\nabla \cdot \boldsymbol{\sigma} = -\nabla p + \mu \nabla^2 \boldsymbol{v}. \tag{79}$$

If viscosity is set to 0, this simplifies to $\nabla \cdot \boldsymbol{\sigma} = -\nabla p$. The solution to this equation is $\boldsymbol{\sigma} = -p\boldsymbol{I}$ which is also the definition used by [3]. Using the definition of bulk modulus (Eq. (17)) and Eq. (20),

$$p = -\sigma_{\text{Hyd}} = -K \cdot \epsilon_{\text{Vol}} = -K \cdot (J-1) = K \cdot (1-J). \tag{80}$$

This definition is similar to the ideal gas law but results in high compressibility [26]. An alternative is Tait's equation

$$p = K\left(\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right) = K\left(\frac{1}{J^{\gamma}} - 1\right), \tag{81}$$

where γ is used to punish large deviations from incompressibility more [3, 26]. Accordingly,

$$\boldsymbol{\sigma}^{\text{PK1}} = \frac{\partial \Psi}{\partial \boldsymbol{F}}(\boldsymbol{F}) = J\boldsymbol{\sigma}\boldsymbol{F}^{-T}. \tag{82}$$

Due to Eq. (41), this model depends only on the determinant of the deformation gradient when used in MPM. This simplifies some steps. Instead of Eq. (42), Eq. (41) can be used directly. No deformation gradient has to be stored, instead, the determinant is evolved as

$$J_p^{n+1} = \left(1 + \Delta t \operatorname{tr}(\nabla \boldsymbol{v}_p^{n+1})\right) J_p^n \quad [3]. \tag{83}$$

5.2 Plasticity

Hyperelastic materials alone only provide a limited variety of material behavior. Real materials cannot be stretched or compressed infinitely. They instead yield at some point, resulting in irreversible plastic deformations, which can also mean fracturing. In multiplicative plasticity theory, the deformation gradient is split into two parts according to Eq. (28) based on its yield criterion. The models described in the following also undergo hardening based on the degree of plastic deformation, which further changes material behavior.

5.2.1 Snow

Stomakhin *et al.* [1] proposed an elasto-plastic model for snow simulation. It successfully captures fracturing as well as the sticky behavior of snow that results in the *packing snow effect*. Compression of snow is also handled and the model allows to change the characteristics of the material enough to model a wide variety of snow types.

The yield criterion is defined in terms of a *critical compression* θ_c and *critical stretch* θ_s. Using the SVD $\hat{\boldsymbol{F}}_{Ep}^{n+1} = \boldsymbol{U}\hat{\boldsymbol{\Sigma}}\boldsymbol{V}^T$, the singular values are clamped to the range $[1-\theta_c, 1+\theta_s]$. In other words, the portion of the singular values exceeding these thresholds is attributed to plastic deformation. The corrected elastic deformation gradient is then defined using the clamped singular values $\boldsymbol{\Sigma}$:

$$\boldsymbol{F}_{Ep}^{n+1} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T \tag{84}$$

and the plastic portion is updated as

$$\boldsymbol{F}_{Pp}^{n+1} = \left(\boldsymbol{F}_{Ep}^{n+1}\right)^{-1}\boldsymbol{F}_{p}^{n+1} = \boldsymbol{V}\boldsymbol{\Sigma}^{-1}\boldsymbol{U}^T\boldsymbol{F}_{p}^{n+1}. \tag{85}$$

Furthermore, the model implements *hardening*. Plastic compression thus hardens the material, while plastic stretch would soften it. This is implemented by defining the Lamé parameters as a function of the plastic deformation:

$$\mu(\boldsymbol{F}_P) = \mu_0 e^{\xi(1-J_P)} \quad \text{and} \quad \lambda(\boldsymbol{F}_p) = \lambda_0 e^{\xi(1-J_P)}, \tag{86}$$

which corresponds to a change of Young's modulus. μ_0 and λ_0 are the Lamé parameters in the absence of plastic deformation. ξ controls the strength of the hardening effect and $J_P = \det(\boldsymbol{F}_P)$.

It is worth noting that the model does not actually require $\boldsymbol{F}_P$ to be computed, as only the determinant is ever required [18]. Because the determinants multiply, Eqs. (48) and (49) can be written in terms of the determinants:

$$\hat{J}_{Ep}^{n+1} = J_{Ep}^{n+1} J_{Pp}^{\star} \quad \text{and} \quad J_{Pp}^{n+1} = J_{Pp}^{\star}\hat{J}_{Pp}^{n+1}. \tag{87}$$

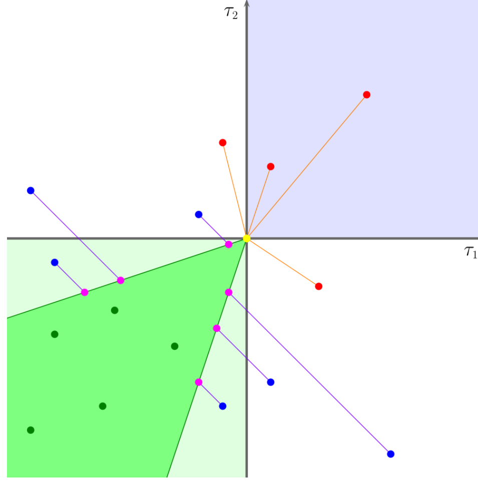


Figure 5: The Drucker-Prager yield surface in 2D principal stress space. Green points correspond to Case I. Blue points belong to Case II and are projected to the yield surface (purple points). Lastly, Case III is represented by the red points which are projected to the tip of the cone. Image source: [2].

This means that the determinant can be evolved as

$$J_{Pp}^{n+1} = \frac{\hat{J}_{Ep}^{n+1}}{J_{Ep}^{n+1}} \hat{J}_{Pp}^{n+1} = \frac{\hat{J}_{Ep}^{n+1}}{J_{Ep}^{n+1}} J_{Pp}^{n} = \frac{\det(\hat{\boldsymbol{\Sigma}})}{\det(\boldsymbol{\Sigma})} J_{Pp}^{n}. \tag{88}$$

Because $\boldsymbol{\Sigma}$ and $\hat{\boldsymbol{\Sigma}}$ are diagonal, their determinants can be computed by multiplying the singular values.

Stomakhin *et al.* [1] used the fixed corotated constitutive model to handle elasticity. Their approach to plasticity is however independent of that and could be used with other hyperelastic models, although this might produce different results.

5.2.2 Sand

Klár *et al.* [2] implemented an elasto-plastic model based on the Drucker-Prager model for plastic flow [27]. Just like Mast [28], they use the constitutive model based on Hencky strain (Section 5.1.2) to describe the elastic behavior.

The elastic portion of the deformation gradient is updated as

$$\boldsymbol{F}_{Ep}^{n+1} = \boldsymbol{Z}(\hat{\boldsymbol{F}}_{Ep}^{n+1}, \alpha_p^n). \tag{89}$$

Here, $\boldsymbol{Z}(\cdot,\cdot)$ is the operator that projects to the yield surface and α_p^n is the current hardening state of the particle which describes the friction between grains and affects the yield surface.

The Drucker-Prager yield condition is defined as

$$\alpha_p^n \operatorname{tr}(\boldsymbol{\sigma}) + \|\boldsymbol{\sigma} - \frac{\operatorname{tr}(\boldsymbol{\sigma})}{d}\boldsymbol{I}\|_F \leq 0 \quad [3]. \tag{90}$$

In their technical document, Klár *et al.* [2] derive that this imposes a constraint on the principal stresses if the Hencky strain is used. The result is that the yield surface is a cone in principal stress space. They define three cases (see also Fig. 5)

I) The stress is in the yield surface and thus responds elastically with static friction between grains.

II) The sand is expanding, so the grains can move freely without friction. The deformation gradient is projected to the tip of the cone.

III) The sand is under compression but the shear stress is too high for friction to compensate. This means dynamic friction and projection to the side of the cone.

The deformation gradient is first decomposed using the SVD $\hat{\boldsymbol{F}}_{Ep}^{n+1} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$. Using

$$\hat{\boldsymbol{\epsilon}} = \boldsymbol{\epsilon} - \frac{\operatorname{tr}(\boldsymbol{\epsilon})}{d}\boldsymbol{I} \quad \text{and} \quad \delta\gamma = \|\hat{\boldsymbol{\epsilon}}\|_F + \frac{d\lambda + 2\mu}{2\mu}\operatorname{tr}(\boldsymbol{\epsilon})\alpha_p^n \tag{91}$$

where $\boldsymbol{\epsilon} = \ln(\boldsymbol{\Sigma})$, the cases can be resolved. If $\delta\gamma \leq 0$, nothing has to be done (Case I). Otherwise, if $\operatorname{tr}(\boldsymbol{\epsilon}) > 0$ or $\|\hat{\boldsymbol{\epsilon}}\|_F = 0$, the deformation gradient has to be projected to the tip (Case II). This is done as $\boldsymbol{F}_{Ep}^{n+1} = \boldsymbol{U}\boldsymbol{V}^T$, i.e. all stress is removed. All other situations map to Case III. This requires projecting by computing

$$\boldsymbol{H} = \boldsymbol{\epsilon} - \delta\gamma\frac{\hat{\boldsymbol{\epsilon}}}{\|\hat{\boldsymbol{\epsilon}}\|_F}. \tag{92}$$

The deformation gradient is then formed as $\boldsymbol{F}_{Ep}^{n+1} = \boldsymbol{U}e^{\boldsymbol{H}}\boldsymbol{V}^T$. $\boldsymbol{H}$ is a diagonal matrix so $e^{\boldsymbol{H}}$ means to apply the operation to each entry. An important property for Case III is that the determinant of the deformation gradient does not change due to the projection, i.e. $\hat{J}_{Ep}^{n+1} = J_{Ep}^{n+1}$, which means that undesired volume loss is prevented. This is however not true in Case II which is the expanding case. This means that volume gain is not prevented. The result is that the volume can expand drastically over time in some scenarios. Tampubolon *et al.* [3] developed a modification of the model that fixes volume gain. They add another particle property v_{cp}^n which is the logarithm of the plastic deformation gradient's determinant, in other words the logarithmic volume change due to plasticity. It can be evolved as

$$v_{cp}^{n+1} = v_{cp}^n + \ln(J_{Ep}^{n+1}) - \ln(\hat{J}_{Ep}^{n+1}) \tag{93}$$

with $v_{cp}^0 = 0$. They then replace $\boldsymbol{\epsilon}$ in the previous equations with $\boldsymbol{\epsilon} + \frac{v_{cp}^n}{d}\boldsymbol{I}$. This greatly reduces the volume gain artifacts, as can be seen in Fig. 6.

Figure 6: Sand is poured from a spout without the volume gain fix (top) and with it (bottom).

Hardening is based on the model of Mast *et al.* [29]. It is dependent on the amount of plasticity, meaning that no hardening occurs for Case I. The hardening state is accumulated in a new particle property q_p^n and it is evolved by adding δq_p to it every timestep. For Case II, $\delta q_p = \|\boldsymbol{\epsilon}\|_F$ because all stress is taken away. In Case III, $\delta q_p = \delta\gamma$. The change in hardening δq_p is always positive. Based on this hardening state, the friction angle is defined as

$$\phi_{Fp}^n = h_0 + (h_1 q_p^n - h_3)e^{-h_2 q_p^n}. \tag{94}$$

h_0 through h_3 are material parameters. They should follow the condition that $h_0 > h_3 \geq 0$ and $h_1, h_2 \geq 0$. If $h_1 = h_2 = 0$, then no hardening takes place and the friction angle is instead fixed to $h_0 - h_3$. If $h_1 = 0$ and $h_2 > 0$, then the friction angle starts at $h_0 - h_3$ and eventually increases to h_0. If h_1 is also greater than zero, then the function will start at $h_0 - h_3$, rise to a maximum and then fall back to h_3. The friction angle should be in the interval $[0, \frac{\pi}{2})$, i.e. between 0 and 90 degrees (exclusive). A friction angle of 0 means that the sand behaves like a fluid. Finally, α_p^n is defined as

$$\alpha_p^n = \sqrt{\frac{2}{3}} \frac{2\sin(\phi_{Fp}^n)}{3 - \sin(\phi_{Fp}^n)}. \tag{95}$$

So far, the model can only describe dry sand. Tampubolon *et al.* [3] further extend it to wet sand by adding cohesion to the model. The yield condition is modified to

$$\alpha_p^n \operatorname{tr}(\boldsymbol{\sigma}) + \|\boldsymbol{\sigma} - \frac{\operatorname{tr}(\boldsymbol{\sigma})}{d}\boldsymbol{I}\|_F \leq c_C \quad [3]. \tag{96}$$

with $c_C \geq 0$ being the cohesion. The projection procedure described previously is however actually defined in terms of the Kirchhoff stress $\boldsymbol{\tau} = J\boldsymbol{\sigma}$.

This did not matter while the right-hand side of the yield condition was zero, but it is relevant now. The condition in terms of Kirchhoff stress can be derived by dividing through J:

$$\alpha_p^n \operatorname{tr}(\boldsymbol{\tau}) + \|\boldsymbol{\tau} - \frac{\operatorname{tr}(\boldsymbol{\tau})}{d}\boldsymbol{I}\|_F \leq \frac{c_C}{J}. \tag{97}$$

The projection procedure can now be modified to support cohesion by redefining $\delta\gamma$:

$$\delta\gamma = \|\hat{\boldsymbol{\epsilon}}\|_F + \frac{d\lambda + 2\mu}{2\mu} \operatorname{tr}(\boldsymbol{\epsilon})\alpha_p^n - \frac{c_C}{J}. \tag{98}$$

Similar to snow plasticity, the plastic deformation gradient does not have to be stored and evolved. It is replaced by the hardening state q_p^n as well as the logarithm of its determinant v_{cp}^n if the volume fix is used.

The model as presented works well with explicit time stepping. Semi-implicit time stepping however introduces strong artificial cohesion. Tampubolon *et al.* [3] thus developed a unilateral modification to the energy density function that removes this effect. This however introduced additional spreading and non-physical behavior during column collapse. Gao *et al.* [4] thus proposed yet another modification that also solves these issues. None of these modifications are necessary if explicit time stepping is used.

Part III

Method and Implementation

6 Method

This section will introduce different approaches to achieve material variety that are solely based on the MPM and not achieved through coupling MPM with different solvers (Section 6.1). After that, Section 6.2 will discuss the different choices that exist with respect to the MPM itself, such as choosing between PIC, FLIP or APIC and explain the decisions that were made in context with the requirements that Section 6.1 imposes.

6.1 Multiple Materials

The MPM described in Section 4 so far assumed the usage of a single constitutive model with fixed material parameters. This is perfectly reasonable for many scenarios. For example, simulating how an object made of a specific material behaves under stress does not need any variety and is still a common task in engineering. Many graphics applications would also only need a single material to be simulated. Thus, research is often focused on how to solve a specific problem in an optimal way. This section will instead propose a general MPM framework that can integrate arbitrary constitutive models with different, user-controllable interactions in between them.

6.1.1 Single Constitutive Model

The first step at introducing material variety would be varying the initial particle state, which is already possible in the MPM framework from Section 4. The quantities stored on particles are position, mass, velocity, initial volume and elastic deformation gradient. Out of these, only varying mass would make sense to vary material behavior. Portions of an object with higher mass would receive less acceleration due to stresses and vice-versa, which would lead to non-uniform behavior across an object. Positions and velocities are set based on where objects are and in which direction they should initially move, while a deformation gradient initialized to something other than $\boldsymbol{I}$ simply describes a pre-deformed object.

In case of constitutive models with plasticity, there are additional properties on particles. Both the snow and sand model add a hardening state, while the sand model can also add a scalar for the volume fix. Initializing the latter one to a value different from 0 would counteract its purpose. Varying the initial hardening state however can be used to create non-uniform material behavior. In case of the snow model, varying hardening state would translate to a spatially varying Young's modulus, which is the material's stiffness.

Based on this observation, there is no reason to restrict the other material parameters to be constant. Instead, all the parameters of the constitutive model can become properties of the particles itself. This approach was also chosen by Stomakhin *et al.* [1] in order to achieve more realistic results with their snow model. They spatially vary both mass as well as material parameters, which can create more interesting fracture amongst other things.

