Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Aslak Sheker Mkadmi

Real-Time Hyperelastic Simulation for Computer Graphics with the Material Point Method

Masteroppgave i Datateknologi
Veileder: Theoharis Theoharis
Juni 2021

NTNU
Kunnskap for en bedre verden

Aslak Sheker Mkadmi

Real-Time Hyperelastic Simulation for Computer Graphics with the Material Point Method

Masteroppgave i Datateknologi
Veileder: Theoharis Theoharis
Juni 2021

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

NTNU

Kunnskap for en bedre verden

Abstract

The Material Point Method is a physical simulation method for simulating bodies described using continuum mechanics. Since its use in Disney's animated movie Frozen[Sto+13], the technique has seen widespread use in computer graphics. However, research has been almost solely focused on offline rendering. This thesis intends to examine the practicability of real-time hyperelastic simulation using a CPU based implementation of the Material Point Method.

Sammendrag

Materialpunktsmetoden er en fysisk simuleringsmetode for å simulere legemer som er beskrevet ved hjelp av kontinuumsmekanikk. Siden metodens bruk i Disney's animasjonsfilm Frozen[Sto+13] har metoden sett mye bruk i datagrafikk. Til tross for dette har forskning nesten utelukkende vært fokusert på offline rendering. Denne oppgaven har som formål å undersøke gjennomførbarheten til hyperelastisk simulering i sanntid ved hjelp av en CPU-basert implementasjon av Materialpunktsmetoden.

Acknowledgements

I would like to thank my advisor Theoharis Theoharis for his support and guidance throughout the project. His feedback has been invaluable for the completion of this thesis.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation And Goals

From the time of its inception to the present day, MPM has been a popular method within various engineering disciplines [Vau+20]. Within the field of computer graphics however, the method has only been of particular interest to researchers since 2013[Sto+13]. During this period, several papers have focused on improving the performance and stability of the method[Fan+18][Gao+18][Wan+20][Hu+18a]. In addition to this, great progress has been made in developing various extensions that allow MPM to simulate more complex materials and phenomena. This, coupled with its proven use in VFX by animation studios like Disney, has made MPM a very popular choice for animated films. However, within real-time applications such as video games, the method is essentially unused.

The purpose of this thesis is to explore the viability of using MPM in interactive real-time applications. In order to achieve this, the following goals are set:

- Map out and implement various optimization techniques and extensions utilized in state-of-the-art MPM, and determine their suitability for real-time simulation.
- Benchmark and evaluate different scene configurations, material methods, and potential concessions that can enable real-time hyperelastic simulation with MPM.

In addition, as MPM is a theoretically dense method, particularly for those lacking a background in physics and computational math, a large part of the thesis is dedicated to explaining the necessary background theory for the method in a structured manner.

As part of the scope of the thesis, a conscious choice has been made to focus on CPU implementations. While recent GPU implementations of MPM have been able to demonstrate massive runtime improvements [Gao+18][Wan+20] over CPU methods, these methods are clearly directed towards offline simulation. In real-time applications, particularly video games, it is highly desirable to have the resources of the GPU available for rendering. For this reason, the primary focus of this thesis is on CPU-based methods.

1.2 Overview

The rest of this thesis is structured as follows:

- **Background**
- **Continuum Mechanics**
- **The MPM Algorithm**
- **Implementing Real Time MPM**
- **Evaluation**
- **Conclusion and Further Work**

The background chapter is divided into four parts: First, time integration of PDEs is discussed. This is subsequently related to the Finite Element Method, a method which is closely related to MPM. Afterwards, an explanation is given of Lagrangian and Eulerian specifications from the context of fluid mechanics. These concepts are subsequently expanded upon in the continuum mechanics chapter. Following this, an explanation is given of basic concepts necessary for effectively utilizing CPU caches. As the standard MPM algorithm is a fairly memory bound method, knowledge of these concepts is key to developing a high performant implementation. Finally, related works are discussed, both for the Material Point Method itself, as well as other real-time physical simulation methods popular in computer graphics.

Following the background chapter, a chapter is dedicated to outlining basic concepts in continuum mechanics. These concepts are discussed in a chapter separate from the background chapter, as they are both a) Fairly complex and requiring a thorough explanation and b) absolutely crucial to understanding MPM. The chapter starts with the basic continuum assumption, and gradually explains concepts until arriving at the material models commonly used for simulating hyperelastics with MPM in computer graphics.

In the fourth chapter, the basic material point method, outlined in [SZS95] is described. In addition to an explanation of each stage of the method, the choice of basis functions is also discussed.

The fifth chapter discusses various extensions and implementation details that are necessary in order to implement real time MPM. The chapter is divided into five sections, with the first four describing various implementation details and the fifth describing the overall choices made for the implementation written for the purposes of evaluation.

Following this, the developed method is evaluated in the evaluation chapter. A number of different scenarios that are likely to impact performance are evaluated, with the goal of determining which concessions are viable for reducing the runtime of the simulation without sacrificing too much in terms of visual quality.

Finally, the results from the evaluation chapter are discussed. A conclusion relating to the goals of 1.1 is given. Potential further work within the field of real-time MPM for computer graphics is also discussed.

Chapter 2

Background

The material point method is essentially a method for solving PDEs. It was originally conceived by Sulsky et al. as an extension of PIC/FLIP methods to solid mechanics[SCS94][SZS95], but has subsequently been extended to encompass all continuum bodies.

Like PIC/FLIP, MPM is a Hybrid Lagrangian-Eulerian simulation method. As a PDE solver, it is conceptually very similar to the Finite Element Method (FEM). In order to simulate continuum bodies with MPM one has to solve PDEs describing the behavior of the body, in particular the conservation of momentum. Much like FEM, it is necessary to rephrase this PDE in its weak form, so that it can be solved through the use of basis functions.

For explaining partial differential equations and Lagrangian/Eulerian simulations, the example of computational fluid dynamics will be used. MPM uses discretizations based on continuum mechanics, which can be used to describe solids, fluids and gases. However, the language of continuum mechanics is fairly complex, so concepts are described in this context in a separate chapter. In particular, it is easier to get the idea behind Lagrangian/Eulerian simulations without also simultaneously having to understand basic continuum mechanics.

2.1 Partial Differential Equations (PDEs)

Differential equations are equations that can describe the development of some system based on the relation of a function to its derivatives. They are extremely useful in describing physical phenoma and processes.

If the functions used to describe the system are univariate (one independent variable), then the system is expressed through ordinary derivatives (e.g. $y' = -y^2$). This class of differential equations are called *Ordinary Differential Equations*, or ODEs.

If the functions used to describe the system are multivariate, then the system is expressed through partial derivatives. These are known as *Partial Differential Equations*, or PDEs. These equations are often spatiotemporal, which is to say they vary both in space (often denoted x) and time (often denoted t).

A common example of a PDE is the *heat equation*, which describes how heat diffuses through some region of space (given by spatial variables $x_1 x_2, ..x_n$) in time (given by t), for a given function $u(x, t)$:

$$\frac{\partial u}{\partial t} = \Delta u$$

A more relevant example are the incompressible Navier-Stokes equations. These equations are analogous to Newton's second law ($F = ma$), and describe the motion of a fluid in time. Fluid simulations in computer graphics are generally based on numerically solvable discretizations of the Navier Stokes equations[Bri08]:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + \nu \nabla \cdot \nabla \vec{u}$$
$$\nabla \cdot \vec{u} = 0$$

Figure 2.1: The Navier-Stokes PDEs for incompressible flow.

In MPM and physical simulation in general, the material being simulated is often described with a spatiotemporal PDE. Assuming some initial state $[x = X, t = 0]$, simulation can be done by numerically solving the PDE to get the state of the system at $t_{n+1} = t_n + \Delta t$ (more in 2.1.6), where Δt is some (small) time step.

2.1.1 Differentiability Class

The PDEs we are interested in for simulating continuum bodies are usually defined over the volume of the body and its surface. The corresponding functions have a *Differentiability Class* C, which is the degree to which the function has continuous derivatives. For example, a function C^1 is continuously differentiable once, and a C^2 function is continuously differentiable twice. A C^0 function is not continuously differentiable, but may be piecewise differentiable.

2.1.2 Weak formulation

In certain cases, the PDE we want to solve imposes differentiability requirements that are simply too hard to uphold, making it impossible to find a classical solution[Sul20]. For MPM, we seek to solve the PDE for conservation of linear momentum, which includes a term for the spatial divergence of stress, $\nabla_x \cdot P$. We are unable to find a numerical solution that ensures the smoothness of this term, but we may be able to find a solution that is piecewise continuously differentiable. To do this, we reformulate the PDE in its *weak form*.

As an example, we will derive the weak form of the 1D Poisson equation defined over Ω with a homogeneous Dirichlet boundary condition, whose strong form is:

$$-\nabla^2 u(x) = f(x) \tag{2.1}$$

$$u(x) = 0, \text{ for } x \in \partial\Omega \tag{2.2}$$

Where f is some given function (e.g. $f = 0$, which yields Laplace's equation). Notice that u has to have continuous second derivatives, i.e. $u \in C^2(\Omega)$.