It is more general to define the elasto-plastic energy density function as

$$\Psi(\boldsymbol{F}_{Ep}, \boldsymbol{F}_{Pp}, \lambda_p, \mu_p) \tag{99}$$

where λ_p and μ_p are now particle properties. The parameters are further extended depending on the constitutive model, i.e. with ξ_p for snow hardening. Additionally, the different parameters for plasticity such as θ_c can also be defined on a per-particle basis.

So far this section concentrated on introducing non-uniform material behavior for single objects. The approach can however also be used by introducing drastically varying parameters across multiple objects, as was shown in [9]. Some constitutive models are capable of expressing distinctly different materials. For example, if $\mu = 0$ for the fixed corotated model, then it depends only on the determinant. This is similar to the model for fluids (Section 5.1.5) and is also how the model behaves in this case [9]. The snow model is based on the fixed corotated model and it is possible to opt-out of the plasticity by setting $\theta_c = 1$ and θ_s to some very large value. This means that the same model can be used to simulate fluids, hyperelastic materials as well as snow and many other elasto-plastic materials at the same time in the same scene. This is demonstrated by Stomakhin *et al.* [9], although their work uses a variation of the original model and further augments MPM to be able to handle a wider range of stiffness and degrees of incompressibility. A similar case can be made for the sand model. If hardening is not used and the friction angle is set to zero, it can be used to model a fluid. Varying levels of cohesion can be used to model dry or wet sand, while very high cohesion levels behave less and less like sand and eventually become hyperelastic.

6.1.2 Multiple Constitutive Models

The previous section concluded that a single constitutive model can already introduce a wide variety of materials. This does however come with limitations. While fluids can be simulated with the fixed corotated model, the model for fluids from Section 5.1.5 was specifically designed for this and uses Tait's equation to achieve less compressible fluids. On the other hand, the snow model cannot be used to model sand and vice-versa. This is not surprising as each model was designed for that specific purpose. Thus, the approach is not yet flexible enough, as the choice of constitutive model restricts the possibilities.

The solution is to make the constitutive model itself a property of each particle. All models that were introduced in this thesis can be combined into

$$\Psi_p(\boldsymbol{F_E}, \boldsymbol{F_P}, \lambda_p, \mu_p, \xi_p, \boldsymbol{h}_p) \tag{100}$$

where $\boldsymbol{h}_p = (h_{0p}, h_{1p}, h_{2p}, h_{3p})$. This includes all parameters needed for the elastic portion of the different models. The calculation of forces in MPM (Eq. (42)) can then be redefined as

$$\boldsymbol{f}_{\boldsymbol{i}}^n = -\sum_p V_p^0 \frac{\partial \Psi_p}{\partial \boldsymbol{F}_E} (\boldsymbol{F}_{Ep}^n)^T \nabla \omega_{\boldsymbol{i}p}^n. \tag{101}$$

Additionally, the plasticity treatment in Step 7 of the algorithm is also selected depending on the constitutive model of each particle, and parameters such as θ_c, θ_s and c_C become particle properties as well.

The reason that this approach works without any further changes is due to the Eulerian grid. Just as the grid automates self-collision if only a single constitutive model is used [1], it acts the same way if multiple constitutive models are used, as all interaction is still done through first transferring onto the grid and then back. The grid stays oblivious of the actual materials that are being used. Thus, collision between objects of varying materials based on any constitutive model is possible.

6.1.3 Porous Materials

Due to the automatic self-collision handling of the MPM, the current approach is not capable of simulating porous material interactions. In order to simulate a fluid entering into a porous material, particles of both species would have to be able to occupy the same space which is prevented by the method. This can be resolved if two grids are used, one for each of the species. The interaction between them then has to be resolved on the grids.

One such approach was published by Tampubolon *et al.* [3] for sand and water mixtures. They simulate water using the model from Section 5.1.5 and developed the volume fixed sand model with cohesion from Section 5.2.2. Each of them are simulated separately using the MPM with their own grid. Quantities belonging to the water species are denoted by a superscript w while the ones belonging to the sand species are denoted with a superscript s. The interaction between the two species is resolved using the momentum exchange terms $\boldsymbol{p}^s$ and $\boldsymbol{p}^w$ from [30] defined as

$$\boldsymbol{p}^s = c_E(\boldsymbol{v}^w - \boldsymbol{v}^s) + p^w \nabla \phi^w, \quad \boldsymbol{p}^w = -\boldsymbol{p}^s. \tag{102}$$

Here, c_E is the drag coefficient. It is defined based on the sand porosity n, the fluid density ρ^w, gravitational acceleration g and sand permeability $\hat{k}$ as $c_E = \frac{n^2 \rho^w g}{\hat{k}}$. On the right side, p^w is the pressure of the fluid and $\phi^w = \frac{\rho^w}{\rho^w + \rho^s}$ is

the volume fraction of the fluid. The left part of the term $\big(c_E(\boldsymbol{v}^w - \boldsymbol{v}^s)\big)$ is dissipative and results in something similar to Coulomb-friction [31]. It models viscous forces due to sand particles that move through the fluid. The corresponding discretized forces that are applied to the grid are

$$\boldsymbol{f}_{\boldsymbol{i}}^s = c_E m_{\boldsymbol{i}}^{s,n} m_{\boldsymbol{i}}^{w,n} (\boldsymbol{v}_{\boldsymbol{i}}^{w,n} - \boldsymbol{v}_{\boldsymbol{i}}^{s,n}), \quad \boldsymbol{f}_{\boldsymbol{i}}^w = -\boldsymbol{f}_{\boldsymbol{i}}^s. \tag{103}$$

The right part is the buoyancy term. This term is not dissipative, which is why it is not used in [31]. Tampubolon *et al.* [3] did implement it for one example in their paper but otherwise omitted it. They discretize it by adding

$$m_{\boldsymbol{i}}^{s,n} m_{\boldsymbol{i}}^{w,n} \sum_p -p_p^{w,n} \nabla \omega_{\boldsymbol{i}p}^n \frac{m_{\boldsymbol{i}}^{w,n}}{m_{\boldsymbol{i}}^{s,n} + m_{\boldsymbol{i}}^{w,n}} \tag{104}$$

to the grid force $\boldsymbol{f}_{\boldsymbol{i}}^w$ and subtracting it from $\boldsymbol{f}_{\boldsymbol{i}}^s$.

In addition to the momentum exchange term, Tampubolon *et al.* [3] vary material behavior based on the water volume fraction. They assumed sand to be initially wet to the degree that gives it its maximum cohesion $c_{Cp}^{s,0}$. Any further addition of fluid then decreases cohesion, so they define it as a function of the volume fraction:

$$c_{Cp}^{s,n+1} = c_{Cp}^{s,0}(1 - \phi_p^{n+1}). \tag{105}$$

They discretize the volume fraction as

$$\phi_{\boldsymbol{i}}^{n+1} = \begin{cases} 1 & m_{\boldsymbol{i}}^{w,n+1} > 0 \text{ and } m_{\boldsymbol{i}}^{s,n+1} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{106}$$

$$\phi_p^{n+1} = \sum_{\boldsymbol{i}} \omega_{\boldsymbol{i}p}^n \phi_{\boldsymbol{i}}^{n+1}. \tag{107}$$

This definition is however different from how ϕ^w is defined, as it is independent of the actual densities and thus symmetric for both sand and water.

Due to the stiff terms in the momentum exchange, Tampubolon *et al.* [3] used a semi-implicit approach to keep time steps small, which required them to adapt the constitutive model to deal with artificial cohesion (see Section 5.2.2). In this thesis, semi-implicit time stepping is not used. This required a closer look at the drag forces in Eq. (103). If c_E is too large for a given time step, the force would result in both grid velocities becoming further apart instead of slowly being equalized. Thus, in order to keep the simulation stable, a reasonable condition is that both velocities should at most become equal during a single timestep. This leads to the following equations:

$$\begin{aligned} & \boldsymbol{v}_{\boldsymbol{i}}^s + \Delta t \cdot c_E m_{\boldsymbol{i}}^w (\boldsymbol{v}_{\boldsymbol{i}}^w - \boldsymbol{v}_{\boldsymbol{i}}^s) = \boldsymbol{v}_{\boldsymbol{i}}^w - \Delta t \cdot c_E m_{\boldsymbol{i}}^s (\boldsymbol{v}_{\boldsymbol{i}}^w - \boldsymbol{v}_{\boldsymbol{i}}^s) \\ \iff & \boldsymbol{v}_{\boldsymbol{i}}^s - \boldsymbol{v}_{\boldsymbol{i}}^w + 2\Delta t \cdot c_E (m_{\boldsymbol{i}}^w + m_{\boldsymbol{i}}^s)(\boldsymbol{v}_{\boldsymbol{i}}^w - \boldsymbol{v}_{\boldsymbol{i}}^s) = \boldsymbol{0} \\ \iff & (\boldsymbol{v}_{\boldsymbol{i}}^s - \boldsymbol{v}_{\boldsymbol{i}}^w) - 2\Delta t \cdot c_E (m_{\boldsymbol{i}}^w + m_{\boldsymbol{i}}^s)(\boldsymbol{v}_{\boldsymbol{i}}^s - \boldsymbol{v}_{\boldsymbol{i}}^w) = \boldsymbol{0} \\ \iff & \big(1 - 2\Delta t \cdot c_E (m_{\boldsymbol{i}}^w + m_{\boldsymbol{i}}^s)\big)(\boldsymbol{v}_{\boldsymbol{i}}^s - \boldsymbol{v}_{\boldsymbol{i}}^w) = \boldsymbol{0} \\ \iff & 2\Delta t \cdot c_E (m_{\boldsymbol{i}}^w + m_{\boldsymbol{i}}^s) = 1 \text{ or } \boldsymbol{v}_{\boldsymbol{i}}^s - \boldsymbol{v}_{\boldsymbol{i}}^w = \boldsymbol{0}. \end{aligned} \tag{108}$$

Using this, the maximum possible drag coefficient can be derived given the time step and vice-versa:

$$c_{E,\max,\boldsymbol{i}} = \frac{1}{2\Delta t(m^w_{\boldsymbol{i}} + m^s_{\boldsymbol{i}})}, \tag{109}$$

$$\Delta t_{\max,\boldsymbol{i}} = \frac{1}{2c_E(m^w_{\boldsymbol{i}} + m^s_{\boldsymbol{i}})}. \tag{110}$$

Both of these are dependent on the masses and thus dependent on the scale of the scene as well as the grid resolutions. This also means that these limits vary throughout grid cells and time. In practice, the actual timestep should be even smaller because an immediate equalizing of the velocities results in collision behavior instead of drag. However, an implementation could use these formulae to limit the drag in cases where it gets too large, thus preventing a catastrophic failure in favor of a less accurate simulation. Alternatively it could be clamped to a value that is smaller than the maximum, which would not result in collision behavior.

An interesting side result of these derivations is that if the drag coefficient is always set to its maximum allowed value for each individual node, single grid behavior is reconstructed as this amounts to a collision response in any case.

In summary, the core ideas of the approach in [3] are:

1. Split the porous solid and the fluid into two species simulated separately with MPM.
2. Resolve species interaction between both grids with a drag-based momentum exchange term.
3. Decrease sand cohesion based on how much fluid is present locally.

Only (3) is dependent on the actual materials that are used. This idea can be generalized to arbitrary materials by defining all parameters of a constitutive model as functions of the volume fraction. This can apply to both species. Thus, Eq. (100) is extended to

$$\Psi_p(\boldsymbol{F_E}, \boldsymbol{F_P}, \lambda_p(\phi_p), \mu_p(\phi_p), \xi_p(\phi_p), \boldsymbol{h}_p(\phi_p)). \tag{111}$$

And consequently, the parameters for plasticity can also be varied based on the volume fraction. The actual functions can be defined arbitrarily depending on the use case. For example, a simulation with snow and water could have the snow become fluid itself if water is present, similar to how water melts snow in reality. It is also possible to have both sand and snow interacting with water at the same time in one scene, each reacting differently. Columns of sand could be made to collapse faster than in [3] by additionally reducing the friction angle based on the water volume fraction. A hyperelastic

material could be modeled as porous and become softer if filled with water, similar to a sponge. These terms can be specified by the user and adjusted to fine-tune the behavior of the simulation.

The approach is still limited in the range of porous phenomena that it can simulate the same way it is in [3]. This includes effects such as capillary action, which would require a more advanced grid interaction.

6.2 MPM Details

The approach introduced in the previous section is general enough to allow any combination of constitutive models to be used. This means the choice of details of the MPM, such as the kernel and the transfer scheme, have to be general enough to support this variety. Choosing these on a per-material basis would greatly complicate the approach and could mean that even more grids have to be used, even if the interaction is just collision.

6.2.1 Particle-Grid Transfer

While PIC is stable, it is also highly dissipative, which is why recent MPM publications do not use PIC. Stomakhin *et al.* [1] instead use a FLIP/PIC blend with $\alpha = 0.95$ for their snow model. Klár *et al.* [2] and Tampubolon *et al.* [3] on the other hand use APIC for their sand models because they proved unstable with FLIP. Both APIC as well as FLIP/PIC blends can be considered reasonable choices for a general, multiple constitutive model framework. As described in Section 4.2.4, APIC can compete with FLIP/PIC blends with respect to dissipative properties, but is significantly better at preserving rotational momentum. It is both stable and non-dissipative [11]. For these reasons, APIC was chosen for the grid transfers in this thesis. This does however come with a performance penalty, which will be alleviated using a different discretization approach known as the Moving Least Squares Material Point Method, which is introduced in Section 6.2.2.

6.2.2 Moving Least Squares Material Point Method

The Moving Least Squares Material Point Method (MLS-MPM) is a new spatial discretization proposed by Hu *et al.* [12]. It is derived from Moving Least Squares (MLS) [32] which is a local fitting scheme based on polynomial least-squares fits. MLS is used in many meshless methods, i.e. the element-free Galerkin (EFG) method [33]. The authors naturally derived APIC [11] as well as its generalization PolyPIC [34] from the MLS point of view and define new transfers that are more efficient than MPM based on APIC.

MLS-MPM can be formulated using either APIC or PolyPIC. Most steps are equivalent to traditional MPM using the respective transfer technique. The only differences are:

1. Velocity gradient approximation (Step 7, Eq. (47))
2. Force calculation (Step 3, Eq. (42))

In traditional MPM the velocity gradient $\nabla \boldsymbol{v}_p^{n+1}$ was estimated using a grid-to-particle transfer operation with the gradient of the weighting function. The MLS approximation is instead defined as

$$\nabla \boldsymbol{v}_p^{n+1} = \boldsymbol{C}_p^{n+1}, \tag{112}$$

where $\boldsymbol{C}_p^{n+1}$ is exactly the affine matrix from APIC (Section 4.2.4). This means that the deformation gradient is updated as

$$\hat{\boldsymbol{F}}_{Ep}^{n+1} = (\boldsymbol{I} + \Delta t \boldsymbol{C}_p^{n+1}) \boldsymbol{F}_{Ep}^n. \tag{113}$$

The affine matrix is already computed for APIC, which means that this approximation only reuses it and can skip a grid-to-particle transfer as well as the evaluation of gradients of the weighting function.

Force calculation in MPM is based on a particle-to-grid transfer using the gradient of the weighting function. MLS-MPM approximates this gradient as

$$\nabla \omega_{\boldsymbol{i}p}^n \approx M_p^{-1} \omega_{\boldsymbol{i}p}^n (\boldsymbol{x}_{\boldsymbol{i}}^n - \boldsymbol{x}_p^n). \tag{114}$$

where M_p^{-1} depends on whether quadratic or cubic B-splines are used and is equivalent to $\boldsymbol{D}_p^{-1}$ from APIC in this case. The force evaluation is now

$$\boldsymbol{f}_{\boldsymbol{i}}^n = -\sum_p V_p^0 \frac{\partial \Psi}{\partial \boldsymbol{F}_E} (\boldsymbol{F}_{Ep}^n)^T M_p^{-1} \omega_{\boldsymbol{i}p}^n (\boldsymbol{x}_{\boldsymbol{i}}^n - \boldsymbol{x}_p^n). \tag{115}$$

This approximation means that the gradient of the weighting function is not needed anymore in the method, which can provide performance benefits.