In order to find its weak form, we first multiply our PDE by a suitable *test function* v:

$$-\nabla^2 uv = fv \tag{2.3}$$

Afterwards, we integrate the equation over the bounds Ω:

$$-\int_\Omega v\nabla^2 u = \int_\Omega fv \tag{2.4}$$

At this point, we can transform the equation by applying integration by parts and the divergence theorem to the left-hand side:

$$\int_\Omega v\nabla^2 u = \int_\Omega \nabla\cdot(v\nabla u) - \int_\Omega \nabla v\cdot\nabla u = \int_{\partial\Omega} v\frac{\partial u}{\partial n} - \int_\Omega \nabla v\cdot\nabla u \tag{2.5}$$

Where n is the surface normal field. With this, we have transformed equation 2.1 into the weak form:

$$\int_\Omega \nabla v\cdot\nabla u = \int_\Omega vf + \int_{\partial\Omega} v\frac{\partial u}{\partial n} \tag{2.6}$$

Where $\int_{\partial\Omega} v\frac{\partial u}{\partial n} = 0$ due to boundary condition 2.2. In this form, the $u \in C^2$ requirement has been transformed into $u \in C^1$ by transferring a differentiation onto the test function v.

With this in mind, a reasonable next question would be to ask *why* this works, and where u and v must lie. Unfortunately, this is a bit more involved. The process of determining the space that holds solutions to the weak formulation is beyond the scope of this thesis, but is given in [Arb20]. It can be shown that both u and v must belong to the *Sobolev Space* $\mathcal{H}^1(\Omega)$.

Both FEM and MPM can be formulated as numerically solving weak form PDEs. For MPM, we seek to numerically solve the weak form of conservation of linear momentum.

2.1.3 Galerkin Method and Basis Functions

Functions u and v lie in the *Sobolev Space* $\mathcal{H}^1(\Omega)$. In a numerical solution, we wish to approximate u. We do so by looking instead at a *finite-dimensional* subspace of $\mathcal{H}^1(\Omega)$, referred to here as $\mathcal{V}_N$. We further assume that the subspace $\mathcal{V}_N$ can be represented by a finite number of *basis functions*, given as $\{\phi_1, \phi_2, ..., \phi_N\}$.

As our approximation u_h is assumed to lie in the subspace, we have:

$$u_h = \sum_{n=1}^{N} u_n\phi_j \tag{2.7}$$

Where u_n denotes the coefficients.

Instead of testing u against an arbitrary $v \in \mathcal{H}^1(\Omega)$, we can say that we instead only test against $v \in (V)_N$, such that $v_h = \sum_{n=1}^{N} v_{n'} \phi_{n'}$. This is known as *Galerkin's Method* for solving PDEs[Sul20], and is what FEM is based on. MPM in turn can be said to be a FEM-style method in terms of solving for conservation of linear momentum.

2.1.4 Finite Element Method

FEM is a particular type of Galerkin method. It is based on the observation that a smooth function oftentimes can be approximated to arbitrary accuracy via *piecewise polynomials*[Arb20]. In FEM, the basis functions are defined on a mesh of *elements* (triangles/rectangles in R^2, tetrahedra/parallelepipeds in R^3). A basis function ϕ_i is given for each node i in the mesh. These basis functions commonly have minimal support, i.e. $\phi_i(j) = 0 \mid i \neq j$.

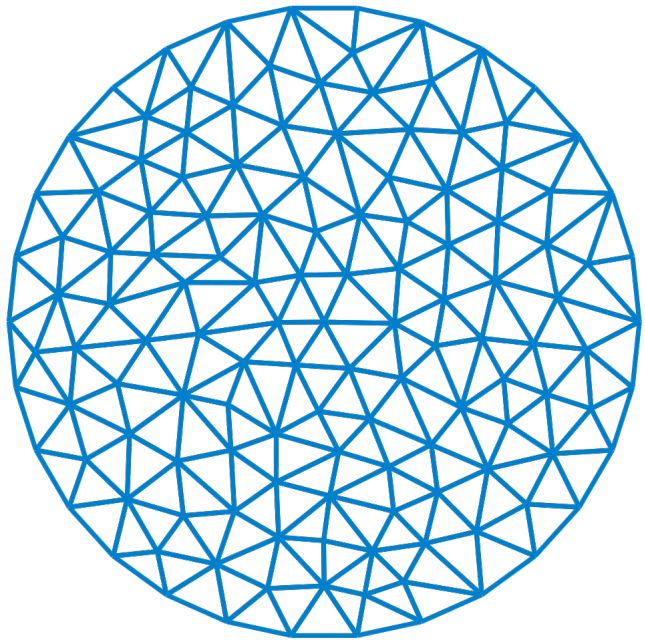

Figure 2.2: Example of a typical FEM mesh. Each triangle is a mesh element.

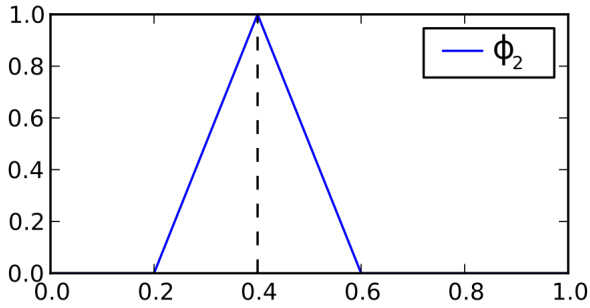

Figure 2.3: A linear 1D basis function.

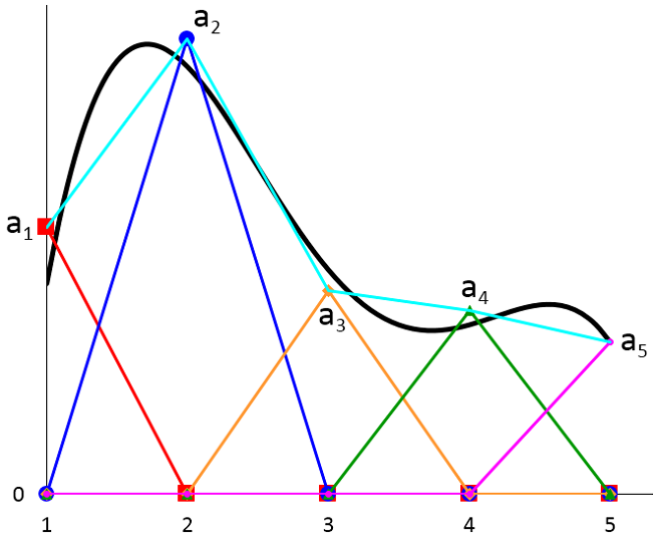

Figure 2.4: Basis functions and their coefficients can approximate the function we wish to solve for. Image from [Liu15]

2.1.5 Numerically solving integrals

As part of FEM and MPM, we need to numerically solve integrals over elements. To do this, we approximate the integrals with quadrature points. The Gaussian quadrature rule, used in FEM, approximates the definite integral of a function f in the following way:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} \omega_i f(x_i) \tag{2.8}$$

Where x_i are the quadrature points, and ω_i are their respective weights. Using n quadrature points, the weights and points are chosen such that polynomials of degree $2n-1$ or less can be exactly integrated.

MPM can be viewed as a special Lagrangian FEM, in which the particles, not the Gauss points, serve as integral points[Hua+08]. The weight ω_p is equivalent to the particle volume, V_p. The Eulerian scratch-pad grid serves as the mesh, and the grid cells are the elements. As these elements are pairwise disjoint, the integral we wish to solve (conservation of linear momentum) can be evaluated by integrating over single elements.

2.1.6 Time integration of PDEs

As our PDEs are temporal as well as spatial, we need some way of evolving them through time. For this purpose, there are two main strategies:

Explicit methods calculate the state of the system at time $t + \Delta t$ from the current state t, i.e. if the current system state is $Y(t)$, then $Y(t + \Delta t) = F(Y(t))$, where F is some function on the system.

Implicit methods calculate the state of the system at time $t + \Delta t$ from an *implicit equation*, i.e. $G(Y(t), Y(t + \Delta T)) = 0$ is solved for $Y(t + \Delta t)$.

For MPM, the **symplectic** or semi-explicit (often just called explicit) method is a very popular choice. This method can be applied to spatio-temporal PDEs where position and velocity are computed. With the symplectic method, the velocity is calculated in an explicit manner, but the position uses the newly calculated velocity. In other words, position and velocity are updated as:

$$x_i^{n+1} = x_i^{n+1} + \Delta t v_i^{n+1} \tag{2.9}$$

$$v_i^{n+1} = v_i^n + \Delta f_i(x_i^n)/m_i \tag{2.10}$$

Note how x_i^{n+1} is computed using v_i^{n+1} instead of v_i^n, which distinguishes it from the explicit method.

In terms of performance in MPM implementations, explicit/symplectic methods are much faster and easier to implement when compared to implicit methods. However, implicit methods can often take drastically higher timesteps, potentially giving them an edge in performance. Nevertheless, as noted by [Fan+18], ex-

plicit/symplectic methods often have the best runtime in the end, as they are easier to optimize. A notable exception are particularly stiff materials such as concrete, which require prohibitively low timesteps for stability with explicit methods [SSS20].

2.2 Lagrangian vs Eulerian

In computational fluid dynamics, the main challenge is keeping track of the development of flow fields. Flow fields represent the current state of various quantities in the fluid. Velocity, represented as a vector field, is the primary flow quantity of interest. Other quantities of interest are density, pressure, and temperature, represented as scalar fields.

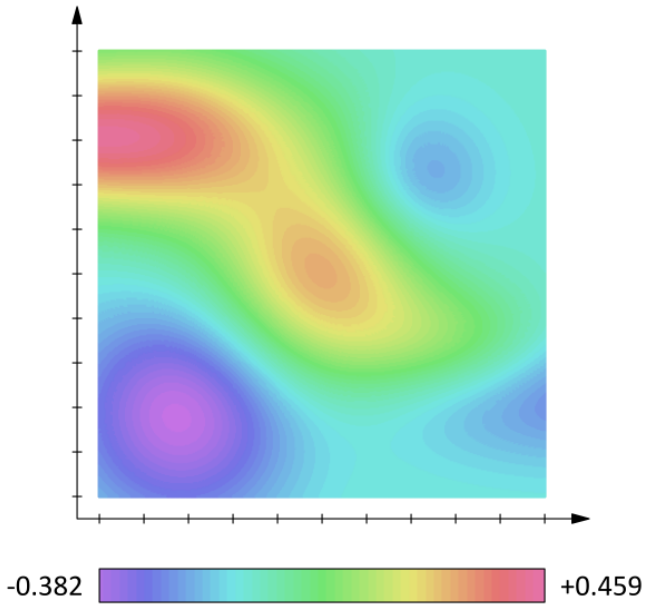

Figure 2.5: A 2D scalar field.

There are two main ways of describing the flow fields of a fluid: Eulerian and Lagrangian. When discretizing Navier Stokes for fluid simulations, this distinction is significant. In fact, physical simulation methods are generally categorized in terms of which description they use.

In the **Eulerian** view, the flow quantities (velocity, heat, etc) are given as a function of space x and time t. For the velocity, this is usually written $u(x,t)$[GS08]. For a given time t, one can think of the Eulerian view as providing a kind of "snapshot" of the current state of the fluid quantities. In an Eulerian discretization, it is common to use a uniformly spaced grid in R^d, where d is the spatial dimension of the simulation. Each cell in the grid keeps track of the current state of the flow quantities at the location of the cell.

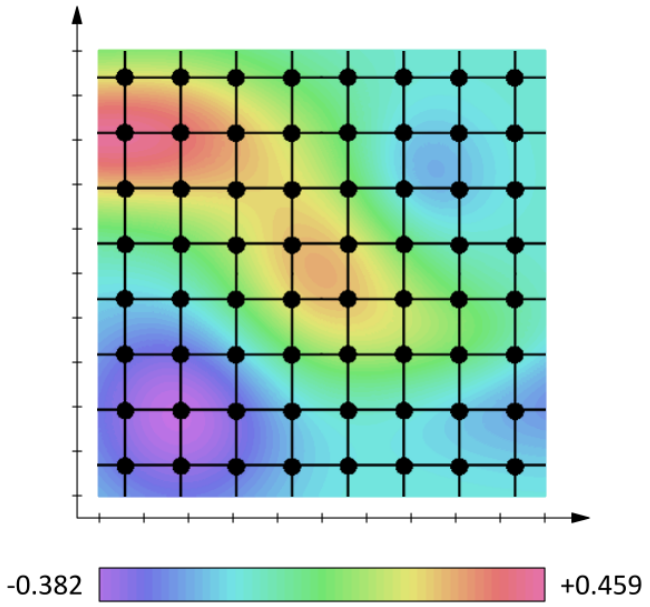

Figure 2.6: Eulerian discretization of a scalar field. Quantities are tracked at each dot.

In the **Lagrangian** view, we focus on the fact that the fluid consists of several individual pieces of matter (particles). Instead of tracking the development of the field through time via a spatiotemporal function, we keep track of the quantities attached to some particle at time t. A common way of specifying the particles is to refer to them by their position a at some initial configuration t_0. We can then get the quantities (like velocity) of the particle at time t with $v(a,t)$. We usually assume that no two particles ever occupy the same space at the same time.

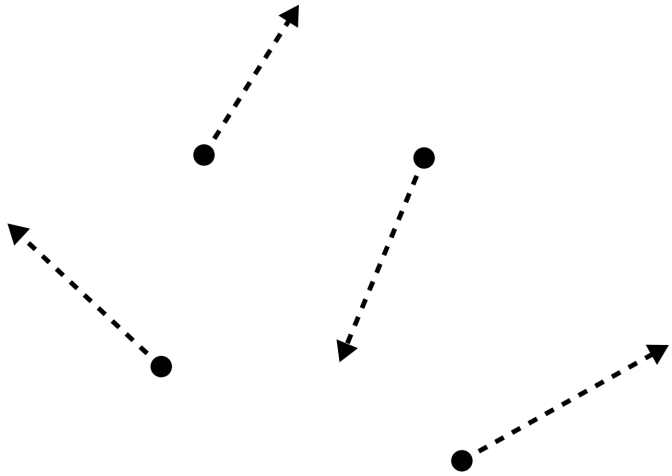

Figure 2.7: Lagrangian discretizations track quantities on free-moving particles.

Both Lagrangian and Eulerian representations have their perks, but neither is clearly superior. For fluid simulation, The Eulerian representation is generally faster and more popular[Bri08]. However, it suffers from issues with fluid advection, as particle movement has to be represented by a grid of fixed points. Having to interpolate velocities from grid cells often leads to diffusion. The Lagrangian representation, on the other hand, suffers from no such issue. As each particle is represented individually and is free to move in R^d, advection is as simple as $\hat{x}_p = x_p + \Delta t v_p$. Because each representation has its own benefits, a popular third alternative is to use both representations, favoring whichever is most appropriate for the current task. This method is known as hybrid Lagrangian-Eulerian, and is what PIC/FLIP and MPM use.

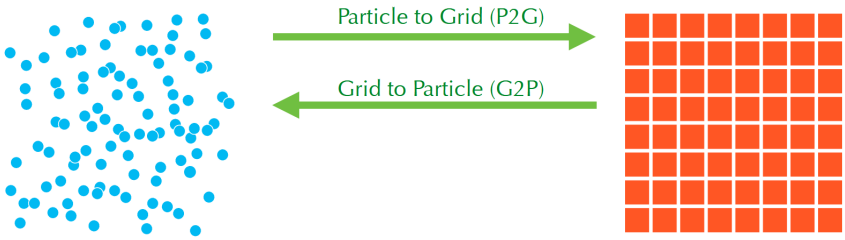

Figure 2.8: Hybrid Lagrangian-Eulerian discretizations use both representations. Image from [Hu+18a].

2.3 Cache utilization

As MPM is a hybrid Lagrangian-Eulerian method, it requires frequent memory accesses to large arrays of particles and grid cells. So much so, in fact, that the time spent reading from and writing to memory generally outweighs the time spent on actual computation. For this reason, knowledge of how to effectively utilize CPU caches is crucial to performant MPM.

Most modern general purpose computers have a memory hierarchy. The hierarchy consists of different types of memory with different sizes and access times, ranging from very large and very slow to very small and very fast. The purpose of structuring memory like this this is to exploit *locality*. In a general program flow, memory accesses are likely to be both temporally and spatially local [Dre07]. Temporal locality states that access to data is likely to be clustered in time, while spatial locality states that data accesses are likely to be close in memory to previous accesses.

The memory hierarchy usually consists of three main components:

1. Secondary memory, for persistent storage.
2. Primary memory, for fast-access transient storage.
3. Registers and Caches, for even faster access to frequently used data.

The secondary memory usually comes in the form of HDD's and SSD's. This memory is intended for storing data that persists past processes or reboots. Persistent memory is not frequently using during physical simulation with MPM (barring some potential inputs), and as secondary memory is by far the slowest, it is not of particular interest.

Primary memory usually takes the form of Dynamic RAM (DRAM). The vast majority of consumer computers used for video games have at least 8 GB of RAM [Sof21]. DRAM is significantly faster than secondary memory, but does not persist beyond termination of a process or rebooting.

Registers and Cache memory are memory units owned by and part of the CPU. They are generally implemented using Static RAM (SRAM)[Dre07], which is a form of RAM that is significantly faster than DRAM. Unfortunately it is also significantly more expensive to produce, which makes it cost-prohibitive to utilize as primary memory. Modern consumer CPUs have multiple cache layers, and together with register memory they form their own memory hierarchy.

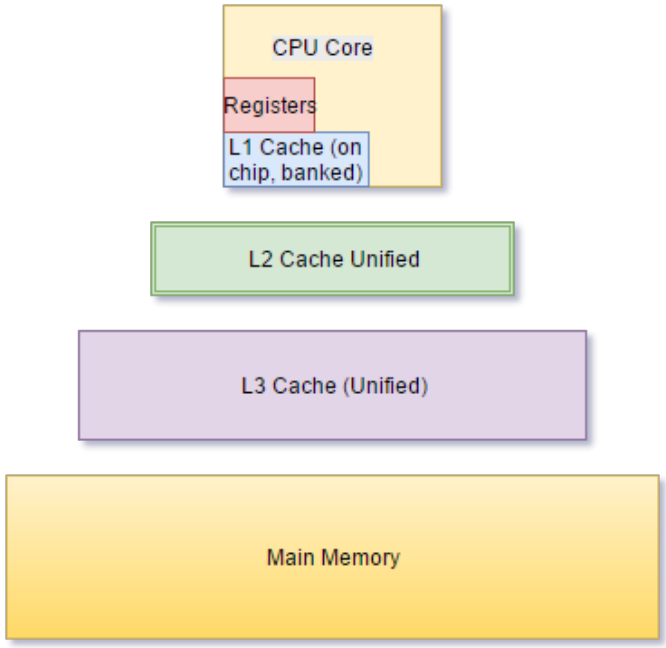

Figure 2.9: Cache hierarchy of some modern consumer CPUs.