Another significant optimization is made possible by these changes. This is accomplished by fusing the transfer of momentum (Eq. (67)) with the force computation. It is achieved by computing the matrix

$$\boldsymbol{Q}_p = \Delta t V_p^0 \frac{\partial \Psi}{\partial \boldsymbol{F}_E} (\boldsymbol{F}_{Ep}^n)^T M_p^{-1} + m_p \boldsymbol{C}_p^n. \tag{116}$$

With this, the grid velocity update in Step 4 (Eq. (43)) is replaced with

$$m_{\boldsymbol{i}}^n \boldsymbol{v}_{\boldsymbol{i}}^\star = \sum_p \omega_{\boldsymbol{i}p}^n \big(m_p \boldsymbol{v}_p^n + \boldsymbol{Q}_p (\boldsymbol{x}_{\boldsymbol{i}}^n - \boldsymbol{x}_p^n) \big) \tag{117}$$

which reduces the amount of matrix-vector multiplications.

The MLS-MPM approximation can also be applied to the buoyancy term from Eq. (104). It changes to

$$m_{\boldsymbol{i}}^{s,n} m_{\boldsymbol{i}}^{w,n} \sum_p -p_p^{w,n} M_p^{-1} \omega_{\boldsymbol{i}p}^n (\boldsymbol{x}_{\boldsymbol{i}}^n - \boldsymbol{x}_p^n) \frac{m_{\boldsymbol{i}}^{w,n}}{m_{\boldsymbol{i}}^{s,n} + m_{\boldsymbol{i}}^{w,n}} \tag{118}$$

MLS-MPM based on APIC is significantly faster than traditional APIC MPM, which addresses the performance impact of using APIC over FLIP or PIC. Results from Gao *et al.* [4] show that APIC MLS-MPM can be faster than FLIP in their highly optimized GPU implementation, although this was not the case with the slower GVDB GPU implementation from Wu *et al.* [35].

6.2.3 Kernels

Quadratic and cubic B-splines are common choices in MPM. Stomakhin *et al.* [1] use cubic B-splines for their snow model. Klár *et al.* [2] make the same choice for their sand model, but mention that quadratic kernels work well too and subsequent work based on their sand model made this choice instead [3, 4]. Both kernels harmonize well with APIC as they reduce $\boldsymbol{D}_p$ to a constant, a property which is also integral to MLS-MPM. In this thesis, quadratic kernels are used when nothing else is specified due to their smaller support which results in faster execution. Cubic kernels are however supported as well, and their differences are explored in Section 9.

7 Implementation

7.1 Overview

The MPM framework introduced in the previous section was implemented using the GPU. A highly optimized implementation was recently published by [4] using CUDA. The authors made use of modern additions to the GPU instructions such as the *shuffle* instruction. They note that particle-to-grid (P2G) transfers are the most time-consuming step for GPU implementations, and that they take up about 90% of time in other solvers. In contrast to [35] which used a *gathering* approach for P2G, they used *scattering* and managed to make P2G 15 times faster in comparison. As a result, P2G amounts to only 40% of time in their solver and is nearly as fast as the grid-to-particle (G2P) transfer.

In this context, gathering means that, given a grid node, the nearby particles are gathered and their state is then transferred onto the node. This could be implemented using spatial data structures. In scattering on the other hand, a given particle scatters its state to all affected grid nodes, leading to write-conflicts due to the parallel nature of the GPU. A naive implementation could resolve these write-conflicts using atomic operations, which is however many times slower than the optimized approach in [4]. As an alternative to their approach, Gao *et al.* [4] mention the use of the GPU *rasterization pipeline*. According to them, a scattering approach based on this can not be easily extended to 3D Eulerian simulations. In this thesis, P2G transfer is implemented using the rasterization pipeline for both 2D and 3D simulations. The implementation is a proof-of-concept and does not focus on achieving

maximum speed. As is mentioned in [4], an approach like this would require a lot of complexity in order to map between sparse grids and the GPU textures needed for rasterization. For this reason, no sparse grids are used in this thesis. Instead, every simulation happens inside a bounded volume containing a fully allocated grid.

The general steps of a MPM framework (Section 4) as well as the extensions for MLS-MPM (Section 6.2.2) and porous material interaction (Section 6.1.3) can be grouped into 3 big steps based on which data is modified and the direction of data flow:

1. **P2G transfer:** This involves all steps that transfer information from particles to the grid, which can be identified in the equations as sums over particles. These are:

    - transfer of particle mass
    - velocity transfer using particle velocities and affine velocity matrix
    - transfer of particle stresses using deformation gradients and material state, resulting in grid forces
    - transfer of particle pressure if the buoyancy term is used

2. **Grid operations:** These are all the steps that only involve grid nodes and no particles. These are:

    - apply the drag term if it is used
    - apply the buoyancy term if it is used
    - apply collision on grid nodes

3. **G2P transfer:** This involves all steps that transfer information from the grid onto particles, which can be identified in the equations as sums over grid nodes. These are:

    - accumulate new particle velocities
    - accumulate affine particle velocity matrices
    - accumulate particle volume fractions if it is a two-grid simulation

    Furthermore, this step includes all steps that involve only particles:

    - evolve deformation gradients
    - treat evolved deformation gradients for plasticity
    - apply collision on particles if desired
    - advect particle positions

P2G is implemented using a program built from *vertex*, *geometry* and *fragment* shaders and thus uses the rasterization pipeline (Section 7.4.1). The grid operations are implemented using a *compute shader* executing one thread for each grid node (Section 7.4.2) and G2P is also implemented with the *compute pipeline* using one thread for each particle (Section 7.4.3). These three steps are executed sequentially in this order, once for each time step.

Before anything can be simulated, the state has to be initialized. For GPU implementations, this involves creating and filling *buffers* and *textures* (Section 7.3). The particles that are initialized are sample points of the actual objects that are simulated and there are multiple ways how this sampling can be implemented (Section 7.5). Computation of particle volumes is another G2P transfer and is thus implemented by executing the three steps with a modified G2P shader that only calculates particle volume.

The implementation includes a user interface that allows the creation of scenes, modification of parameters, loading and saving of scenes as well as visualization and video recording (Section 7.7).

7.2 Libraries and Technologies

The implementation is written in C++17 with OpenGL 4.6[3] as graphics API. As OpenGL extensions, `NV_depth_buffer_float`[4] was used for exponential shadow maps [36], which is only needed for visualization. The second extension used is `ARB_shading_language_include`[5] in order to support `#include` -directives in GLSL shaders. This could be replaced by using an external preprocessor instead.

The following is a list of all external libraries and code used with a brief explanation what they are used for:

svd_glsl.glsl [6]A GLSL implementation of the minimal branching SVD for 3×3 matrices from [37], written by Alexander Scheid-Rehder. The SVD is needed for many of the supported constitutive models.

glbinding [7] Provides the OpenGL API bindings while leveraging the C++ type system.

GLFW [8] Used to create an OpenGL context as well as a window and provides input handling.

[3]`https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf` Last accessed on 22.04.2020

[4]`https://www.khronos.org/registry/OpenGL/extensions/NV/NV_depth_buffer_float.txt` Last accessed on 22.04.2020

[5]`https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_shading_language_include.txt` Accessed on 22.04.2020

[6]`https://gist.github.com/alexsr/5065f0189a7af13b2f3bc43d22aff62f` Last accessed on 22.04.2020

[7]`https://github.com/cginternals/glbinding` Last accessed on 22.04.2020

[8]`https://www.glfw.org/` Last accessed on 22.04.2020

thinks/poisson-disk-sampling [9] Implements the algorithm from [38]. Poisson disk sampling is one of the methods used for sampling particles from objects in this thesis.

glm [10] Provides the mathematical capabilities of GLSL shaders as C++ implementations, as well as convenience functions for creating transformations and more.

Assimp [11] Used to import 3D models, which can be used as objects in simulations.

stb_image(_write) [12] Used to load image files and save screenshots. Images can be used to define objects in the simulation.

ImGui [13] Used to implement the graphical user interface.

nlohmann/JSON [14] Used to (de-)serialize scenes and MPM parameters as JSON files.

readerwriterqueue [15] A single-consumer, single-producer lock-free queue. It is used in order to write benchmarks to files.

tiny file dialogs [16] Used to add OS dialogs for loading and saving files to the user interface, as well as for displaying error messages.

NvPipe [17] A wrapper around Nvidia NVENC, allowing to record OpenGL framebuffers as videos directly on the GPU. Uses CUDA internally.

spdlog [18] Provides logging capabilities.

7.3 Buffers and Textures

7.3.1 Particle State

Particle properties are spread among 7 *Shader Storage Buffer Objects* (SSBOs) in order to achieve better cache coherency. OpenGL guarantees 16 storage blocks per shader, which means that some of the buffers could still be split further without problems. With the exception of one buffer, the implementation allocates all buffers for all particles, even if some properties

[9] `https://github.com/thinks/poisson-disk-sampling` Last accessed on 22.04.2020
[10] `https://glm.g-truc.net/0.9.9/index.html` Last accessed on 22.04.2020
[11] `https://www.assimp.org/` Last accessed on 22.04.2020
[12] `https://github.com/nothings/stb` Last accessed on 22.04.2020
[13] `https://github.com/ocornut/imgui` Last accessed on 22.04.2020
[14] `https://github.com/nlohmann/json` Last accessed on 22.04.2020
[15] `https://github.com/cameron314/readerwriterqueue` Last accessed on 22.04.2020
[16] `https://sourceforge.net/projects/tinyfiledialogs/` Last accessed on 22.04.2020
[17] `https://github.com/NVIDIA/NvPipe` Last accessed on 22.04.2020
[18] `https://github.com/gabime/spdlog` Last accessed on 22.04.2020

```
layout(std430, binding = 0) buffer PositionMass { vec4[]
    position_mass; };
layout(std430, binding = 1) buffer VelocityVolume { vec4[]
    velocity_volume; };
layout(std430, binding = 2) buffer AffineVelocity { mat3[]
    affine_velocity; };
layout(std430, binding = 3) buffer Determinant { float[]
    determinants; };
layout(std430, binding = 4) buffer DeformationGradient { mat3[]
     deformation_gradient; };
layout(std430, binding = 5) buffer AddInfo { uvec4[]
    additional_info; };
layout(std430, binding = 6) buffer Fractions { float[]
    fractions; };
```

Figure 7: GLSL declarations of the SSBOs containing particle properties. In the respective shaders `readonly` and `writeonly` are added depending on the usage.

are not needed by the respective constitutive model. For example, the water model does not need a deformation gradient. Memory usage could thus be reduced by introducing indexing tables. These tables would however require memory as well and additional buffer read operations would be required at runtime. In this implementation, unnecessary properties are allocated but never read at runtime, which sacrifices memory for efficiency and simplicity.

Figure 7 shows the GLSL storage blocks for the particle SSBOs. All buffers use the `std430` layout. In this layout, a `vec3` is stored exactly like a `vec4` in arrays. For this reason, particle positions are grouped with their mass in a single vector and velocities are grouped with the initial particle volume. Affine velocity and deformation gradient are both 3×3 matrices. They are treated similar to three consecutive `vec3`, which means that the actual memory is padded to 3×4 matrices, which has to be considered during buffer allocation and if data is set on the CPU side. The `Determinant` buffer stores the determinant of the deformation gradient for the water model, the plastic deformation gradient's determinant for the snow model and the hardening state q_p^n for the sand model, while the `DeformationGradient` buffer always stores the elastic deformation gradient, but is not used for water. `Fractions` stores the volume fractions in two-grid simulations. This buffer is not allocated in one-grid simulations.

`AddInfo` is used to store multiple properties of varying data types. The first two components of the `uvec4` store the particle normal at 16-bit floating point precision. The normal is estimated during the G2P step and only used for visualization purposes. Storing the normal is done using the GLSL function `packHalf2x16` and `unpackHalf2x16` retrieves it, where two calls of each

are necessary. The third component stores 8 bit per channel RGB colors in the upper 24 bits, which is also used in visualization. Material indices are stored in the lower 8 bits, which means that 256 different materials are possible in this implementation. If more are needed, the material index would have to be moved to a separate buffer with full 32 bit indices. Retrieval of the individual color channels and the material index is done using bitwise AND as well as bit shifting, which is supported in GLSL. Finally, the last component of the `uvec4` is used to store the logarithm of the plastic deformation gradient's determinant (v_{cp}^n) if the volume corrected sand model is used. It is stored using `floatBitsToUint` and read with `uintBitsToFloat`.

The particles sampled from a single object are stored next to each other. In two-grid simulations, all objects that belong to the first species store their particles in the first part of the buffer, followed by the particles belonging to objects of the second species. Compile time constants are defined in shaders in order to provide starting offsets of each species as well as particle count. As each species has its own shader, no branches are necessary. The respective offset and particle count is provided through generated shader includes.

7.3.2 Materials

Materials are defined using a unified `struct` that contains all parameters of the implemented constitutive models (Fig. 8). These structs are stored as an array in an SSBO. As mentioned in the previous section, this is currently limited to 256 materials because the index is stored as an 8 bit unsigned integer on particles. A global instance of a material is defined in shaders and initialized from the SSBO to the respective particle's material at runtime. This material is then available throughout the functions that implement the constitutive model. The global instance can be modified based on the volume fraction before the constitutive models are evaluated. This only affects the global instance, which is a copy of a material in the SSBO. The modifications are not written back. Thus, the functions that modify materials only use the initial material and the current volume fraction as parameters. In order to implement functions that use the already modified material instead, per-particle materials would be required, which would mean that the material SSBO has one entry per each particle and no material index has to be stored. This was not implemented, but it would require only minor modifications to the implementation.

The implementation allows for either a single constitutive model per grid, or alternatively varying constitutive models across materials. In the first case, the shaders for each species have the model selected at shader compile time via includes and preprocessor directives. In the second case, the constitutive model is stored in the `type` field of each material. This is an integer that is used in a `switch` statement to select the correct functions and data needed for each model. It would be possible to implement this without any branches by

```
struct Material
{
  // Lamé Parameters
  float mu_0; // μ/μ0
  float lambda_0; // λ/λ0

  // Snow plasticity parameters
  float hardening; // ξ
  float theta_c; // θc
  float theta_s; // θs

  // JP (snow model) and JE (water model)
  // are clamped to [det_min, det_max] for stability
  float det_min;
  float det_max;

  // Sand parameters
  // φF, overrides hardening formula if >= 0
  float friction_angle;
  vec4 hardening_vec; // h0, h1, h2, h3
  float cohesion; // cC

  // water parameters
  float k; // bulk modulus
  float gamma; // γ

  int   type; // specifies constitutive model
};

// Global instance that is set from SSBO
// Can be modified based on volume fraction
Material material;

layout(std430, binding = 8) readonly buffer MaterialBlock {
    Material materials[]; };
```

Figure 8: Declaration of the material `struct` and storage block in GLSL, as well as a global material instance that can be modified before it is used in constitutive models.

creating separate shaders for each constitutive model and grouping particles respectively. In this implementation, particles are already grouped by objects without any reordering and each object always has only one material, which means that most particles that are close to each other in the buffers already use the exact same branch. Thus, branch divergence is minimal and it was deemed unnecessary to do this additional step.

7.3.3 Grid

```
layout(std140, binding = 0) uniform GridBlock
{
    ivec3 g_res; // Resolution
    vec3  g_ll; // Lower left corner
    vec3  g_mid; // Midpoint
    vec3  g_ur; // Upper right corner
    vec3  g_half; // g_mid - g_ll
    mat4  g_proj; // Orthographic projection
    float g_h; // Grid spacing
    float g_i_h; // Inverse grid spacing
};

// Transforms from world to grid coordinates
vec3  toGridCoords(vec3 p) { return (p - g_ll) * g_i_h - vec3
    (0.5f); }
// Transforms a grid index to a world coordinate
vec3  indexToPos(ivec3 idx) { return g_ll + g_h * (vec3(0.5) +
    vec3(idx)); }
// Minimum indices affected by a particle at grid coordinates p
ivec3 getMinCoords(vec3 p) { return max(ivec3(ceil(p - vec3(
    kernel_support))), ivec3(0)); }
// Maximum indices affected by a particle at grid coordinates p
ivec3 getMaxCoords(vec3 p) { return min(ivec3(floor(p + vec3(
    kernel_support))), g_res - ivec3(1)); }
```

Figure 9: The uniform block containing the grid information in GLSL, as well as the core functions that transform to and from grid coordinates and calculate the grid indices affected by a particle. The `kernel_support` is provided as a compile time constant.