Figure 2.9 shows a cache hierarchy common to many modern consumer CPUs. Notice that there are three cache levels: L1, L2, and L3. Each cache from L1 to L3 is bigger and slower than the last, and generally contains a superset of the data contained in the smaller cache (per the inclusion property [PH90]). Furthermore, the L1 cache is often divided into a data cache (L1d), and an instruction cache (L1i) for accessing decoded instructions [Dre07].

While cache sizes may differ between modern consumer CPUs, they are generally quite similar. These are the cache sizes for the 12-core AMD Ryzen 3900x CPU used in this thesis:

1. L1i: 12 x 32KB
2. L1d: 12 x 32KB
3. L2: 12 x 512KB
4. L3: 4 x 16MB (Shared between cores)

As a consequence of how reads from RAM function, cache data is stored in "lines" of (usually) 64 bytes, so that each cache read retrieves a contiguous 64

byte memory region. As a consequence of this, storing sequentually read data in a contiguous manner is highly important. Ensuring that data is aligned on a 64-byte boundary is also desirable.

2.4 Related Works

2.4.1 Material Point Method

The Material Point Method (MPM) has its origins in the Particle-in-Cell (PIC) method. PIC is a hybrid Lagrangian-Eulerian method for solving certain classes of partial differential equations. The method was developed at Los Alamos Scientific Laboratory during the 1950s by F. Harlow et al. [HER55], and saw early use for problems in computational fluid dynamics [Har63].

In 1986, Brackbill and Ruppel [BR86] developed the Fluid-Implicit Particle method (FLIP), which extends the PIC method with a new transfer scheme designed to counter dissipation issues. Whereas PIC transfers the computed *quantities* from grid to particle, FLIP transfers the *changes in quantity*. While this results in significantly less diffusion, the method is far more unstable and tends to develop noise. Therefore, PIC and FLIP transfer schemes are often combined to yield the best results[Bri08]. In computer graphics, the method was used by Zhu and Bridson [ZB05] to simulate sand acting as a fluid.

In 1994-95, D. Sulsky et al. [SCS94] [SZS95] extended the FLIP method to solid and general continuum mechanics. This new approach was dubbed the Material Point Method, where the term *material point* refers to Lagrangian particles in the material view of a continuum body. As a "meshless" method, MPM compares favourably to the popular FEM method in some aspects. Consequently, it has found application in the solution of various problems in solid mechanics.

The initial MPM method used piecewise linear C^0 continuous functions. This could be troublesome, as particles crossing differential discontinuities could cause numerical issues, noise, or even physically impossible values, like negative mass (see appendix B). The Generalized Interpolation Material Point Method (GIMP), which was proposed in 2004 [BK04], is a generalization of MPM that allows for C^1 continuity using C^0 piecewise continuous basis functions. Another popular alternative is the use of quadratic B-splines, which are already in C^1[SKB08].

In 2013, A. Stomakhin et al. [Sto+13] introduced MPM to computer graphics,

using it to simulate snow for use in the animated movie Frozen by Disney. Following this, the method has seen widespread interest from computer graphics researchers. Several extensions and improvements to MPM have been developed for its use in CG, and the method has been successfully applied to simulate a wide variety of materials.

An improved transfer scheme for PIC methods, the Affine Particle-in-Cell method, was developed by C. Jiang et al. in 2015 [Jia+15]. The method significantly improves both the dissipation issues of standard PIC transfers and the noise issues of FLIP, while also maintaining conservation of angular momentum. The APIC transfer scheme has subsequently become a highly popular choice for MPM. As a further improvement, PolyPIC[Fu+17] is capable of providing lossless particle-grid transfers by generalizing APIC.

In 2018 Y. Hu et al. developed a Moving Least Squares version of MPM[Hu+18a]. This method foregoes computation of B-spline kernel gradients and reuses the affine momentum matrix of APIC. MLS-MPM is both faster and simpler to implement than standard MPM.

2.4.2 Real-Time Physical Simulation

Within the context of computer graphics, multiple methods exist for simulating physical bodies in real time. For simulating fluids, several approaches exist. Jos Stam's "Stable Fluids" [Sta01] uses an Eulerian discretization of Navier Stokes to simulate fluids flows in real time. Despite being comparatively old (published in 1999), the method is still very popular for simulating phenomena such as smoke and fire. Among others, the method is utilized by EmberGen[Jan21], a popular commercial solution for real-time simulation of fluids.

Another method frequently used for real time fluid simulation is Smoothed Particle Hydrodynamics[Mon92]. SPH is a fully Lagrangian solver, whose distinguishing feature is the advection of particles via neighborhood kernels, wherein particles are updated by examining the values of adjacent particles. By utilizing GPU computation and acceleration of nearest neighbor lookup, SPH methods can simulate fluids with particle counts $> 100\ 000$ at real-time rates [Hoe13].

Real-time simulation of solids, sometimes known as soft-body simulation in computer graphics, is also widespread in the literature. A popular approach is the mass-spring model, where a body is modeled as a series of point masses that interact with each other through imagined springs, which follow Hooke's Law.

Another method is Position Based Dynamics (PBD), pioneered by Müller et al. [Mül+05]. PBD is able to effectively simulate various materials such as cloth and hyperelastics, but is less physically motivated compared to other methods.

Methods also exist for coupled interactions between solids and fluids. Stable Fluids and SPH are limited to the simulation of fluids, and the mass-spring model is not a comfortable fit for fluid simulation. However, Müller et al. have proposed a method which encapsulates PBD in a unified framework that allows for multi-phase coupling[Mac+14].

As part of a differentiable physics simulator, Y. Hu et al. [Hu+18b] developed a real-time GPU implementation of MPM for the purpose of soft robotics. Their method is adapted from [Gao+18] and is capable of simulating hyperelastic bodies consisting of more than 100 000 particles in real-time. However, their implementation is not CPU-based, nor is it focused on computer graphics applications.

Grant Kot, an independent developer, is currently developing a game engine featuring a real-time CPU-based MPM implementation[Kot21]. His method is capable of simulating fluids and soft hyperelastics with very high ($> 500\ 000$) particle counts in real-time on modern hardware.

Chapter 3

Continuum Mechanics

3.1 Continuum Assumption

The basis of continuum mechanics is the assumption that the quantities of the material we are modeling can be thought of as continuous. In other words, *we assume that quantities like velocity, mass/density, and temperature are well-defined even at infinitesimal sizes.* This is the **continuum assumption**, and it allows us to model the material as a continuum. In reality, of course, the body we are modeling is comprised of molecules of non-infinitesimal size. However, the assumption should not lead to inaccuracies at the macroscopic level[GS08].

3.2 Continuum Body

In continuum mechanics, we define bodies in a particular manner. We define a material body as an open subset B of Euclidean Space E^d. In other words, *the body is defined by the space it occupies.* B is called a placement or **configuration** of the body. The reasons for representing bodies in this way are purely mathematical[GS08].

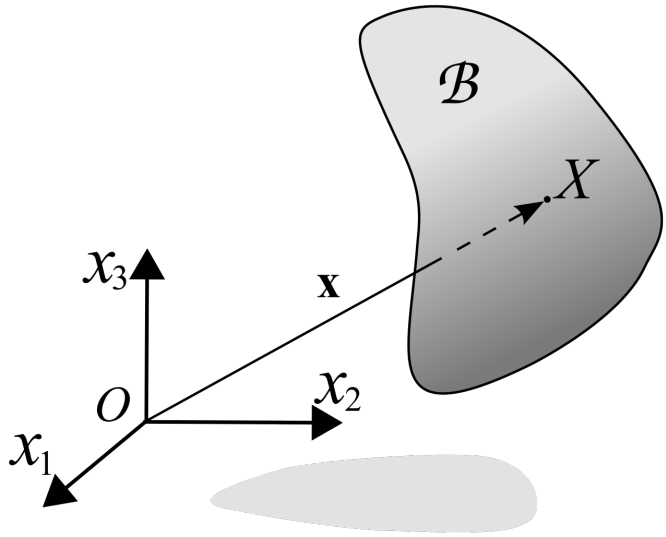

Figure 3.1: A continuum body is a region of space B.

3.3 Deformed configurations and the deformation map

A body can be displaced and deformed. To capture this, we need some way to describe change from the initial body B.

We define a *deformed configuration* B' as a change in configuration from some *reference configuration* B. Points in B are referred to with X, and are called *material/reference coordinates*. Points in B' are referred to with x, and are called *spatial/world coordinates*.

We define the **deformation map** as the function $\phi : B \to B'$ which maps each point X in B to a point $x = \phi(X)$ in B'. Assuming B' is the configuration at some t, $x = \phi(X, t)$. We usually say that $x = x(X, t) = \phi(X, t)$, so x and ϕ can be used interchangeably[GS08].

3.4 Lagrangian versus Eulerian for Continuum Bodies

Recall from 2.2 that Eulerian fields are specified in terms of space and time, e.g. $u(x,t)$ for velocity. Lagrangian fields, on the other hand, are specified in terms of the initial position of the particles we are tracking, e.g. $v(a,t)$, where a is the position of the particle at some initial reference configuration. For continuum bodies, this translates to specifying a body in terms of its *spatial coordinates* x or its *material coordinates* X. In other words, the Eulerian description is in terms of the deformed body B', whereas the Lagrangian description is in terms of the undeformed body B. This distinction is significant, as the methods for calculating the quantities we need to solve for the next timestep of our body (stress, strain) are different depending on the frame of reference we take.

MPM tracks the current state of the body through quantities on the particles, and the grid is reset at the start of each iteration. This means our tracked quantities are specified in terms of change from the undeformed body.

3.5 Push-Forward and Pull-Back

The Lagrangian description is in terms of the undeformed configuration B, while the Eulerian description is in terms of the deformed configuration B'. Functions such as velocity and acceleration are therefore defined separately, either for material coordinates X or spatial coordinates x. However, as explained in 3.3, the deformation map $x = \phi(X,t)$ yields x from X. It can also be shown [Jia+16] that the inverse function ϕ^{-1} exists.

Therefore, functions defined over Eulerian quantities can be transformed to be defined over Lagrangian quantities, and vice versa. This is known as *push-forward* (Lagrangian to Eulerian) or *pull-back* (Eulerian to Lagrangian). For example, the push-forward and pull-back of velocity are:

$$v(x,t) = V(\phi^{-1}(x,t),t)$$
$$V(X,t) = v(\phi(X,t),t)$$

3.6 Strain

Strain is a general measure of deformation. Like stress (see appendix A), there are many ways to measure strain.

Consider a sphere Ω of radius $a > 0$ centered at some point X_0 in B. The deformed sphere is $\Omega' = \phi(\Omega)$. Strain is a measure of the *relative difference in shape* between Ω' and Ω when the sphere radius $a \to 0$. How we measure that difference depends on what measure of strain we use. In our case, we base our measure on the *deformation gradient*.

3.7 Deformation gradient

The deformation gradient F is a second order tensor describing the local behavior of a deformation ϕ. Formally, it can be defined as $F(X) = \nabla\phi(X)$. That is, the Jacobian of the deformation map for material coordinate X.

In practice, it is easier to understand with an example. Consider the deformation map of a constant translation:

$$x(X,t) = \phi(X,t) = X + tvn$$

Where t is time, v is velocity and n is direction. In this case, the deformation gradient is:

$$F(X) = \nabla(X + tvn) = I$$

Where I is the identity matrix. If we have a rotation R, the deformation gradient would be:

$$F(X) = \nabla(RX) = R$$

In MPM, we track the deformation gradient F as a matrix on every particle p. This is in turn used to find the forces acting on the particles, which we find by relating the deformation gradient to the stress.

In addition, the determinant of the deformation gradient, $det(F) = J$, characterizes infinitesimal volume change[Jia+16]. If $J > 1$, then the volume has increased, and if $J < 1$, the volume has decreased. This can be used to approximate the current volume of a particle from its initial volume:

$$V_p^n \approx V_p^0 J \tag{3.1}$$

3.8 Cauchy-Green Strain Tensor

Notice that in the case where the body is standing still or being translated, the deformation gradient is equal to the identity matrix. This implies no deformation. However, when rotating the body, we get $F = R$, even though the rotation does not deform the body. We would like to have some measure of deformation that excludes rotations. This is the Cauchy-Green strain tensor C.

The Cauchy-Green Strain Tensor is simply defined as:

$$C = F^T F \tag{3.2}$$

And is equivalent to F with the rotational element removed[Jia+16].

The Cauchy-Green strain tensor has three commonly used tensor *invariants*, which are often used in a strain energy density function. A tensor invariant is a property of the tensor that does not change with a rotation of the coordinate system. The three main invariants of C are:

$$I_1 = tr(C) \tag{3.3}$$
$$I_2 = tr(CC) \tag{3.4}$$
$$I_3 = det(C) = J^2 \tag{3.5}$$

Where tr is the *trace*, the sum of diagonal entries of the tensor.

3.9 Strain Energy

Energy is a broad and general concept. The most well-known forms of energy are kinetic energy (related to the motion of a body) and potential energy ("stored" in the object). *Strain energy* is a form of energy stored in a body undergoing elastic deformation. The most common example is that of a spring. When the spring is stretched or compressed, it stores strain energy proportional to the deformation. When the stretching/compressing force is removed, the energy is "spent" to return the spring to its initial configuration.

3.10 Strain Energy Density Function (SEDF)

A Strain Energy Density Function (SEDF) is a scalar function $\Psi(F)$ that relates the strain energy density (strain energy per volume) of a material to its strain measure.

Hyperelastic solids are defined as ideally elastic solids where the derivative of the SEDF yields the *stress* of the material. This allows us to determine the constitutive stress-strain relation and obtain the stress of a particle by finding the derivative of the SEDF for a given F. The SEDF is given by the hyperelastic model we use to model our body.

3.11 Neo-Hookean Constutive Model

The purpose of the constitutive model of a material is to relate the stimuli (e.g. deformations) to the material responses (e.g. force, stress, energy) they trigger[SB12].

The Neo-Hookean hyperelastic model is one of the earliest and most common hyperelastic models. Its SEDF is:

$$\Psi = \frac{\mu}{2}(I_1 - d) - \mu \cdot log(J) + \frac{\lambda}{2} log^2(J) \tag{3.6}$$

Where μ and λ are material parameters (see 3.16) and d is the spatial dimension of our simulation. Note that $\Psi = 0$ if our object has only undergone translation and/or rotation. This is because $I_1 = tr(C) = tr(I) = d$ when we have no deformation, and $J = det(F) = 1$ when we have no change of volume.

3.12 First Piola Kirchoff and Cauchy Stress

Just as with strain, there are various measures for stress. Two of the most common are *First Piola Kirchoff* (PK1) and *Cauchy Stress*. The difference between the two is what frame of reference they are defined in terms of. PK1 stress is defined in terms of the *undeformed* body, i.e. the material space. Cauchy stress is defined in terms of the *deformed* body. We will be calculating stress from the Lagrangian particles, which are defined in material space. Furthermore, PK1 stress is easily found for hyperelastic solids.

For a hyperelastic solid, PK1 stress is defined as:

$$P = \frac{\partial \Psi}{\partial F}$$

Where Ψ is the SEDF and F is the deformation gradient. For a Neo-Hookean model, this is equivalent to:

$$P = \mu(F - F^{-T}) + \lambda \cdot log(J) \cdot F^{-T} \tag{3.7}$$

Note that we can easily find the Cauchy stress from the PK1 stress with:

$$\sigma = \frac{1}{J} \cdot P \cdot F^T \tag{3.8}$$

In MPM, we calculate the stress forces as part of the P2G momentum transfer.

3.13 Balance Laws

Let us, for a moment, treat our material body as a discrete collection of particles. We can then state rules about how we expect these particles to behave. For a given particle i, we expect:

$$\dot{m}_i = 0 \tag{3.9}$$

That is, the mass of the particle should remain constant. Further, we can state Newton's Second Law w.r.t. these particles:

$$m_i \ddot{x}_i = f_i^{env} + \sum_{\substack{j=1 \\ j \neq i}}^{N} f_{ij}^{int} \tag{3.10}$$

In other words, the sum of forces on each particle is equal to its mass times acceleration. As $m\ddot{x} = m\dot{v}$, it can also be thought of as describing the rate of change of linear momentum.

If we postulate that these rules also apply under the continuum assumption (effectively having an infinite number of infinitesimal particles), then we can define the essential **balance laws** that help determine the behavior of the system.

We are mainly interested in **Conservation of mass** and **Conservation of linear momentum**. In the Lagrangian frame, these are:

$$R(X,t)J(X,t) = R(X,0) \tag{3.11}$$

$$R(X,0)A(X,t) = \nabla^x \cdot P + R(X,0)b_m \tag{3.12}$$

Where $R(X,t)$ is the mass density ρ in the Lagrangian description, $A(X,t)$ is acceleration, and b_m are the body forces (usually just gravity g).

Conservation of mass (3.11) states that the mass density of a material point should not change with deformation. In the Lagrangian frame each particle has a fixed mass by definition, so conservation of mass is automatically satisfied[SZS95].

The balance law for linear momentum (3.12) is derived from:

$$\frac{d}{dt} l[\Omega_t] = r[\Omega_t] \tag{3.13}$$

Which is analogous to 3.10 for continuum bodies. $l[\Omega_t])$ is the linear momentum of an open subset Ω_t of B, and $r[\Omega_t]$ is the external force on Ω_t. In other words, the rate of change of linear momentum is equivalent to the external forces on Ω_t. The term $\nabla^x \cdot P$ (divergence of PK1 stress w.r.t. to space) describes surface forces, and follows from the Divergence theorem (see [GS08] for a more thorough derivation).

3.14 Weak form of conservation of momentum

Applying the Divergence theorem in order to define 3.12 requires a continuously differentiable stress field[Sch14]. However, in our numerical solution, finding the spatial divergence of the stress tensor is not feasible. In order to find the $\nabla^x \cdot P$ term, we instead solve the *weak form* of 3.12, using the process outlined in 2.1.2.