Spatial grid information is provided in shaders using a *Uniform Buffer Object* (UBO) (Fig. 9). The buffer contains some redundant information in order to avoid additional calculations to derive properties, i.e. both the grid spacing h and $\frac{1}{h}$ are provided. An orthographic projection `g_proj` is provided that transforms particles into grid space such that the corresponding fragments are generated in a way that exactly match the cell centers in the XY planes of the grid if rendered using the same XY resolution. It is calculated using

```
uniform layout(binding = 0, rgba32f) image3D vel_mass_img_1;
uniform layout(binding = 1, rgba32f) image3D vel_mass_img_2;
uniform layout(binding = 4, rgba32f) readonly image3D
    pressure_img;
uniform layout(binding = 0) sampler3D vel_mass_tex_1;
uniform layout(binding = 1) sampler3D vel_mass_tex_2;
uniform layout(binding = 4) sampler3D pressure_tex;
```

Figure 10: The GLSL declarations of the textures representing grid data as both images and samplers. Only the textures that are actually needed are allocated and used in shaders.

glm as

```
g_proj = glm::ortho(g_ll.x, g_ur.x, g_ll.y, g_ur.y, g_ll.z,
    g_ur.z) * view;
```

where `view` performs the necessary z-flip as

$$\texttt{view} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{119}$$

Unlike SSBOs, UBOs cannot use the `std430` layout, so `std140` is used instead. This means that all properties in the grid definition are aligned as `vec4`, including `g_h` and `g_i_h` even though they are only a single `float` each. This has to be considered when filling the UBO in C++, but it can be done easily by defining a struct with the same properties as the UBO in the same order. The matching types are provided by glm and the alignment can be specified in C++ as `alignas(sizeof(glm::vec4))` in front of each property.

The grid is always defined in alignment with the world axis as it simplifies all calculations. In a multi-species simulation both grids are defined equal, which is why only one grid definition is needed.

Velocity and mass of grid nodes are stored in 3D RGBA textures with full floating point precision. Simulations with porous media need two such textures, one for each grid, as well as a third texture that stores the pressure gradient (Eq. (102)) if the buoyancy term is used. While this last texture only needs 3 channels, OpenGL would allocate it as RGBA anyway to get the right alignment. Textures can be created as RGB using `GL_RGB`, however, there is no `rgb32f` layout declaration for images in shaders because they do not exist internally.

Figure 10 shows how the 3D textures are declared as `image3D` and `sampler3D` in GLSL. The images are used for direct write access in shaders using image

```
#ifndef WEIGHT_GLSL // Include Guard
#define WEIGHT_GLSL
#include "/dimension.glsl"
#include "/kernel.glsl"

// Inputs:
//  p: position in grid coordinates
//  coords: grid index
// Result is 1/h · (x_i − x_p)
vec3 dPos(vec3 p, ivec3 coords) { return vec3(coords) - p; }

// Input: return value of dPos
float weight(vec3 d_pos)
{
#ifdef MPM_3D
    return N(d_pos.x) * N(d_pos.y) * N(d_pos.z);
#else
    return N(d_pos.x) * N(d_pos.y);
#endif
}
#endif
```

Figure 11: The shader code that contains the definition of the weighting function. It depends on both the dimension of the simulation as well as the chosen kernel, which are provided through shader includes.

load/store operations. The samplers on the other hand can be used if only read access is required. Usage of the `texelFetch` function allows to use indices instead of texture coordinates and avoids any interpolation. A texture can be bound to an image unit with `glBindImageTexture`, where the unit is simply set the same as the specified binding in the GLSL declaration. This function corresponds to `glBindTextureUnit` for samplers.

In a 2D simulation, the textures are still 3D but they only have a z-resolution of 1. A 3D texture can be attached to a framebuffer the same way as a 2D texture, which means that they can be rendered to using the rasterization pipeline. This does require either binding only one of the 2D layers in the texture to the framebuffer or alternatively binding the full texture and selecting the layer that a primitive is rendered to inside the shader. Rasterization is thus always 2D, there is no built-in support to rasterize as 3D fragments.

7.4 Shader

The implementation makes heavy use of shader includes to reduce code duplication and to put generated code fragments into shaders. This is

also used to set user-defined constants and to add preprocessor defines for conditional code compilation, reducing the need for branches at runtime as well as the overall size of the compiled shader programs.

Shader includes are possible using the `ARB_shading_language_include` extension, or alternatively using an external preprocessor. Only the former was implemented. It has to be enabled in each shader before it can be used via

```
#extension GL_ARB_shading_language_include : enable
```

which then allows to include code as `#include "/named-string"`. Includes are provided in C++ by calling `glNamedStringARB` with the name of the include as well as the shader code, which could come from a file or be generated. Names have to start with `/`.

An example of this technique can be seen in Fig. 11. This code is available through includes as `/weight.glsl`, and it uses other includes in its implementation. `/dimension.glsl` contains defines that vary depending on whether the simulation is 2D or 3D, which is used here to vary the definition of the weighting function. That function itself relies on `kernel.glsl`, which contains either the cubic B-spline definition (Fig. 12) or the quadratic one (Fig. 13). Both of these kernels include further definitions needed throughout shaders.

7.4.1 P2G Shader

The P2G transfer is implemented using a scattering approach based on the rasterization pipeline. In 2D, this can be done by rendering particles as points with a size that is dictated by the kernel support. Accumulation happens by using additive blending. This automatically takes care of write-conflicts. No depth buffer is needed. The corresponding OpenGL functions that set up this state are

```
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ONE);
glDisable(GL_DEPTH_TEST);
glPointSize(kernel_point_size);
```

where `kernel_point_size` is 3 for the quadratic B-spline and 4 for the cubic one. Particle positions are already available in a SSBO, which is why no *Vertex Buffer Object* (VBO) is generated. Instead, only a *Vertex Array Object* (VAO) is generated without any additional setup, as a VAO has to be bound to perform any draw calls. Rendering is initiated as

```
glDrawArrays(GL_POINTS, 0, num_particles_species_a);
```

```
#ifndef KERNEL_GLSL
#define KERNEL_GLSL

float N(float x)
{
  x = abs(x);
  return x >= 0 ?
    (x < 2 ?
      (x < 1 ?
       (0.5f * x * x * x - x * x + (2.f / 3.f)) :
       (-(1.f / 6.f) * x * x * x + x * x - 2 * x + (4.f / 3.f))
      ) :
      0.f
    ) :
    0.f;
}

// Definitions used for geometry shader
#define MAX_VERTICES 5
#define LAYER_DIFF 2

const float kernel_support = 2.f;
// M_p^{-1} = 3/h^2 = apic_factor * g_i_h * g_i_h
const float apic_factor    = 3.f;

#endif
```

Figure 12: The one-dimensional cubic B-spline kernel, as well as further definitions related to its support and its usage in APIC MLS-MPM.

for the first particle species. This happens while a *framebuffer object* (FBO) is bound that has the grid texture attached as color attachment. The second species is rendered while a different FBO is bound that has the second grid texture attached to it. A second shader program is used for this. Rendering is initiated as

```
glDrawArrays(GL_POINTS, num_particles_species_a,
    num_particles_species_b);
```

which means that the specified offset affects `gl_VertexID` in this call, so that particle properties can be accessed in both vertex shaders by accessing the SSBOs with `gl_VertexID` as index.

If the buoyancy term is used, a third grid texture exists that stores the pressure gradient. This texture is bound as the second color attachment to the FBO for the species that represents water. The respective shader program will have `MPM_PRESSURE` defined through an include, which triggers

```
#ifndef KERNEL_GLSL
#define KERNEL_GLSL

float N(float x)
{
  x = abs(x);
  return x >= 0 ?
    (x <= 1.5f ?
      (x <= 0.5f ?
        (-x * x + 0.75f) :
        (0.5f * x * x - 1.5f * x + 1.125f)
      ) :
      0.f
    ) :
    0.f;
}

// Definitions used for geometry shader
#define MAX_VERTICES 3
#define LAYER_DIFF 1

const float kernel_support = 1.5f;
// M_p^{-1} = 4/h^2 = apic_factor * g_i_h * g_i_h
const float apic_factor    = 4.f;

#endif
```

Figure 13: The one-dimensional quadratic B-spline kernel, as well as further definitions related to its support and its usage in APIC MLS-MPM.

the additional calculation of Eq. (118). This is also accumulated via additive blending by means of a second output variable in the fragment shader.

Most of the work happens in the vertex shader, which calculates the matrix $\boldsymbol{Q}_p$ from Eq. (116) depending on the constitutive model and the material parameters, which are potentially modified based on the volume fraction. This matrix is passed on as a vertex shader output along with particle mass, velocity, initial volume, the position in grid coordinates as well as the pressure if it is needed for the buoyancy term. All of these outputs are declared as `flat` because no interpolation happens. This means they are copied as-is to the generated fragments based on the point size. Thus, all calculations that relate to the actual grid nodes happen in the fragment shader. The vertex shader sets `gl_Position` by transforming particle positions with the orthographic projection `g_proj` that matches the grid. Thus, fragments are generated exactly at grid centers.

The fragment shader calculates weights as well as $\boldsymbol{x}_{\boldsymbol{i}}^n - \boldsymbol{x}_p^n$ using the functions from Fig. 11 and outputs velocity weighted by mass according to

the inner part of the sum in Eq. (117). Weighted mass (Eq. (38)) is written to the fourth component of the output vector. The sums in the formulae are then automatically evaluated due to additive blending. For the buoyancy term, the fragment shader of the fluid species additionally outputs the pressure gradient. Normalization of velocity by mass happens in a later step, as this requires the grid mass to be fully accumulated first.

The approach is extended to 3D by means of a geometry shader (Fig. 14). This shader duplicates each particle for every affected grid cell along the z-dimension and then sets `gl_Layer` accordingly. The generated fragments will then write into the 2D layer of the bound 3D texture at the z-index corresponding to `gl_Layer`. This variable is also available inside the fragment shader and is used to identify the actual node that is being written to, as this affects all calculations. The geometry shader additionally has to pass all of the vertex shader outputs through to the fragment shader for each copy of the original vertex.

7.4.2 Grid Shader

The grid shader implementation is a simple compute shader that executes one thread for each grid node. In a multi-grid simulation, the same thread handles the nodes with same index in both grids.

Execution happens with a work group that matches the dimension of the simulation. In 3D, the work group size is $(8, 8, 8)$, resulting in 512 threads per work group. 2D simulations use a work group size of $(16, 16, 1)$, meaning 256 threads per group. This scheme ensures spatial coherence during execution which can help with branch divergence as well as cache usage.

Each thread is responsible for the nodes with index `gl_GlobalInvocationID`. Due to the nature of work groups, this index can exceed the dimensions of the grid texture, in which case the thread simply returns.

In single grid simulations the thread first reads the value of the grid at its assigned index using `imageLoad`. If the mass, which is contained in the fourth component, is zero, no work has to be done and the thread returns. Otherwise, the velocity is first normalized by mass. Afterwards, collision is applied as described in Section 4 Step 5. Level sets as well as friction coefficients and sticky flags are provided through shader includes. Both friction as well sticky flag can vary spatially. The level sets are build by concatenation of primitive signed distance fields that can additionally be animated based on time. Lastly, the normalized and collided velocity is written back using `imageStore`.

Multi-grid simulations first check the masses of both grids. If only one of them is zero, the algorithm proceeds with the remaining one as described in the previous paragraph. If both are zero, the thread returns. Otherwise, both grid velocities have to be normalized and the momentum exchange is applied using drag term (Eq. (103)) and optionally pressure term (Eq. (118)) which can be read from the third grid texture. The drag term can optionally

```
layout(points) in;
layout(points, max_vertices = MAX_VERTICES) out;

// Position is already in grid coordinates
layout(location = 0) flat in vec4 in_pos_mass[];
layout(location = 1) flat in vec4 in_vel_vol[];
layout(location = 2) flat in mat3 in_Q_p[];
#if defined(MPM_PRESSURE)
layout(location = 6) flat in float in_pressure[];
#endif

layout(location = 0) flat out vec4 pos_mass;
layout(location = 1) flat out vec4 vel_vol;
layout(location = 2) flat out mat3 Q_p;
#if defined(MPM_PRESSURE)
layout(location = 6) flat out float pressure;
#endif

void main()
{
    const int p_z = int(in_pos_mass[0].z + 0.5);
    const int min_coords = max(int(p_z - LAYER_DIFF), 0);
    const int max_coords = min(int(p_z + LAYER_DIFF), g_res.z -
        1);

    for(int i = min_coords; i <= max_coords; ++i)
    {
        pos_mass = in_pos_mass[0];
        vel_vol  = in_vel_vol[0];
        Q_p    = in_Q_p[0];
#if defined(MPM_PRESSURE)
        pressure = in_pressure[0];
#endif
        gl_Position  = gl_in[0].gl_Position;
        gl_Layer     = i;
        EmitVertex();
        EndPrimitive();
    }
}
```

Figure 14: The geometry shader that copies particles onto the affected layers. `LAYER_DIFF` and `MAX_VERTICES` depend on the kernel support (see Figs. 12 and 13). `MPM_PRESSURE` is only defined if the buoyancy term is used and only in the geometry shader corresponding to the fluid species.

be clamped according to Eq. (109) or fixed to be constant based on the same derivation, which adds the possibility of reverting to collision behavior instead of porous interaction. Collision is applied after the momentum exchange.

7.4.3 G2P Shader

G2P transfer is also implemented with a compute shader. It uses a one-dimensional work group size of $(512, 1, 1)$, executing one thread for each particle. Similar to the P2G shader, there are two distinct shaders, one for the first species and another one for the second species if it exists.

Each shader first checks whether `gl_GlobalInvocationID.x` is larger or equal to `PARTICLE_COUNT`, which is a constant defined based on the number of particles in the respective species. This is used to check for overflow, in which case the thread returns. The actual particle index is retrieved by adding `PARTICLE_OFFSET`, a constant which is 0 for the first species and equivalent to the number of particles in it for the second species, as this is the index where those particles start.

The shader continues by accumulating grid properties with a gathering approach. Using the functions `getMinCoords` and `getMaxCoords` from Fig. 9, the grid nodes that affect the particle are determined and subsequently iterated. This accumulates velocity (Eq. (64)), affine velocity matrix (Eq. (69)) as well as volume fraction (Eq. (107)) in two-grid simulations. For visualization purposes, surface normals are additionally estimated based on the normalized mass gradient, which is discretized as

$$\hat{\boldsymbol{n}}_p^{n+1} = \sum_{\boldsymbol{i}} \omega_{\boldsymbol{i}p}^n m_{\boldsymbol{i}}^n (\boldsymbol{x}_{\boldsymbol{i}} - \boldsymbol{x}_p^n), \quad \boldsymbol{n}_p^{n+1} = -\frac{\hat{\boldsymbol{n}}_p^{n+1}}{\|\hat{\boldsymbol{n}}_p^{n+1}\|}. \tag{120}$$

M_p^{-1} is left out here as it is constant and thus irrelevant due to normalization.

After estimating the particle properties using the grid, the shader continues by loading the particle material and modifying it given the volume fraction as well as the material interaction functions assigned to this species, just like in the P2G shader. If the material is water, the determinant is evolved (Eq. (83), where $\nabla \boldsymbol{v}_p^{n+1} = \boldsymbol{C}_p^{n+1}$ due to MLS-MPM). Otherwise the deformation gradient is evolved according to Eq. (113) and subsequently treated for plasticity depending on the constitutive model, which additionally updates hardening state and determinants of the plastic deformation gradient.

If desired, particle velocities are subsequently changed based on the same collision algorithm used in the grid shader as explained in Section 4 Step 9. Lastly, particles are advected using explicit time stepping (Eq. (50)).

7.5 Particle Initialization

Particles are initialized by sampling the objects that they are supposed to represent. The sampled positions are generated in C++ using the CPU and

subsequently uploaded to the GPU buffer. Each particle is initially assigned the same mass based on the volume and density of the object:

$$m_p = \frac{\rho_{\text{obj}} V_{\text{obj}}}{\#\text{particles}}. \tag{121}$$

which is uploaded alongside positions as the fourth component of the vector.

After positions and mass for every object are uploaded to the GPU, a P2G transfer is executed, followed by a modified G2P shader. The P2G transfer is only necessary in order to accumulate grid masses, which are subsequently used in the G2P shader to estimate the initial volume V_p^0 for each particle according to Section 4 Step 2. Just like the regular G2P shader it also calculates the initial volume fraction and particle normals for visualization. The volume estimation can alternatively be skipped, in which case the volume is set as

$$V_p^0 = \frac{m_p}{\rho_{\text{obj}}} = \frac{V_{\text{obj}}}{\#\text{particles}}. \tag{122}$$

In this case, the volume is constant. Variations in volume mostly occur at borders or if the sampling is not globally uniform, which means that the difference between proper initialization and constant initialization is relatively small.