The weak forms of conservation of momentum are given following the derivation process in [Jia+16]. The forms are written using tensor notation, with which 3.12 is written as $R_0 A_i = P_{ij,j}$ (with body forces omitted). An introduction to tensor calculus can be found in [GS08].

For a given test function $Q : \Omega \to R^d$, the Lagrangian weak form is:

$$\int_{\Omega^0} Q_i(X,t) R(X,0) A_i(X,t) dX = \int_{\partial\Omega^0} Q_i T_i ds(X) - \int_{\Omega^0} Q_{i,j} P_{ij} dX \tag{3.14}$$

Where T is the boundary traction field (see appendix A), ds(X) denotes the infinitesimal surface. Body forces have been omitted for simplicity. In the method described in chapter 4 the body forces are uniform over the body, and so they are added during the grid update stage.

If we push-forward (see 3.5) Q and T to q and t, and use equation 3.8 to find the Cauchy stress, we can formulate the Eulerian weak form:

$$\int_{\Omega} q_i(x,t) \rho(x,t) a_i(x,t) dx = \int_{\partial\Omega^t} q_i t_i ds(x) - \int_{\Omega^t} q_{i,k} \sigma_{ik} dx \tag{3.15}$$

3.15 Young's Modulus and Poisson's Ratio

Young's modulus is a fundamental way of determining the stress-strain relation of a material. It is simply defined as the relation stress over strain[Jas76]:

$$E = \frac{\sigma}{\epsilon} \tag{3.16}$$

It is common to examine the behavior of a material by charting its Young's modulus. If the relation can be charted as a straight line, the material is said to be

linearly elastic. Linearly elastic models work well for materials such as wood and metal, where the deformation is relatively small.

Hyperelastic materials, which derive their stress-strain relation from the derivative of the SEDF, have a nonlinear modulus. These materials are better suited for materials with greater deformation, e.g. rubber-like materials.

Poisson's Ratio relates longitudinal deformation to lateral deformation. Consider, for example, an elastic cube. If the cube is compressed inwards along the x-axis, we expect the cube to expand along the y and z axis. How much expansion actually occurs is given by Poisson's Ratio. Therefore, it is defined as:

$$\nu = \frac{-\epsilon_{lateral}}{\epsilon_{longitudinal}} \tag{3.17}$$

3.16 Lamé Coefficients

With Young's modulus E and Poisson's ratio ν, we can define the **Lamé Coefficients**[GS08]. Combined with a constitutive model, these coefficients can be used to determine the stress forces affecting an object, given its strain (in our case via the deformation gradient).

$$\mu = \frac{E}{2(1+\nu)} \qquad \lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} \tag{3.18}$$

3.17 Additional Constitutive Models

The Neo-Hookean model described in 3.11 is one of the most popular models for elastic solids. However, multiple other models also exist. This section describes some other common material models for solids.

3.17.1 Linear

While the Green Strain Tensor is effective at removing the inaccurate deformation caused by rotation, it has the undesirable property of being a quadratic function

of deformation (due to $F^T F$). Because of this, material models that use this strain tensor have nontrivial computational costs.

A strain measure that alleviates these issues is the *small strain tensor*. This tensor is given as a linear approximation of the Green strain tensor based on a Taylor expansion[SB12]. This approximation leads to the following strain tensor:

$$\epsilon = \frac{1}{2}(F + F^T) - I \tag{3.19}$$

This strain tensor gives rise to the Linear elasticity model, which is the simplest practical model in terms of computational complexity.

The SEDF for the linear elasticity model is given as:

$$\Psi(F) = \mu\epsilon : \epsilon + \frac{\lambda}{2} tr^2(\epsilon) \tag{3.20}$$

Where : is the double dot product, and μ and λ are Lamé coefficients (see 3.16).

Based on this, the PK1 stress is found to be:

$$P(F) = \mu(F + F^T - 2I) + \lambda tr(F - I)I \tag{3.21}$$

While the linear elasticity model is inexpensive, it is only considered to be an accurate measure of deformation for *small motions*. This is highly disadvantageous for real-time CG applications, as we are mainly interested in visually rich large deformation scenes.

3.17.2 St. Venant-Kirchoff

As the linear model is ill-suited for most scenes of interest in CG, methods which utilize the Cauchy-Green strain tensor are generally favored. Besides the Neo-Hookean, another popular choice is the *St. Venant-Kirchoff* material model. This model uses the same SEDF as the linear elasticity model, but with C instead of ϵ:

$$\Psi(F) = \mu C : C + \frac{\lambda}{2} tr^2(C) \tag{3.22}$$

With PK1 being:

$$P(F) = F[2\mu C + \lambda tr(C)I] \tag{3.23}$$

As this model utilizes C, it has the important benefit of being rotationally invariant. While this makes it more expensive, it also makes it better suited for the large deformation scenes common in CG.

Unfortunately, the model suffers from problems with large compressions. As the body is compressed, it reacts with a restorative force proportional to the compression. However, once a critical threshhold is reached - $\approx 58\%$ compression from the undeformed state along a given axis - the resisting force actually begins to *decrease*. As compression approaches 100%, the resistance tends to 0. If it goes beyond 100%, the resisting force will attempt to push the body towards complete material inversion.[SB12]

In practice, this makes the St. Venant-Kirchoff model ill-suited to scenes that feature significant compression.

3.17.3 Corotated Constitutive Model

The *Corotated Linear Elasticity* model attempts to address the issues with both the linear elasticity model and the St. Venant-Kirchoff model. It does this by utilizing the polar decomposition of the deformation gradient $F = RS$, where R is rotation and S is scaling. Using the polar decomposition, a new strain measure is given as $\epsilon_c = S - I$. Note that this measure is rotationally invariant as we have decomposed and removed the rotation term R.

The SEDF corotated linear elasticity model is the same as 3.20, with the updated strain measure:

$$\Psi(F) = \mu\epsilon_c : \epsilon_c + \frac{\lambda}{2}tr^2(\epsilon_c) = \mu||S - I||_F^2 + (\lambda/2)tr^2(S - I) \tag{3.24}$$

The PK1 stress can be shown to be[SB12]:

$$P(F) = 2\mu(F - R) + \lambda tr(R^T F - I)R \tag{3.25}$$

In the computer graphics literature, it is common to apply a slight modification to the SEDF of the linear elasticity[Jia+16]. This yields the *Fixed Corotated*

Constitutive Model. The SEDF of this model, along with its PK1 stress derivation, is given in [Jia+16]. The PK1 stress is found to be:

$$P(F) = 2\mu(F - R) + \lambda(J - I)JF^{-T} \tag{3.26}$$

Chapter 4

The MPM Algorithm

In this chapter, the stages of the basic MPM algorithm are described for a hyperelastic Neo-Hookean solid using symplectic time integration as detailed in [Sto+13] and [Jia+16]. Note that the described method does not use the affine momentum matrix of APIC, nor the fused and simplified computations of MLS-MPM. These are explained in section 5.1. Furthermore, although MPM is based on a discretization of the weak form of momentum (given in 3.14), an explanation of this discretization has been purposefully omitted due to its length and complexity. A rigorous explanation of the discretization of equations 3.14 and 3.15 is given in [Jia+16].

4.1 Overview

The Material Point Method is a hybrid Lagrangian-Eulerian method. The simulated body is represented both as a grid and as a set of particles. The quantities that make up the body (position, velocity, deformation gradient etc.) are tracked solely on the particles. The grid is used as a "scratch-pad" grid, meaning its values are reset at the beginning of each step.

The basic MPM algorithm has three steps:

1. Particle-to-Grid (P2G)
2. Grid Update

3. Grid-to-Particle (G2P)

During the **P2G** stage, we rasterize the mass and velocity values of the particles to the grid. In order to do this, we calculate the influence each particle has on its neighboring grid cells (usually the 3^d neighbourhood). As part of this step we also compute and rasterize the stress induced forces, and add them into the updated velocity.

In the **Grid Update** stage, external forces and boundary conditions are applied. In the simplest case the external forces are only gravity, and the boundary conditions simply set the velocity to zero at the edges.

Finally, in the **G2P** stage, the particle values are updated. Each particle updates its velocity based on the values of the neighboring grid cells, and particle positions are subsequently updated as well. The deformation gradient is also updated.

For different materials, the key differentiating factor is the stress computation of the P2G step. Material behavior can be vastly different depending on how stress is computed for a given particle. In addition, since all other steps are the same, independent of stress computation, simulation of bodies with different constitutive models is extremely easy. This is one of the key benefits of MPM over other methods, as it allows easy coupling between bodies that have different material models, or even different phases.

The rest of this chapter explains the method in detail. In addition to a more thorough explanation of the three main stages, the choice of basis function is also discussed.

4.2 Basis functions

Basis functions are used to interpolate values between particles and the grid. Restricted on a single element, they are called *shape functions*, and are denoted N_i. For example, the lumped mass of a particle p is given by:

$$m_p^n = \sum m_i^n N_i(x_p) \tag{4.1}$$

Where $N_i(x_p)$ is the basis function of grid node i for particle p. As this acts as a weight for P2G/G2P transfers, the shorthand $N_i(x_p) = w_{ip}$ is often used.