After the volume is initialized, the G2P shader continues with the remaining particle properties. The initialization code is built based on the objects and provided through shader includes. Each particle determines which object it belongs to based on its index and then executes the initialization code that was generated for this object.

The properties that are initialized this way are:

- The contents of the fourth SSBO (see Fig. 7), i.e. J_{Ep} for water, J_{Pp} for snow and hardening state q_p^n for sand.
- particle velocity
- material index
- particle mass
- color

Velocity, determinant and mass can either be constant across all particles of an object, or alternatively initialized with a user-defined GLSL expression that spatially varies the assigned value. Material index is the same for every particle of an object. Color is assigned by first generating a pseudorandom number $\xi_c \in [0, 1)$, which is then used to select from a set of user-defined colors with assigned probabilities. This can for example be used for sand which is often made up of grains of varying color.

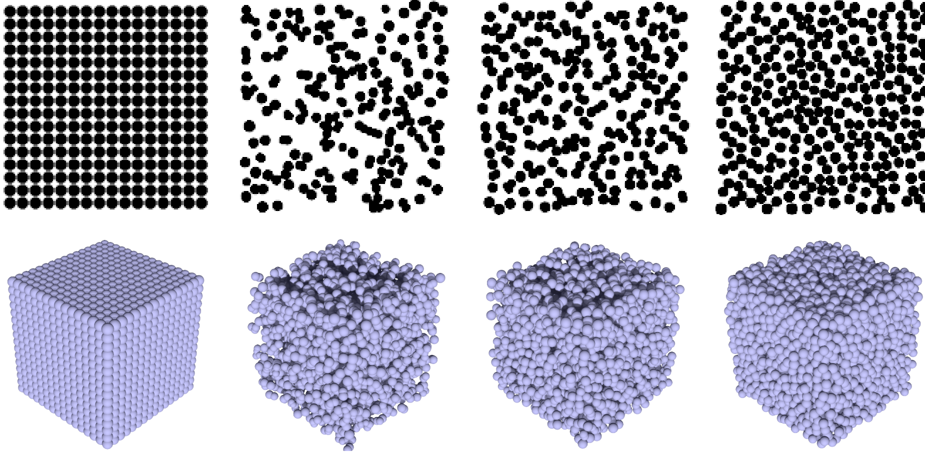


Figure 15: The four different sampling techniques implemented in this thesis. From left to right: grid, random, stratified and poisson disk sampling. The 2D examples in the top row are 256 particles each, except for the poisson disk one which is 280 particles. For 3D the numbers are 4096 and 4297. Sample density is set artificially low for visualization purposes.

If the mass is changed in this step, the volume that was determined beforehand is not correct anymore. Thus, another P2G transfer followed by a new volume calculation using the modified G2P shader are executed after this step.

All remaining properties are initialized using their default values. Those are $\boldsymbol{C}_p^0 = \boldsymbol{0}$, $\boldsymbol{F}_{Ep}^0 = \boldsymbol{I}$ and $v_{cp}^0 = 0$.

7.5.1 Sampling Strategies

Particles are always first sampled from the whole axis-aligned bounding box (AABB) of the object. They are subsequently filtered depending on the actual object so that all particles that are not part of it are removed.

Sampling density is controlled by the particle spacing parameter P_s, which is defined as a portion of the grid spacing h. This means that the actual spacing is $P_s \cdot h$. The particle spacing translates to the number of particles per grid cell as

$$\#\text{PpC} = \frac{1}{P_s^d}, \tag{123}$$

where d is the dimension. For example, a particle spacing of 0.5 means that 4 particles are spawned per cell on average in 2D and 8 in 3D. This is also the default value, as it worked well in most cases.

Four different sampling strategies are implemented (see Fig. 15):

Grid sampling. Particles are placed as cell-centers of a grid with spacing $P_s \cdot h$.

Random sampling. The same number of particles that would be generated via grid sampling is instead drawn from a uniform random distribution.

Stratified sampling. Like grid sampling, but the position inside each cell is drawn from a uniform random distribution.

Poisson disk sampling. Samples a random set of particles that additionally fulfills the property that no two particles are closer than a specified minimum distance. This distance is set to $\frac{P_s \cdot h}{1.7^{1/d}}$. The value was chosen empirically as it generated similar amounts of particles as the other methods with `thinks::poisson_disk_sampling` and a maximum number of sample attempts of 30.

Poisson disk sampling is the best sampling strategy out of the four, as it generates globally uniform spaced sets of particles. Grid sampling has similar properties but the regular pattern can lead to artifacts in the simulation. Accurate simulations with random sampling would require higher particle numbers in order to avoid some areas ending up under-sampled. The typical clusters of higher and lower density in random sampling may however produce visually interesting results in some circumstances. Lastly, stratified sampling is the closest approach to poisson disk sampling of the four. While it does not guarantee a minimum distance between sample points, it does guarantee a maximum distance as well as an upper bound on how many particles can clump together because each particle is constrained to positions inside its assigned cell. Stratified sampling is roughly as fast as grid sampling and random sampling, which is in the order of milliseconds. In comparison, poisson disk sampling can take seconds or minutes depending on particle counts.

7.5.2 Object Types

The primitive object types are cuboids (rectangles in 2D) and spheres (circles), the implementation of which is straightforward. In both cases, total volume can be calculated directly. More complex shapes can be created using either meshes or bitmaps. In these cases the total volume is estimated using the fraction of initial to remaining sample positions:

$$V_{\text{obj}} = V_{\text{AABB}} \cdot \frac{\#\text{points_remaining}}{\#\text{points_initial}}. \tag{124}$$

Once the actual sample positions that represent the object have been determined, the object is positioned in space by rotating and translating the generated particles.

Meshes. The AABB of a mesh can be directly calculated by determining the minimal and maximal coordinates of the vertices on each axis. To simplify

positioning of objects, the mesh is then translated such that the midpoint of its AABB is the origin. Given the particles spawned in the AABB, all particles that are outside of the mesh have to be removed. This could for example be implemented using raytracing or by determining the signed distances to the closest point on the mesh using the angle weighted pseudonormal [39]. Instead of these, a fast approximate solution was chosen in this thesis.

The inputs for the algorithm are a set of directions, the mesh and a buffer containing the sampled positions. These are stored as 4-component vectors, where the fourth component is initially set to 1. For each direction, the following steps are repeated:

1. Create an orthographic projection from the given direction. This projection is used during rasterization in the subsequent steps.
2. Enable depth testing with `glEnable(GL_DEPTH_TEST)` and writing to the depth buffer with `glDepthMask(GL_TRUE)`.
3. Clear the depth buffer and rasterize the mesh into a FBO with only a depth buffer attached.
4. Set the depth buffer to read only via `glDepthMask(GL_FALSE)`.
5. Rasterize the sampled positions as points of size 1. The fragment shader has early depth test enabled via `layout(early_fragment_tests)in;`. This means that the body of the shader does not execute for points that are behind the mesh from this direction. If the fragment shader does execute, it sets the fourth component of its respective sample position to 0.

The set of directions used for all examples in this thesis are the three main axes as well as their respective negatives. Once the algorithm is done for each direction, the sample position buffer is downloaded from GPU memory. All positions that have the fourth component set to 0 are discarded.

It is obvious that this algorithm cannot handle hollow objects as well as certain types of cavities and holes. However it is still sufficient for many objects, such as the *Stanford Bunny* (if its holes are filled) or the *Armadillo*[19]. More complex objects could be supported by extending this approximate algorithm with depth peeling [40] and then checking intervals of depth values to determine whether a position is inside or outside.

Bitmaps. A grayscale image alongside a user-defined threshold can be used as a mask to define which particles are part of the object. This is done by mapping the coordinates of the sampled positions to texture coordinates. Using these, the bitmap is sampled with either nearest neighbor or bilinear

[19] `http://graphics.stanford.edu/data/3Dscanrep/` Last Accessed on 27.04.2020

interpolation. If the value exceeds the threshold, the position is discarded. Alternatively, the criterion can be to discard if the value is lower than the threshold. A second image (which can be the same one) can optionally be sampled to initialize the colors of each particle.

The implementation is straightforward in 2D. 3D objects can use the exact same procedure by mapping only the x and y coordinates to texture coordinates. In this case, the object defined by the mask is simply elongated in the z-direction. Alternatively, texture coordinates can be mapped in a *solid of revolution* scheme. The distance of a given position to the midpoint of the AABB in the XZ plane is then mapped to the x-coordinates of the texture and the y-coordinates of each position are mapped to the y-coordinates of the texture as before.

7.6 Visualization

Because all data is already available on the GPU, visualization with OpenGL can be easily done during simulation. The user can visualize grid textures and particles.

7.6.1 Grid Visualization

The grid is visualized by rendering a screen filling triangle that is textured with the user-selected layer of the 3D texture, which is either one of the two velocity/mass textures or the pressure gradient texture.

Mass and velocity are visualized separately. As all properties are full floating point values, the user selects the beginning and end values of a range that is then mapped to displayable colors. For velocity and pressure gradient it is also possible to display the per-component absolute values or the vector length.

Collision objects can optionally be visualized as well. This is done by evaluating the level-set at each pixel. If it is less than or equal to zero, the pixel is drawn with a user-selected color instead of the contents of the texture.

7.6.2 Particle Visualization

Each spawned object can be assigned its own shader and settings, which will then be used to render all particles of that given object. Objects are rendered one after the other in the order that the user specifies.

Geometry types. Particles can be rendered as points of a specified size in pixels, spheres or icosahedra (see Fig. 16). For the latter two, size is calculated based on particle volume by assuming each particle to be a sphere (or circle in 2D) in both cases. While spheres look the same no matter how they are rotated, icosahedra do not. The rotation matrix is estimated from the deformation gradient by computing the polar decomposition.

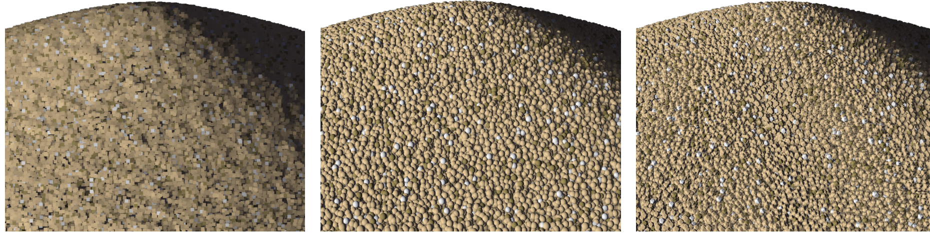


Figure 16: A pile of sand rendered as points with estimated normals (left), spheres (middle) and icosahedra (right).

If points are rendered, there is no actual 3D geometry that can be shaded. If shading is desired, each point is assigned the estimated normal of its respective particle. This successfully captures the shape of the object without having to render proper geometry.

Spheres are rendered as quads instead of actual sphere geometry. In the vertex shader, each quad is aligned to face the camera and its position is set to the front-most point of the sphere that shall be rendered. The quad's sides are the length of the sphere diameter. This setup ensures that the generated fragments cover the area that would have been generated for a real sphere. The fragment shader then resolves the actual sphere with raytracing. This yields the actual intersection point of the camera ray and the sphere. If the ray misses the sphere, the fragment is discarded. Otherwise the normal is calculated using the sphere center and intersection point and `gl_FragDepth` is reassigned based on the actual depth of the fragment, which ensures proper depth tests.

Collision objects. If the user wants to visualize collision objects, they are rendered using sphere tracing [41] because they are defined with signed distance fields. A problem with this is that the camera is usually positioned inside a collision object. This happens because all scenes occur in a bounded volume which means that there is always a collision object surrounding the whole volume in order to prevent particles from escaping the simulation. To solve this, any negative values during sphere tracing are treated as if they were positive until the first positive value is encountered. After this, sphere tracing continues as usual.

Shadows. In 3D visualizations, shadows are an important part for depth and shape perception. A single directional light source is used in this implementation. Shadows are implemented with exponential shadow maps [36]. The implementation is kept simple and does not contain the more advanced failure classification with fallback described in the original paper. Filtering of the shadow maps is done by generating a few layers of mipmaps. A lower

resolution level of the mipmap is then used to resolve shadows, which yields slightly soft shadows.

It is possible to independently decide whether an object casts shadows, as well as whether it receives shadows. The geometry used during rendering of the shadow map can also be either points, spheres or icosahedra and can be selected independently of the geometry used for actual rendering.

If desired, the collision objects can also cast shadows. This is implemented by sphere tracing from the light source the same way as is done in order to display the collision objects themselves.

Colors. Each object can be rendered using either the assigned particle colors or alternatively with colors based on particle properties. The properties that can be visualized are:

- particle mass m_p
- particle velocity $\boldsymbol{v}_p^n$
- particle volume V_p^0
- particle determinant/hardening state $J_{Ep}^n / J_{Pp}^n / q_p^n$
- logarithm of particle determinant v_{cp}^n

Properties can be mapped to colors the same way as is possible when visualizing grid properties.

Shading. All objects are rendered with diffuse shading. Every object can additionally be assigned an opacity. Transparency is implemented on a per-object basis with two passes:

1. Enable depth testing and blending.
2. Set the depth test to `glDepthFunc(GL_LESS)` and the blending to `glBlendFuncSeparate(GL_ZERO, GL_ONE, GL_ZERO, GL_ONE)` which will prevent any writes to the color buffer.
3. Draw all particles with the selected geometry type.
4. Change the depth test to `glDepthFunc(GL_LEQUAL)` and the blending to
```
glBlendFuncSeparate(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA,
                    GL_SRC_ALPHA_SATURATE, GL_ONE).
```
5. Draw all particles again. Only the front-most fragments will write to the color buffer and be blended with whatever was rendered before this object.

This algorithm is not order-independent and not designed with multiple transparent objects in mind. It works best when displaying solid objects alongside a transparent fluid in order to allow to see inside. It is possible to change the order in which the objects are drawn to achieve the desired results, but the order is still only on a per-object basis, meaning that it is not always possible to produce correct results.

7.7 User Interface

A user interface is provided to create and configure scenes, select visualization options, start and pause simulations and benchmarks as well as record videos and screenshots. The functionality is organized into multiple tabs.

Scene tab. The scene tab is used to save, load and create scenes. This includes:

- selection of collision scene
- Defining friction and sticky flags for collisions, either as constant values or using a GLSL expression for spatially varying values. These can be set separately for each species (and thus grid).
- Setting the gravity, which can also vary spatially if a GLSL expression is used.
- Adding, deleting, duplicating and reordering objects, which includes selecting the properties of the object:
  - type of object (cuboid, sphere, mesh or bitmap) and their respective settings
  - size, position and rotation of the object (using euler angles or angle axis representations)
  - material index
  - the species in multi-grid simulations
  - the shader that is used to render this object
  - density
  - whether to properly initialize volume or set it as constant
  - whether to use constant mass or spatially varying mass by specifying a GLSL expression
  - the sampling strategy, including random seed used during sampling
  - the sample spacing
  - initial velocity as either a constant or GLSL expression
  - initial determinant as either a constant or GLSL expression
  - a list of colors with their respective probabilities

Shader tab. This tab contains all settings related to visualization:

- background color and opacity
- selection of render mode as either grid visualization or particle visualization
- whether to render collision objects before or after grid and particles or not render them at all
- color of the collision objects
- whether to use shadow mapping as well as the resolution of the shadow map
- light direction and color as well as ambient light color
- grid visualization settings, i.e. which property and range of values shall be visualized
- adding, deleting and duplicating shaders, including their respective settings

Camera tab. The camera tab can be used to reset the view and change camera movement sensitivity and speed. It is also possible to save the current view to a list so that the same view can later be restored.

Simulation tab. This tab is used to configure the simulation. The simulation can be paused and reset here, where resetting applies newly configured settings. It is possible to limit how many simulation steps are allowed to happen in one frame, which can be used to increase responsiveness of the application. Further settings include:

- timestep
- grid resolution and spacing
- whether to additionally apply collision to particles
- whether to use the quadratic or cubic B-splines
- whether to use a single or multiple constitutive models per grid
- adding, deleting and duplicating materials along with specifying their settings and selecting the constitutive model
- whether to perform a single or multi-grid simulation
- the drag coefficient and whether to clamp it or always set it to the limit

- whether to use the buoyancy term and if so, which species is treated as the fluid that contributes its pressure to the term
- GLSL code that defines the material interaction of each species by modifying the material properties based on the volume fraction

Benchmark tab. This tab is used to measure how long the P2G, grid and G2P shaders take. The measurements can be written to file and it is also possible to select a number of scenes and then start a batch benchmark for all of them.

Recording tab. The recording tab is used to save a screenshot or record the simulation to a video. Videos can be recorded using either the h264 or HEVC (also known as h265) codecs because these are the codecs that NVENC supports. The user can choose bitrate and how many frames are recorded per second, at what time to start and stop recording and what the playback speed of the resulting video should be. It is also possible to record videos at higher or lower resolutions than the window.

Part IV
Evaluation

8 Sampling Methods

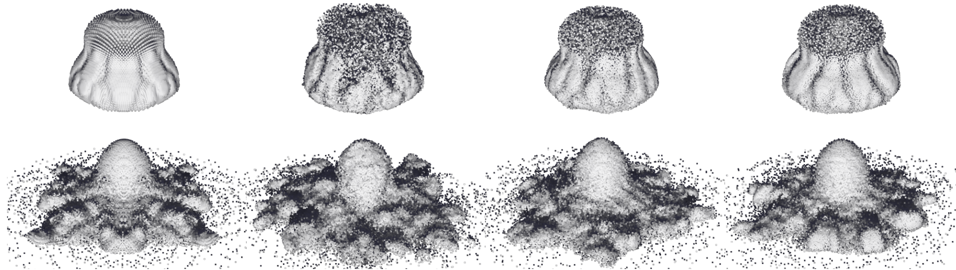


Figure 17: Two frames of a snow sphere dropping on the floor and shattering. From left to right the sphere was sampled using grid, random, stratified and poisson disk sampling.

The impact that the initial particle configuration can have on a simulation is shown in Fig. 17. In all four cases a sphere of the same size is simulated as it falls on the ground. The material is set to the base snow material from [1], namely $E = 140000$, $\nu = 0.2$, $\xi = 10$, $\theta_c = 0.025$, $\theta_s = 0.0075$ and a density of 400. Grid sampling results in a very unnatural perfect symmetry of the four quadrants in the XZ-plane. Random sampling on the other hand produces a very unpredictable and noisy result, which is partially alleviated by stratified sampling. Poisson disk sampling however clearly shows the smoothest result and behaves uniformly without introducing unnatural symmetry or patterns. It also produces more clearly pronounced chunky fracturing which is desirable in snow simulation.

While the visual differences of the sampling techniques are immediately apparent, the problems can reach much further and result in severe numerical errors. To illustrate this, a purely hyperelastic sphere is simulated with a single particle per cell. This is typically insufficient for complex materials such as sand or snow, but can be enough for hyperelasticity as can be seen in Fig. 18. Poisson disk sampling leads to no issues in this simulation. Grid sampling on the other hand suffers from severe numerical fracture, resulting in the sphere being shattered to pieces even though no fracturing was desired. Random sampling manages to keep the sphere intact, but large cracks form on the outside. Lastly, stratified sampling only shows very minor numerical fracturing on the outside that is mostly only visible in motion.

Figure 18: Numerical fracture on a hyperelastic (Neo Hookean) sphere sampled at one particle per cell. From left to right: grid, random, stratified and poisson disk sampling.

The tests clearly show that grid sampling and random sampling are unsuitable for simulations. Poisson disk sampling has the best properties across the board, however, stratified sampling is a close contender and might be preferable in some circumstances such as test-runs as samples are generated many times faster with it.

9 Basis Functions

In order to illustrate the possible benefits of cubic B-splines over quadratic ones, the simulation from Fig. 18 is run again with grid sampling. While a sample rate of one particle per cell still resulted in numerical fracture for both basis functions, a particle spacing of 0.9 (1.37 particles per cell) changes this. The result is shown in Fig. 19. The small increase in sample rate was insufficient in the quadratic case, while the cubic B-spline fully removes any numerical fracture in this situation. This shows that cubic B-splines can handle larger deformations, which is a direct result of their larger support.

Cubic B-splines can also have large benefits if the sample rate is sufficiently high. This is shown in Fig. 20 using a fluid simulation. The constitutive model for fluids (Section 5.1.5) suffers a large volume loss of 38% with the quadratic B-spline. This is reduced to only 8% using cubic B-splines. Note that these numbers are specific to this example, as they depend on many factors such as grid spacing, density, time step, constitutive model and material parameters, meaning that the gap can be much smaller in other circumstances. For example, the volume loss changed to only 22% using the volume corrected

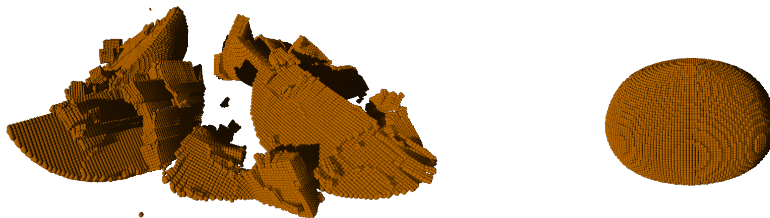


Figure 19: The sphere from Fig. 18 with grid sampling using a spacing of 0.9. The left simulation used quadratic B-splines while the right one used the cubic ones.

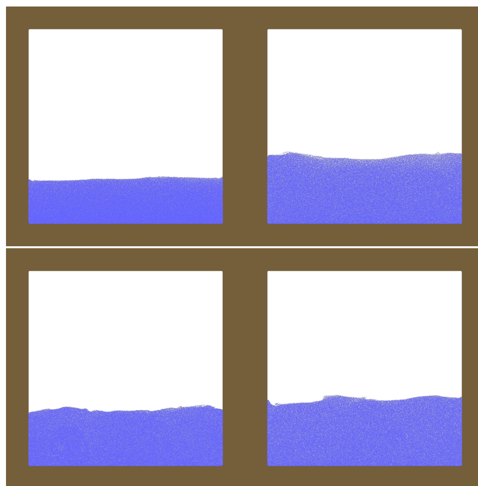


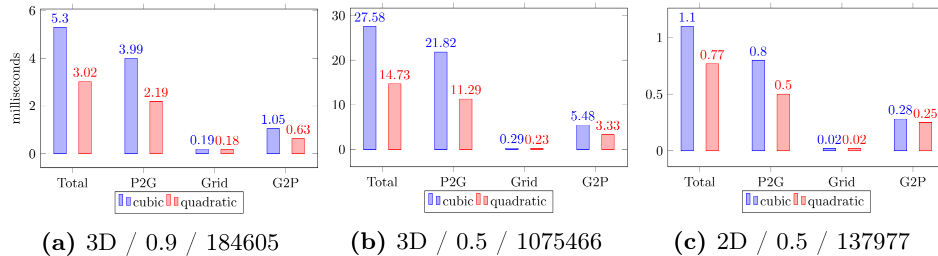
Figure 20: Volume loss in a fluid using quadratic (left) and cubic (right) B-splines. The top row uses the constitutive model for fluids, resulting in 38% and 8% volume loss. The volume corrected sand model (bottom row) shows a smaller gap with 22% and 9% if used to model a fluid.

Figure 21: The snow simulation from Fig. 17 with quadratic (left) and cubic (right) B-splines. The latter behaves less noisy and energetic.

sand constitutive model (Section 5.2.2) in the quadratic case, while increasing slightly to 9% for the cubic one.

Lastly, Fig. 21 shows the impact of using the cubic basis function in a snow simulation. The result is much smoother and appears to be less energetic and noisy. While it could be argued that a less energetic result means worse energy conservation, the increased spray radius in the quadratic case is likely a numerical artifact and related to the aforementioned volume artifacts. In this case however the problem is volume gain, resulting in less dense snow that sprays further. Volume is gained when critical stretch is exceeded. This is attributed to J_P, which can grow indefinitely. More advanced constitutive models thus attempt to ensure that $J_P = 1$, preventing volume gain due to plasticity [9]. Volume gain additionally results in less particles per cell which in turn decreases the accuracy of the simulation.

The benefits of cubic B-splines come at the cost of performance. Figure 22a shows time measurements of the scene from Fig. 19 with poisson disk sampling. These benchmarks were executed on a laptop with a *GTX 1060 6GB* mobile



(a) 3D / 0.9 / 184605 **(b)** 3D / 0.5 / 1075466 **(c)** 2D / 0.5 / 137977

Figure 22: Time measurements of a single simulation step of the scene from Fig. 18 with poisson disk sampling and both basis functions. The plots use a different dimension and particle spacing and thus different numbers of particles. These three quantities are listed in the respective subcaption.

GPU and an *i7–8750H* CPU[20]. Cubic B-splines result in a 1.75× longer simulation. This affects the P2G and G2P steps, while the grid shader is unaffected. The grid shader is mostly irrelevant to performance as it takes only a very small portion of the total time. Computation time thus scales linearly with the number of particles. Figure 22b illustrates this with a reduced particle spacing. In this case, cubic B-splines took 1.87× as long. Based on the basis functions support, computation time would be estimated to be $\frac{4^3}{3^3} \approx 2.37\times$ longer. This is much larger than the actual values. In 2D, this would instead be $\frac{4^2}{3^2} \approx 1.78$, which is also larger than the experimental result of 1.43 (Fig. 22c). However, computation times of 2D simulations compared to 3D do adhere to the predicted theoretical values of three and four times longer simulation times with the same number of particles using quadratic and cubic B-splines, respectively.

10 Single Constitutive Model

This section explores the possibilities of simulations that use a single constitutive model on a single grid. The first step to material variety is to vary particle properties such as mass or determinant during initialization. This is very useful with the snow constitutive model and was also employed in [1] to achieve more realistic behavior. Figure 23 illustrates this with snowballs that are thrown against a wall. Uniform initialization of particle properties results in a fine spray of particles, while most of the snowball simply sticks to the wall. Chunky fracture is achieved by initializing portions of the snow to be more dense and stiff. This can produce drastically different results based on how these dense areas are distributed.

Most constitutive models can be used to model fluids. Figure 24 compares fluid behavior using the volume corrected sand model, the snow model and

[20]Due to the coronavirus pandemic access to the lab with more powerful hardware was restricted.

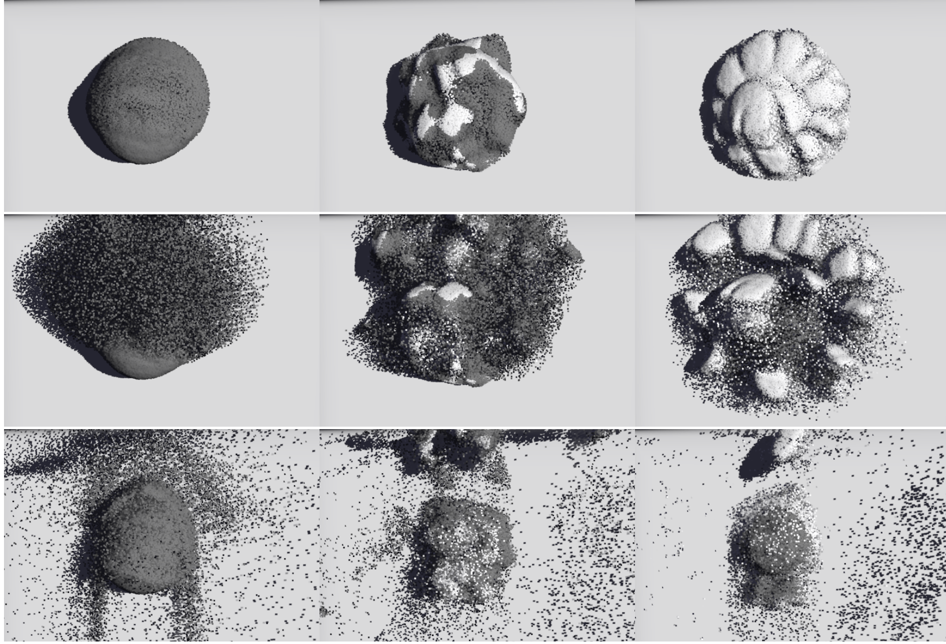


Figure 23: Snowballs are shattered on a wall with sticky collision. Each snowball is initialized differently. Darker particles use $J_P = 1$, while bright particles have 1.5 times the mass and $J_P = \frac{2}{3}$. They are spatially varied with a noise texture (second column) and with a dense outer shell (last column).

the purely hyperelastic Neo Hookean model to the model that is specifically created for weakly compressible fluids. The sand model manages to reproduce the thin fluid sprays, but fluid that rushes up along the collision surfaces looks very jagged and behavior during free-fall is different. Free-fall using the snow model is also different, however, the fluid along the walls behaves more smoothly compared to the sand model. The fluid spray is however much thicker. Lastly, the Neo Hookean model behaves closest to the fluid model at $\mu = 0$. This is however not shown in Fig. 24 because it quickly turned unstable and subsequently exploded. Using small values for μ stabilizes the results, but causes chunky fracturing.

Apart from the ability to model fluids, the constitutive models for sand and snow are able to represent a wide range of material behavior (Fig. 25). Both can be reverted to behave purely hyperelastic and they are also capable of modeling materials that do not fracture, but compress due to plastic deformations. The snow model can also be used to mimic a granular material like sand. This is however accompanied by volume gain.

Figure 26 illustrates the possibilities of a simulation that uses only the volume corrected sand model. It is used to model sand, fluid and hyperelastic solids in the same scene. Regular sand is modeled with a density of 2200, $E =$

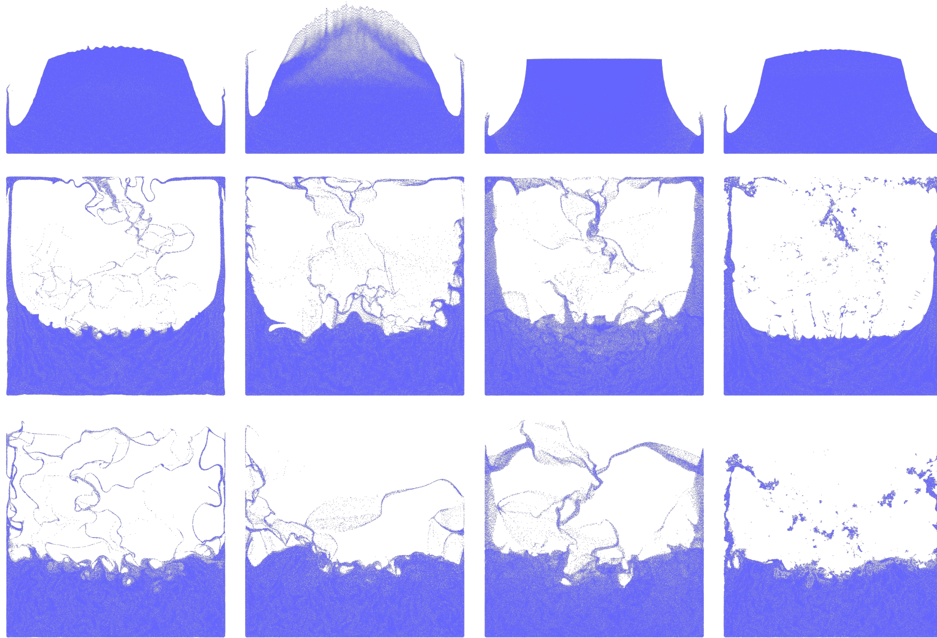


Figure 24: Fluids simulated using the fluid constitutive model (left), volume corrected sand (middle left), snow with $\mu = 0, \xi = 5, \theta_c = \theta_s = 0.4$ (middle right) and Neo Hookean with $\mu = 0.1$ (right).