Traditional MPM as proposed by D. Sulsky et al.[SCS94] uses linear Lagrangian polynomials as basis functions. While linear Lagrangian functions are fast and easy to implement, they have several numerical issues. In particular, they are susceptible to *cell crossing* issues[BK04] (see appendix B). If particles of an element are uniformly stressed, particles crossing discontinuous points of the basis function will cause inaccurate force imbalance. Quadratic Lagrange basis functions have similar issues, are discontinuous at element boundaries, and can even take on negative values[Tie+17].

Quadratic B-spline (basis spline) functions are a popular alternative to traditional MPM basis functions. Splines are defined by piecewise polynomials, i.e. polynomials defined across subdomains of some interval. B-splines are constitutive polynomials of a spline that take on the value of 0 outside of their subdomain. Quadratic splines are continuously differentiable, both at joining points and points of merging.

Both Stomakhin et al.[Sto+13] and C. Jiang et al.[Jia+16] use dyadic products of one-dimensional B-splines, as described in [SKB08]:

$$N_i(x_p) = N(\frac{1}{\Delta x}(x_p - x_i))N(\frac{1}{\Delta x}(y_p - y_i))N(\frac{1}{\Delta x}(z_p - z_i)) \tag{4.2}$$

Where $x_p = (x_p, y_p, z_p)$ is the evaluation position, and N is the kernel used. [Sto+13] uses a cubic kernel, while [Jia+16] describes both cubic and quadratic kernels. The quadratic kernel has better computational efficiency and lower memory requirements, while the cubic kernel is more accurate. The quadratic kernel is given as:

$$N(x) = \begin{cases} \frac{3}{4} - |x|^2, & 0 \leq |x| < \frac{1}{2} \\ \frac{1}{2}(\frac{3}{2} - |x|)^2, & \frac{1}{2} \leq |x| < \frac{3}{2} \\ 0, & \frac{3}{2} \leq |x| \end{cases} \tag{4.3}$$

As the weak formulation requires finding the gradient of the test function, the gradient of the basis function $\nabla N_i(x) = \nabla w_i p$ is also required. The differentiation is given in [Jia+16].

4.3 Volume Estimation

In the Lagrangian representation, the body is made up of infinitesimal material points. As explained in 3.7, the current particle volume V_p^n can be approximated using the initial volume and the determinant of the deformation gradient:

$$V_p^n \approx V_p^0 J \tag{4.4}$$

However, we still need to determine the initial volume. [Sto+13] does this using a preprocessing step. First, the cell densities are estimated as $m_i^0/\Delta x^d$, where Δx^d is the cell width. Then, the particle density are estimated from the grid cells with $\rho_p^0 = \sum_i m_i^0 w_{ip}^0/\Delta x^d$. Finally, the initial particle volume is set to be $V_p^0 = m_p/\rho_p^0$.

4.4 Particle To Grid

In the particle-to-grid stage, we transfer mass and velocity from the particles to the grid. Mass is transferred as:

$$m_i^n = \sum_p m_p w_{ip}^n \tag{4.5}$$

Where w_{ip} is the weight determining the contribution of particle p to grid cell i.

To transfer velocity while also maintaining conservation of linear momentum, we transfer momentum and divide by the transferred weight, giving velocity:

$$v_i^n = \frac{1}{m_i^n} \sum_p v_p^n m_p w_{ip}^n \tag{4.6}$$

As part of the momentum transfer, we also transfer the stress based forces affecting the body. Recall that a hyperelastic body is one where the PK1 stress is equivalent to the derivative of its strain energy density function. The discretized total strain energy is:

$$e = \sum_p V_p^0 \Psi(F_p) \tag{4.7}$$

Where V_p^0 is the initial volume of particle p, and F_p is the deformation gradient.

The elastic force of each node can then be estimated by taking the negative gradient of e[Jia+16]. This gives a discretized nodal stress force of:

$$f_i^n = -\sum_p V_p^0 P_p^n F_p^{nT} \nabla w_{ip}^n \tag{4.8}$$

Which can also be written using the Cauchy stress tensor σ:

$$f_i^n = -\sum_p V_p^n \sigma \nabla w_{ip}^n \tag{4.9}$$

The change in velocity as a result of stress forces can be said to be:

$$\Delta v_i^{n+1} = \frac{\Delta t}{m_i^n} \sum_p V_p^n \sigma \nabla w_{ip}^n \tag{4.10}$$

Which follows from conservation of linear momentum, or analogously $F = m\dot{v}$. Since $v_i^{n+1} = v_i^n + \Delta v_i^{n+1}$, the full transfer of velocity from particle to grid is:

$$v_i^{n+1} = \frac{1}{m_i^n} \sum_p (v_p^n m_p w_{ip}^n + \Delta t V_p^n \sigma \nabla w_{ip}^n) \tag{4.11}$$

Note: The final division $\frac{1}{m_i^n}$ is not done as part of the P2G step, as it requires m_i^n, which we are in the process of computing. Therefore, we temporarily store the momentum $m_i^n v_i^{n+1}$ in the velocity field of the grid cells.

4.5 Grid Update

In the grid update stage, we first find v_i^{n+1} by diving the momentum $m_i^n v_i^{n+1}$, currently stored in the velocity field of the cell, by m_i^n. Afterwards, we enforce boundary conditions and apply body forces, usually just gravity. Boundary conditions can be enforced both for the surface of the body and for the simulation domain itself. We commonly enforce a homogeneous Dirichlet boundary condition of 0 for the velocity across the simulation domain, ensuring that the body does not move out of bounds.

This update scheme assumes we have no collision objects. If we do, we also have to take these into account as part of the velocity update.

4.6 Grid To Particle

There are three main parts to the grid-to-particle stage:

- Transfer updated velocity back to particles
- Advect particles
- Compute updated deformation gradient

We transfer velocity with the same weights we used in the P2G step:

$$v_p = \sum_i v_i w_{ip} \tag{4.12}$$

Afterwards, we perform advection. As the particles are free-moving, this is simple:

$$x_p^{n+1} = x_p^n + \Delta t v_p \tag{4.13}$$

Finally, we compute the updated deformation gradient.

$$F_p^{n+1} = I + \sum_i (x_i - x_i^n)(\nabla w_{ip}^n)^T F_p^n \tag{4.14}$$

The derivation of the deformation gradient update is given in [Jia+16].

Chapter 5

Implementing Real Time MPM

MPM is both a powerful and a versatile method for simulating physical bodies. However, this power comes at a cost. Recent improvements within Computer Graphics research on MPM have seen simulation times drop low enough to occasionally be measured in seconds per frame, but this is still a far cry from real time. In order to achieve a framerate of about 30 frames per second, often considered a threshold for real time, a single frame should take no longer than 33 milliseconds to simulate and render. In practice, however, real-time applications such as video games may impose additional overhead from other computations, shrinking the practical time-budget further.

With this in mind, multiple questions are important when aiming to implement MPM in real time:

- What concessions have to be made in terms of the visual quality, stability, and accuracy of the simulation?
- How can the simulation best utilize the resources available for its application?
- What scenes and materials are best suited for real-time?
- What optimizations can be made to improve performance?

This chapter is divided into five parts. Part one describes extensions to the basic method described in chapter 4. These extensions either improve performance, or significantly improve visuals without incurring meaningful performance overhead. Part two deals with the issue of time integration, explicit vs implicit methods, and

temporal adaptivity. Part three talks about implementation details surrounding memory utilization and parallelization techniques. Part four discusses methods for rendering the simulated bodies. Finally, a brief description of the final implemented method is given.

5.1 Extensions

5.1.1 Affine Particle-In-Cell (APIC)

The Affine Particle-In-Cell method (APIC) is an improved transfer scheme for PIC-style hybrid Lagrangian-Eulerian methods. It seeks to improve the dissipation issues of PIC transfers, while also avoiding the noise issues of FLIP transfers. In the APIC paper by C. Jiang et al. [Jia+15], the authors start out by attempting to develop a method that preserves angular momentum without the noise of FLIP. To do this, they note that the angular momentum lost for a particle in the grid-to-particle stage is:

$$L_p^{n+1} = \sum_i w_{ip}^n (x_i - x_p^n) \times m_p v_i^{n+1}$$

To alleviate the issue of unpreserved angular momentum, this value can be computed and stored for each particle. The particle-to-grid step would then be modified to incorporate this sample into the velocity transfer.

However, this would still damp out non-rigid motions such as shearing. Therefore, the authors propose a method that will avoid both types of damping. Instead of tracking L_p^n, they store a locally affine velocity matrix C_p^n at each particle. Using this, the local velocity contribution of a particle at grid node i becomes $v_p^n + C_p^n(x_i - x_p^n)$, and the P2G velocity/momentum transfer becomes:

$$m_i^n v_i^n = \sum_p w_{ip}^n m_p (v_p^n + B_p^n (D_p^n)^{-1} (x_i - x_p^n)) \tag{5.1}$$

Where $C_p^n = B_p^n (D_p^n)^{-1}$.

D_p^n is analogous to an inertia tensor. It is given by:

$$D_p^n = \sum_p w_{ip}^n (x_i - x_p^n)(x_i - x_p^n)^T$$

B_p^n is the sample matrix tracked at each particle, and is given by:

$$B_p^{n+1} = \sum_i w_{ip}^n v_i^{n+1} (x_i - x_p^n)^T \tag{5.2}$$

Note that D_p^n is not strictly speaking an inertia tensor. In fact, for quadratic B-spline interpolation, it takes on the simple form of $D_p^n = \frac{1}{4}\Delta x^2 I$, which amounts to a constant scaling factor.