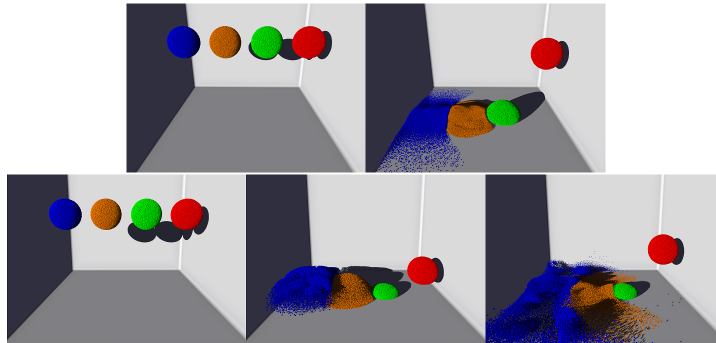


Figure 25: Spheres with varying material parameters. The top row uses the volume corrected sand model with cohesion $0, 0.004, 0.025$ and 1 from left to right. On the bottom row, the snow model is used. The blue sphere has higher Young's modulus, poisson ratio, critical compression and critical stretch compared to regular snow. The orange sphere has $\theta_s = 0$, while the green one uses $\xi = 1$. Lastly, the red sphere is reverted to hyperelasticity with $\theta_c = 1$ and $\theta_s = 100$.

340000, $\nu = 0.3$, $h_0 = 35, h_1 = 0, h_2 = 0.2, h_3 = 10$ and $c_C = 0$, where h_0, h_1 and h_3 are given in degrees. For the fluid, the friction angle is set to zero and hyperelasticity is achieved by setting the cohesion to a very large number.

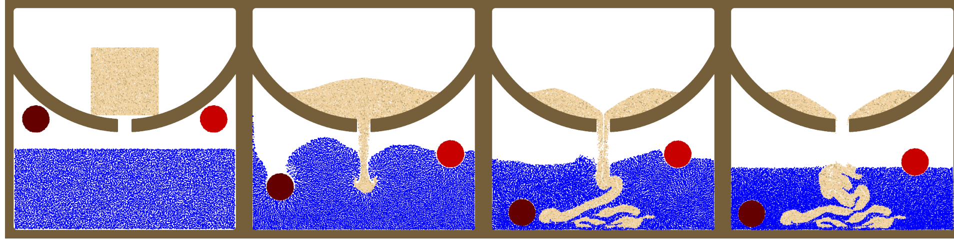


Figure 26: The volume corrected sand model is used to simulate hydrophobic sand, fluid ($\rho = 1000$) as well as hyperelastic circles ($\rho = 1600, \rho = 400$) interacting in one scene. Sand is stabilized by fluid pressure, preventing it from flowing freely. The less dense circle correctly stays buoyant while the other one sinks.

As this is a single-grid simulation, behavior between objects is collision. The result of this is that the sand acts similar to hydrophobic „magic sand“. Instead of flowing freely into a pile, the sand forms stable shapes under water. This is possible because the pressure of the fluid stabilizes the sand. The simulation also properly handles buoyancy. Note that all of this is handled implicitly by the method. No boundary handling is necessary and the individual materials retain their properties even if they are surrounded by particles of drastically different material properties. However, the submerged objects are surrounded by a thin film of fluid instead of touching each other and the floor directly. These are resolution artifacts. Thin portions of material cannot retain their properties as they only fill a small area or volume of the transfer function. This means that their mass is underrepresented on the corresponding grid nodes and thus their behavior is dictated by the different objects that surround it. Low resolution furthermore can give objects more thickness than they seem to have in their particle representation [8].

11 Multiple Constitutive Models

While a single constitutive model can already offer many possibilities, each model was designed for a specific purpose and is generally less suitable for other types of materials compared to a model that was designed for them. For example, the fluid model is better suited for fluids compared to the other models, but it cannot model anything else. The snow model on the other hand cannot represent granular materials like sand well, while the sand model cannot be used for snow. In order to integrate these types of materials in the same simulation, multiple constitutive models have to be used.

Figure 27 shows multiple hyperelastic objects dropping into moving fluid. The materials are modeled with separate constitutive models. Just like in a simulation that only uses one model, buoyancy is correctly handled in the simulation. The hyperelastic objects remain stable at their boundaries

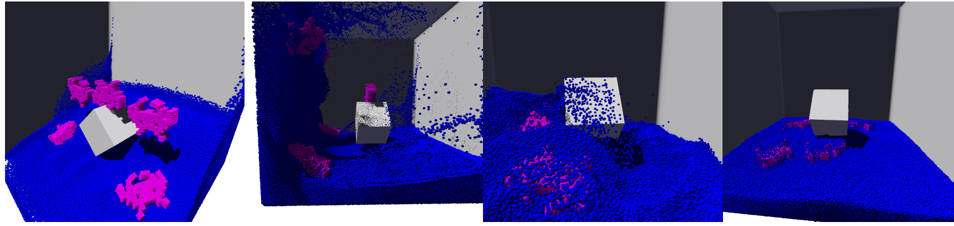


Figure 27: Multiple low-density hyperelastic (fixed corotated) objects falling into water. They are swept away by the current and stay afloat. The thin film of water covering the objects is a numerical artifact and not due to adhesive forces, but it could be exploited in rendering to make the objects look wet.

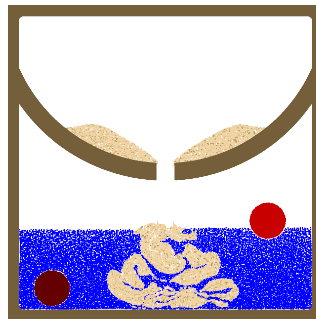


Figure 28: The scene from Fig. 26 with separate constitutive models for each material, resulting in a performance benefit and increased plausibility of sand flow inside the water.

without any fracturing. However, the simulation suffers from the same resolution problem that was previously described. This time it is expressed by individual fluid particles that remain stuck to the hyperelastic objects.

The scene from Fig. 26 is recreated using separate constitutive models in Fig. 28. This simulation only took 70% of the time of the one with a single constitutive model. The reason for this is that the fluid model and the fixed corotated model are much simpler to evaluate in comparison to the complex sand model, which incurs unnecessary calculations when used to describe these materials. Furthermore, the sand that is poured into the fluid forms a much more plausible shape in this simulation because the fluid behavior is different.

Buoyancy is further explored in Fig. 29 using cubes of lower, higher and the same density as the fluid. The lower the density (and thus mass) of the object, the easier it is pushed by the fluid. Same density objects still rise to the surface, although only a small part of the object rises above the surface. This happens because the fluid is slightly compressible, meaning that the fluid density increases with depth instead of being uniform. The behavior is thus correct and expected.

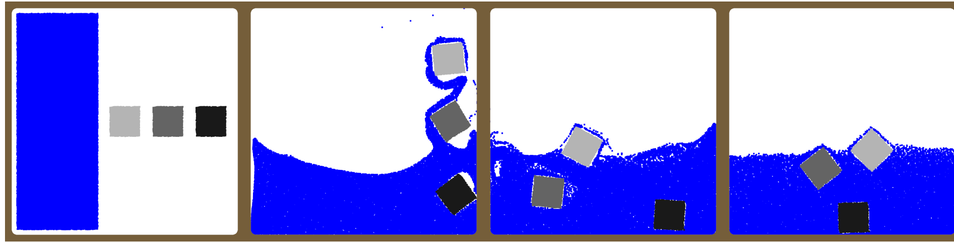


Figure 29: Hyperelastic cubes with densities 400, 1000 and 2600 are swept away by water ($\rho = 1000$). The cube with same density floats barely under the water surface, while the other ones either sink or stay afloat.

The usage of multiple constitutive models enables simulations that contain both sand and snow. Figure 31 shows two such simulations. Apart from the aforementioned resolution artifacts, this works as intended. Both materials retain their respective properties at their boundaries. However, it is possible to circumvent this by finely mixing two materials together. In this case, the properties are a mixture of both constitutive models. Figure 30 shows the results of first mixing different materials together with a fast rotating and moving cross-shaped collision object and subsequently compressing them to the right side of the scene. The compressive force is then quickly released, resulting in the mixture being flung to the left. For comparison, the same process is repeated with pure materials as well. In a snow-sand mixture, the strong sticky properties of snow are largely lost and only small chunks are formed. The behavior is more granular and overall gives the impression of a dry material. A snow-water mixture on the other hand forms a jelly-like material that quickly absorbs momentum. Sand-water mixtures act like a fluid during fast movement, but then quickly solidify and revert to more granular behavior as momentum is lost. This is similar to quicksand, which is a suspension of water and sand in the real world.

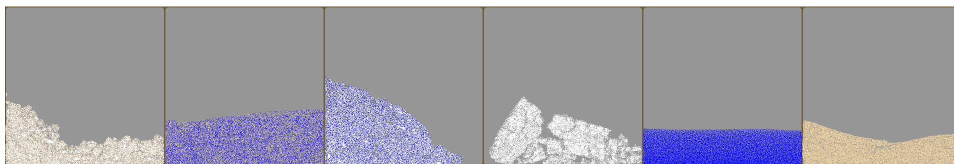


Figure 30: From left to right: Mixtures of sand and snow, water and sand, water and snow as well as pure snow, water and sand. Snow, water and sand use densities of 400, 1000 and 2200, respectively.

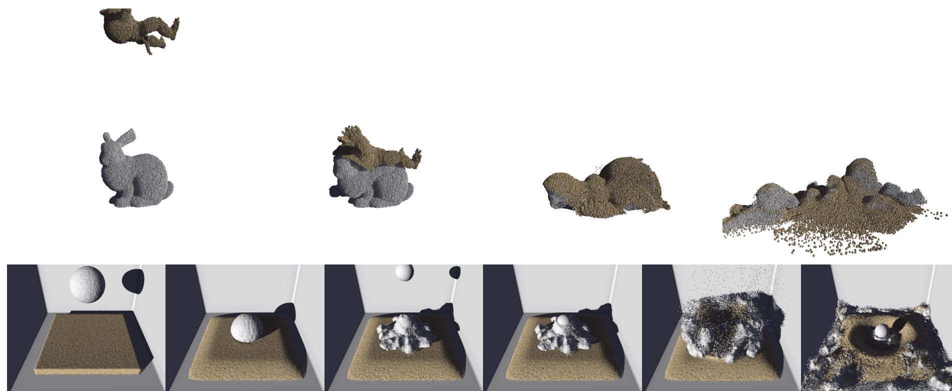


Figure 31: Sand and snow interacting in the same scene. Top row: A sand armadillo is dropped on a snow bunny. Bottom row: A snowball is dropped on sand and then hit by a fast moving rigid body sphere.

12 Porous Materials

In single-grid simulations, materials such as sand always act hydrophobic, because the MPM implicitly prevents objects from penetrating each other and occupying the same space. The multi-species approach for porous materials thus significantly expands possibilities. Figure 32 shows a sand column collapse which is made possible due to the porous interaction. Water can seep into sand and reduce its cohesion, allowing sand particles to be carried away by water via drag and decreasing the stability of the column until it collapses and is eventually fully submerged in water. The optional buoyancy term yields very similar results, but produces slightly more energetic water spray. It also slightly changes the speed at which water seeps into sand and results in a more pronounced directional flow. Overall, it results in more forces being applied to the sand that push towards the right. As the term adds a lot

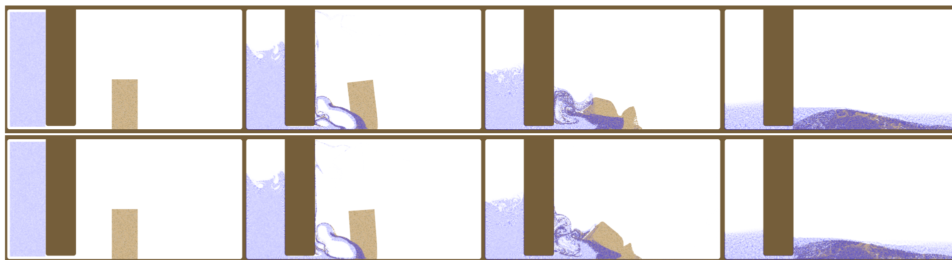


Figure 32: Porous material interaction without (top) and with the buoyancy term (bottom). If the buoyancy term is included, water seeps into the sand at a different rate and with a more directional flow. The difference is however small and comes at the cost of a 30% longer calculation, practically solely contributed by the P2G step.

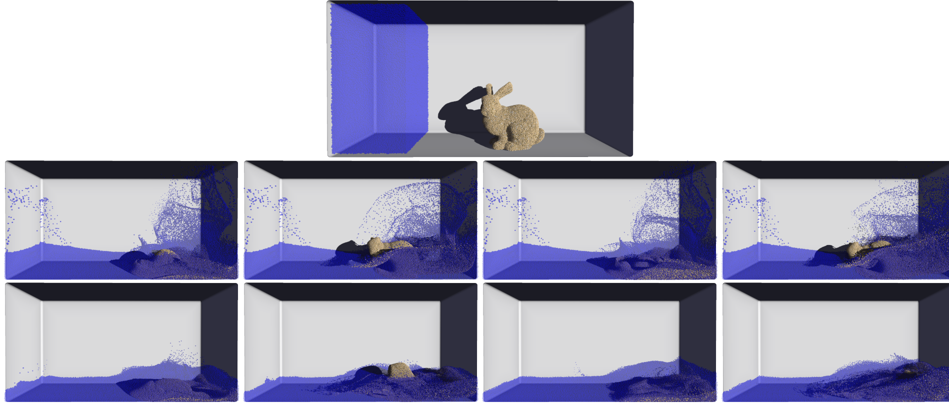


Figure 33: Two frames of a sand bunny being hit with water. The first and third column use a drag coefficient of 10^6, while the others use 10^7. Sand cohesion is reduced by the water volume fraction as in [3]. The last two columns additionally reduce the friction angle with rising volume fraction.

of computational complexity compared to the small changes caused by it, the remaining examples do not use it. It is however possible to construct examples in which these small changes result in a big difference, such as the column collapsing in a different direction.

The driving parameter in these simulations is the drag coefficient. Figure 33 shows how it affects the simulation. Lower drag coefficients mean that the fluid can more easily enter the porous species, which means that the sand loses cohesion faster. It also means that both species can move more freely as the momentum is exchanged slower. Both factors together cause the sand to collapse faster and end up fully submerged in this example. The simulation with high drag coefficient on the other hand keeps portions of the sand intact for longer and an island of sand remains after the collapse. High drag also causes sand to be washed away with the water flow, which typically looks more convincing.

Tampubolon *et al.* [3] only use the water volume fraction to reduce the sand cohesion. Changing this material interaction opens up new possibilities. For example, Fig. 33 contains simulations with the original term as well as an extended interaction that additionally reduces the friction angle according to

$$\phi_{Fp}^{s,n+1} = \phi_{Fp}^{s,0}(1 - 0.9\phi_p^{n+1}). \tag{125}$$

With the hardening model the formula is instead applied individually to h_0, h_1 and h_3. The modified interaction makes it possible to couple high drag with a full collapse. As the implementation allows the user to easily specify custom material interaction terms from inside the graphical user interface, simulations can be fine-tuned very fast. This also means that the framework can be

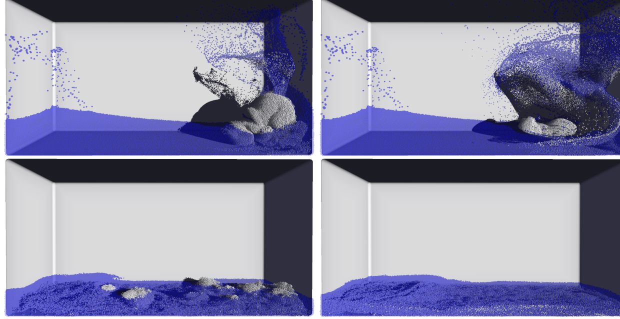


Figure 34: The simulation from Fig. 33 with a snow bunny instead. Both simulations use $c_E = 10^7$. The left one is run without any material interaction term, resulting in the bunny mostly being swept away and shredded due to its low mass. In the second simulation, the water volume fraction reduces μ which makes the snow behave more like a fluid.

used for more than just column collapse, as a different interaction term that instead increases cohesion can be used too.

The framework is not limited to porous sand and water mixtures. Figure 34 shows the previous simulation with a bunny made from snow instead of sand. Due to its low density, the snow is pushed against the wall as a result of water drag. Without any material interaction term these forces are already sufficient to fully fracture the bunny. In the real world, snow would however become mushy and eventually turn into water itself. This cannot be simulated exactly in this framework as the modeling of snow as porous means that it shares the same space with the fluid. It can however be approximated with a material interaction that modifies μ:

$$\mu_p^{n+1} = \mu_p^0(1 - \phi_p^{n+1}). \tag{126}$$

With this term the snow quickly dissolves and subsequently acts like a liquid. However, this liquid occupies the same space as the water species and thus sinks to the bottom. It could be possible to improve this interaction by changing which species a particle belongs to as it becomes liquid, which would prevent those particles from further porous interaction with the fluid species and thus correctly increase fluid volume. This is not implemented in this thesis and thus left to future work.

Section 6.1.3 derived a limit for the drag coefficient that depends on the time step and the masses of each grid node. This limit can be different for each cell. If it is exceeded, the result is that both species start repelling each other strongly, which adds energy into the system and typically causes a chain reaction of continuously increasing velocities. Clamping to the limit prevents this and instead causes simple collision if the drag coefficient is too large. However, drag coefficients of up to 10^8 resulted in stable simulations with time steps of 10^{-4}. This is in contrast to [3], where the authors report

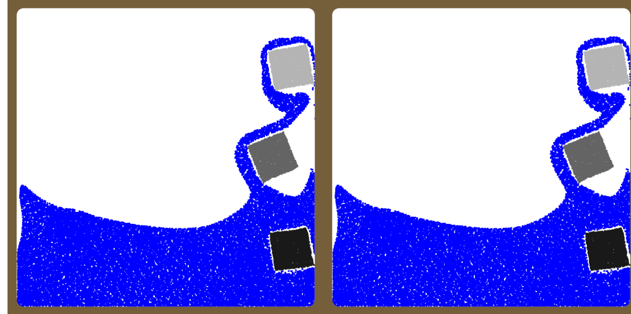


Figure 35: The simulation from Fig. 29 is run on a single grid (left) and in a multi-grid simulation where $c_E = c_{E,\max,i}$ for every node. Both images are exactly equivalent, which means that the derived limit for the drag coefficient exactly reproduces single-grid behavior.

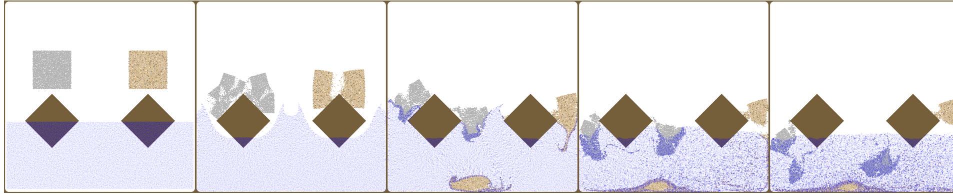


Figure 36: A multi-species simulation where one species consists of multiple constitutive models (snow and sand). Sand uses the extended material interaction term from Fig. 33, i.e. both cohesion and friction angle are reduced. The material interaction for snow reduces both μ and λ this time without letting them reach zero. Note how snow manages to stay afloat for some time until water has seeped in enough, which is due to its low density.

that they required time steps in the order of 10^{-5} and 10^{-6}, which is why they used implicit time stepping. The reason for this difference is most likely the scale of the scenes, as a higher mass per grid cell results in a lower limit for the drag coefficient. An interesting side result of the derived limit is that it can be exploited to revert a multi-grid simulation to behave like a single-grid simulation. This is done by always setting the drag coefficient to the limit for every cell. The result can be seen in Fig. 35. Single-grid behavior is reconstructed exactly. In theory, it would be possible to use more than two grids and achieve the same results. Potentially this could mean usage of one grid for each particle. While this may not have a practical application, it helps in understanding why multi-material simulations work so well in regular single-grid MPM. Every particle only ever considers itself and its own material and is coupled to other particles via an implicit momentum exchange due to the grid transfers.

Figure 36 combines the porous multi-species simulation with multiple constitutive models in the solid species, namely snow and sand. The material

interaction can be defined separately for each constitutive model in these cases. While not shown here, it is also possible to change the constitutive model as part of the material interaction. This works, but as stress is calculated differently depending on the model, it can result in a discontinuity which can be visible in the behavior of the material. The object might suddenly contract or expand from one time step to the next and this is also a potential source of errors. Future work could expand on this by introducing linear interpolation between constitutive models.

13 Particle Collisions

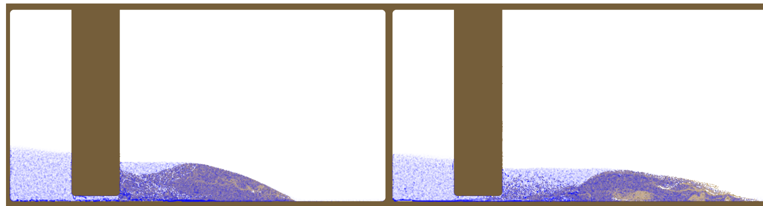


Figure 37: The simulation from Fig. 32 with particle collisions enabled. The left one was run without the buoyancy term and the right one with it. Particle collisions produce artifacts where the fluid flows along collision boundaries, which is strongly amplified where the water initially hits the sand. The result are severe volume artifacts which are visible as the dark blue areas in the images. This in turn amplifies the differences due to the buoyancy term, resulting in entirely different simulation behavior.

As mentioned in Section 4, Stomakhin *et al.* [1] used particle collisions in addition to grid collisions. This did not produce noticeable problems with their snow model. Particle collisions start being problematic when there is a lot of particle movement along collision surfaces, most notably in the case of fluids. This results in many particles occupying the same space. The problems proved to be even worse in multi-species simulations, where these artifacts are generated a lot at the interface where water initially hits the porous solid just above a collision object. As this results in high mass in a small volume, forces exerted by the fluid are artificially enhanced, leading to wildly different results. This is shown in Fig. 37, which shows the simulation from Fig. 32, this time with particle collisions. Artifacts are visible as dark blue spots. Many particles overlap here, which causes them to be rendered opaque despite transparency. The result of the simulation is very different from the previous example, and this time, the buoyancy term completely changes results, causing the column to collapse to the right instead of the left. This example is an extreme case. While artifacts are also generated at these spots in single-grid simulations, those do not use the momentum exchange

terms which heavily depend on the grid cell masses and subsequently are not affected as much. Simulations without fluids do not generate nearly as much artifacts and usage of particle collisions is fine for them.

Part V

Conclusion and Future Work

The MPM proved to be very effective for multi-material simulations. Not only does it implicitly handle self-collision and fracture, but also collisions between separate objects without the need for additional boundary handling. These properties apply even if drastically different material parameters are used. This work shows that this also applies to the usage of multiple constitutive models, which greatly expands the material variety that can be simulated in a single scene. While only weakly compressible fluids, snow, sand and hyperelastic materials have been used during evaluation, the general framework is independent of the selected constitutive model and can thus easily be extended with new materials.

Furthermore, the extension of the method for porous material interaction is straightforward. The proposed generalization of the original approach by Tampubolon *et al.* [3] is very flexible and opens up many possibilities to adjust how materials interact. This does not necessarily have to be physically motivated and can instead be used to generate the desired visual result, which is very useful for computer graphics applications. Multi-species simulations naturally combine with the usage of multiple constitutive models. Future work could try to build on this and research how transitioning between species as well as constitutive models can be introduced to the framework. This could build upon work such as [9] that can simulate melting, i.e. transitions from solid state to fluid state.

A general weakness of MPM are incompressible and nearly incompressible materials. Stomakhin *et al.* [9] developed an extended MPM that alleviates this. However, their approach is tied to a specific constitutive model. Future work could research a generalization that is independent of the constitutive model.

This thesis additionally introduced a simple GPU implementation of the MPM that uses the rasterization pipeline to resolve write conflicts during particle-to-grid transfers for both 2D and 3D. As the implementation uses OpenGL, it is general enough to work across different hardware. No vendor-specific or non-standard extensions to the instruction set are necessary, meaning that the implementation works across common consumer hardware. The algorithm can also be implemented with other graphics APIs such as Vulkan or Direct3D. As this thesis did not focus on performance, there is still a lot of potential for optimization. For example, some of the ideas used in [4] such as spatial sorting of particles could be integrated. It would also be interesting to extend the implementation to handle sparse grids as this can conserve memory and removes the need to specify the bounds of the simulation. This is however non-trivial to achieve with the proposed rasterization-based implementation.

References

[1] A. Stomakhin, C. Schroeder, L. Chai, J. Teran, and A. Selle, „A material point method for snow simulation“, *ACM Transactions on Graphics (TOG)*, vol. 32, no. 4, pp. 1–10, 2013.

[2] G. Klár, T. Gast, A. Pradhana, C. Fu, C. Schroeder, C. Jiang, and J. Teran, „Drucker-prager elastoplasticity for sand animation“, *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, pp. 1–12, 2016.

[3] A. P. Tampubolon, T. Gast, G. Klár, C. Fu, J. Teran, C. Jiang, and K. Museth, „Multi-species simulation of porous sand and water mixtures“, *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, pp. 1–11, 2017.

[4] M. Gao, X. Wang, K. Wu, A. Pradhana, E. Sifakis, C. Yuksel, and C. Jiang, „GPU optimization of material point methods“, *ACM Transactions on Graphics (TOG)*, vol. 37, no. 6, pp. 1–12, 2018.

[5] G. Daviet and F. Bertails-Descoubes, „A semi-implicit material point method for the continuum simulation of granular materials“, *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, pp. 1–13, 2016.

[6] D. Ram, T. Gast, C. Jiang, C. Schroeder, A. Stomakhin, J. Teran, and P. Kavehpour, „A material point method for viscoelastic fluids, foams and sponges“, in *Proceedings of the 14th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2015, pp. 157–163.

[7] Y. Yue, B. Smith, C. Batty, C. Zheng, and E. Grinspun, „Continuum foam: A material point method for shear-dependent flows“, *ACM Transactions on Graphics (TOG)*, vol. 34, no. 5, pp. 1–20, 2015.

[8] C. Jiang, T. Gast, and J. Teran, „Anisotropic elastoplasticity for cloth, knit and hair frictional contact“, *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, pp. 1–14, 2017.

[9] A. Stomakhin, C. Schroeder, C. Jiang, L. Chai, J. Teran, and A. Selle, „Augmented MPM for phase-change and varied materials“, *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, pp. 1–11, 2014.

[10] A. Stomakhin, R. Howes, C. Schroeder, and J. M. Teran, „Energetically consistent invertible elasticity“, in *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, Eurographics Association, 2012, pp. 25–32.

[11] C. Jiang, C. Schroeder, A. Selle, J. Teran, and A. Stomakhin, „The affine particle-in-cell method“, *ACM Transactions on Graphics (TOG)*, vol. 34, no. 4, pp. 1–10, 2015.

[12] Y. Hu, Y. Fang, Z. Ge, Z. Qu, Y. Zhu, A. Pradhana, and C. Jiang, „A moving least squares material point method with displacement discontinuity and two-way rigid body coupling“, *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, pp. 1–14, 2018.

[13] J. Wittenburg, *Festigkeitslehre - Ein Lehr- und Arbeitsbuch*, 3. Auflage. Berlin Heidelberg New York: Springer-Verlag, 2011, ISBN: 978-3-642-56457-4. DOI: 10.1007/978-3-642-56457-4.

[14] F. Irgens, *Continuum Mechanics*. Berlin Heidelberg: Springer Science & Business Media, 2008, ISBN: 978-3-540-74298-2. DOI: 10.1007/978-3-540-74298-2.

[15] H. Hencky, „Uber die Form des Elastizitatsgesetzes bei ideal elastischen Stoffen“, *Zeit. Tech. Phys.*, vol. 9, pp. 215–220, 1928.

[16] J. Arghavani, F. Auricchio, and R. Naghdabadi, „A finite strain kinematic hardening constitutive model based on Hencky strain: general framework, solution algorithm and application to shape memory alloys“, *International Journal of Plasticity*, vol. 27, no. 6, pp. 940–961, 2011.

[17] D. Sulsky, S.-J. Zhou, and H. L. Schreyer, „Application of a particle-in-cell method to solid mechanics“, *Computer physics communications*, vol. 87, no. 1-2, pp. 236–252, 1995.

[18] C. Jiang, C. Schroeder, J. Teran, A. Stomakhin, and A. Selle, „The material point method for simulating continuum materials“, in *ACM SIGGRAPH 2016 Courses*, 2016, pp. 1–52.

[19] Y. Hu, X. Zhang, M. Gao, and C. Jiang, „On hybrid lagrangian-eulerian simulation methods: practical notes and high-performance aspects“, in *ACM SIGGRAPH 2019 Courses*, 2019, pp. 1–246.

[20] M. Steffen, R. M. Kirby, and M. Berzins, „Analysis and reduction of quadrature errors in the material point method (MPM)“, *International journal for numerical methods in engineering*, vol. 76, no. 6, pp. 922–948, 2008.

[21] F. H. Harlow and M. Evans, „A machine calculation method for hydrodynamic problems“, *LAMS-1956*, p. 32, 1955.

[22] J. Brackbill, „The ringing instability in particle-in-cell calculations of low-speed flow“, *Journal of Computational Physics*, vol. 75, no. 2, pp. 469–492, 1988.

[23] J. U. Brackbill, D. B. Kothe, and H. M. Ruppel, „FLIP: a low-dissipation, particle-in-cell method for fluid flow“, *Computer Physics Communications*, vol. 48, no. 1, pp. 25–38, 1988.

[24] E. Love and D. L. Sulsky, „An unconditionally stable, energy–momentum consistent implementation of the material-point method“, *Computer Methods in Applied Mechanics and Engineering*, vol. 195, no. 33-36, pp. 3903–3925, 2006.

[25] C. Jiang, *The material point method for the physics-based simulation of solids and fluids*. University of California, Los Angeles, 2015.

[26] M. Becker and M. Teschner, „Weakly compressible SPH for free surface flows“, in *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, 2007, pp. 209–217.

[27] D. Drucker and W. Prager, „Soil mechanics and plasticity analysis of limit design, Q“, *Appl. Math*, vol. 10, 1952.

[28] C. M. Mast, „Modeling landslide-induced flow interactions with structures using the material point method“, PhD thesis, 2013.

[29] C. M. Mast, P. Arduino, P. Mackenzie-Helnwein, and G. R. Miller, „Simulating granular column collapse using the material point method“, *Acta Geotechnica*, vol. 10, no. 1, pp. 101–116, 2015.

[30] S. Bandara, A. Ferrari, and L. Laloui, „Modelling landslides in unsaturated slopes subjected to rainfall infiltration using material point method“, *International Journal for Numerical and Analytical Methods in Geomechanics*, vol. 40, no. 9, pp. 1358–1380, 2016.

[31] P. Mackenzie-Helnwein, P. Arduino, W. Shin, J. Moore, and G. Miller, „Modeling strategies for multiphase drag interactions using the material point method“, *International journal for numerical methods in engineering*, vol. 83, no. 3, pp. 295–322, 2010.

[32] D. Levin, „The approximation power of moving least-squares“, *Mathematics of computation*, vol. 67, no. 224, pp. 1517–1531, 1998.

[33] T. Belytschko, Y. Y. Lu, and L. Gu, „Element-free Galerkin methods“, *International journal for numerical methods in engineering*, vol. 37, no. 2, pp. 229–256, 1994.

[34] C. Fu, Q. Guo, T. Gast, C. Jiang, and J. Teran, „A polynomial particle-in-cell method“, *ACM Transactions on Graphics (TOG)*, vol. 36, no. 6, pp. 1–12, 2017.

[35] K. Wu, N. Truong, C. Yuksel, and R. Hoetzlein, „Fast fluid simulations with sparse volumes on the GPU“, in *Computer Graphics Forum*, Wiley Online Library, vol. 37, 2018, pp. 157–167.

[36] T. Annen, T. Mertens, H.-P. Seidel, E. Flerackers, and J. Kautz, „Exponential shadow maps.“, in *Graphics Interface*, ACM Press, 2008, pp. 155–161.

[37] A. McAdams, A. Selle, R. Tamstorf, J. Teran, and E. Sifakis, „Computing the singular value decomposition of 3x3 matrices with minimal branching and elementary floating point operations“, University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 2011.

[38] R. Bridson, „Fast Poisson disk sampling in arbitrary dimensions.“, *SIGGRAPH sketches*, vol. 10, pp. 1 278 780–1 278 807, 2007.

[39] J. A. Bærentzen and H. Aanaes, „Signed distance computation using the angle weighted pseudonormal“, *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 3, pp. 243–253, 2005.

[40] L. Bavoil and K. Myers, „Order independent transparency with dual depth peeling“, *NVIDIA OpenGL SDK*, vol. 1, p. 12, 2008.

[41] J. C. Hart, „Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces“, *The Visual Computer*, vol. 12, no. 10, pp. 527–545, 1996.