The APIC transfer scheme provides significant improvements to the physical correctness of MPM, and has become the transfer scheme of choice for many recent MPM papers in computer graphics. As explained in 5.1.2, the method works well with MLS-MPM.

5.1.2 Moving least Squares MPM (MLS-MPM)

In [Hu+18a], Y. Hu et al. develop a *Moving Least Squares* Material Point Method. The authors show that MLS-MPM can be up to twice as fast as standard MPM, while also being easier to implement without any degradation in accuracy. The main difference between MPM and MLS-MPM is the choice of function space for the basis functions. While standard MPM in computer graphics generally uses B-spline basis functions (as explained in 4.2), MLS-MPM uses MLS shape functions.

MLS is a method that is used to approximate a continuous function from a set of scattered data samples. It extends from weighted least squares (WLS), which extends from regular least squares (LS).

The principle of LS is fairly straightforward: Given a set of N points located at x_i in R^d where $i \in [1...N]$, we wish to approximate some $f(x)$ where the values $f(x_i) = f_i$ are given. This is done through least squares minimization against the set of polynomial functions $f \in \Pi_m^d$, where Π_m^d is the space of monomial polynomials of total degree at most m in d spatial dimensions:

$$\min_{f \in \Pi_m^d} \sum_i ||f(x_i) - f_i)||^2 \tag{5.3}$$

The functions $f \in \Pi_m^d$ can be written as:

$$f(x) = b(x)^T c = b(x) \cdot c \tag{5.4}$$

Where $b(x)$ is the polynomial basis vector, e.g. $b(x) = [1, x, y, x^2, xy, y^2]$ for $m = 2$ and $d = 2$, and $c(x)$ is the vector of unknown coefficients.

The underlying idea behind WLS is the observation that we may not want to consider the contribution of each sample equally in our approximation. Therefore we weigh sample contribution by distance to a fixed point, which leads to a different error functional from 5.3:

$$\min_{f \in \Pi_m^d} \sum_i \theta(||\bar{x} - x_i||)||f(x_i) - f_i||^2 \tag{5.5}$$

Where $\bar{x} \in R^d$ is a fixed point, and θ is a weighting function which weighs samples by their distance to $\bar{x}$. Many weighting functions have been proposed, such as the Gaussian and the Wendland function[Nea04].

MLS extends WLS to be defined over the entire domain. The method starts out with an arbitrarily placed point in R^d, which is then *moved* over the entire parameter domain, computing a WLS fit for each point individually. The locally defined functions are combined to yield a global function, defined over the domain:

$$f(x) = f_x(x), \min_{f \in \Pi_m^d} \sum_i \theta(||x - x_i||)||f(x_i) - f_i||^2 \tag{5.6}$$

Which is continuously differentiable if and only if the weighting function θ is continuously differentiable.

In MLS-MPM, the basis functions used for P2G and G2P are replaced with an MLS basis function over the domain. θ is chosen to be a quadratic/cubic B-spline, and the polynomial space is chosen to be linear (i.e. $m = 1$ for the function space Π_m^d).

The authors derive the discretization of the weak form of conservation of momentum with MLS basis functions. They show that the new stress momentum contribution does not require evaluating the gradient of the weighting function, which results in a speedup of the P2G stage.

In addition, the authors show that if a linear polynomial basis is chosen $m = 1$ then the APIC method is equivalent to applying MLS to velocity v(x) with B-splines as the weighting function θ. Therefore, if APIC and MLS-MPM are used together, the G2P stage can be simplified by reusing the computed affine momentum matrix C_p^{n+1} for evaluating the particle velocity gradient. In total, MLS-MPM halves the number of FLOPs needed for each particle.

5.2 Timestep

5.2.1 Stability of explicit and implicit methods

In the earlier discussion of time integration of PDEs (2.1.6), it was stated that explicit methods generally are faster than implicit methods due to ease of optimization. However, explicit methods suffer from a significant drawback due to their lack of stability.

In physical simulation, a system that exhibits stable behavior is not prone to the accumulation and amplification of numerical errors. While implicit methods are generally unconditionally stable, explicit methods are only conditionally stable. This is due to the fact that explicit methods advance the system forwards in time based on the current state of the system. As an elastic material will tend towards its equilibrium, a material which is moved outside its equilibrium state will attempt to return to it, like a pendulum returning to its center. However, just like a pendulum moving towards its equilibrium may carry enough momentum to overshoot the center, the momentum of a material body may cause it to overshoot its equilibrium position.

For implicit methods, this is not a serious concern, as these methods are unconditionally dissipative, meaning energy will never inaccurately be added into the system. However, for explicit methods, whose movement (barring external forces) can be simplified as $\vec{equilibrium} * velocity * \Delta t$, this is a big issue. Taking the example of the pendulum, a large enough timestep may cause it to consistently overshoot its extreme positions. Not only would this introduce more energy into the system, it would also cause a positive feedback loop. For material bodies, this phenomena results in the simulation "blowing up". As the inaccurate physical forces are amplified, the tracked values grow exponentially or take on NaN values. For this reason, explicit methods are limited by a largest permissible timestep. One way of computing the largest timestep is through the CFL condition.

5.2.2 CFL Condition

The CFL condition [CFL67] is a well-known condition for the convergence (stability) of some explicitly time-integrated PDE's. The condition relates the maximum stable timestep to the ratio between the cell size Δx and the maximum speed by which information can travel in the medium. The CFL condition states that *information should not be able to cross more than one cell at a time.*

In Computer Graphics MPM research, Y. Hu et al.[Fan+18] have utilized the CFL condition to compute effective upper bounds for stable explicit timesteps. Two CFL conditions are utilized: One based on particle velocity, and one based on the speed of sound in the material (i.e. pressure wave speed).

The CFL condition for particle velocity as stated in [Fan+18] is:

$$\Delta t <= \beta \frac{\Delta x}{|v_p|} \tag{5.7}$$

Where $|v_p|$ is the maximum absolute velocity of the particles in the domain. β is the *Courant Number*, which must be below 1 and is often set to 0.5.

Setting the timestep in accordance with the CFL condition can reduce unstable behavior. However, complications arise if the symplectic scheme (see 2.1.6) is used. This scheme first calculates the updated velocity v_p^{n+1} and then advects the systems to x_p^{n+1} using this velocity. However, computation of v_p^{n+1} happens as part of the P2G and G2P transfers. As the timestep must naturally be set at the beginning of a step, the CFL-bound timestep has to be computed using v_p^n. Therefore, the authors of [Fan+18] suggest experimentally adjusting the Courant Number β depending on the scene to ensure stability.

A recent, alternative approach [SSS20] first computes the timestep as in 5.7, followed by a recomputation after v_p^{n+1} has been found. If the timestep is found to be smaller, the velocity is "patched" to account for the change in timestep:

$$v_p^{\hat{n+1}} = v_p^n + \frac{\Delta t'}{\Delta t}(v_i^{\hat{n+1}} v_i^n) \tag{5.8}$$

Where $\Delta t'$ is the new timestep, and $v_i^{\hat{n+1}}$ is the updated grid velocity.

The second CFL condition is based on the dilational pressure wave (or P-wave)

velocity. Like the particle velocity CFL, the condition states that the velocity must be smaller than the grid cell size. This leads to a timestep size of:

$$\Delta t <= \alpha \frac{\Delta x}{c} \tag{5.9}$$

Where $\alpha \in [0.5, 0.9]$ is a scale for stability, and c is the computed pressure velocity. Derivation of pressure velocity computation for various material models can be found in [Fan+18] and [SSS20].

5.2.3 AsyncMPM

In [Fan+18] Y. Fang et al. describe a temporally adaptive MPM implementation based on regional time stepping. The method, dubbed AsyncMPM, uses SPGrid (see 5.3.1) to represent the Eulerian scratch-pad grid. In a temporally synchronous MPM implementation, runtime is limited by the region with the lowest timestep. This can cause significant slowdowns if MPM is used to simulate complex materials with different stability behavior. AsyncMPM seeks to alleviate this issue by updating particles based on their individual timesteps estimates.

Ideally, each particle would be updated individually based on its own timestep calculation. This would unfortunately be highly inefficient, as all the neighboring particles would need to be rasterized to the background grid in order to resample the updated local velocity field for P2G. In addition, the neighboring particles would have to be updated to the same point in time t. Nevertheless, this method could potentially be viable if particle values could be temporally interpolated. Unfortunately, while values like velocity and position are easy to interpolate, interpolating the deformation gradient is nontrivial.

Because updating particles individually is infeasible, the authors bundle particles in blocks (ex. 4x4x8) using the SPGrid data structure. Timesteps are set block-wise as powers of two for synchronization purposes. In order to update a block, one needs knowledge of the immediate neighborhood of blocks. This presents an issue at borders of granularity. If a block is neighbored by a block with a higher timestep, steps must be taken to synchronize the two. AsyncMPM achieves this by the use of "buffer blocks", which are essentially ghost cells inserted at timestep borders. By evolving these blocks to fill the temporal gaps of the high timestep block, the low timestep block can be updated without issue.

Because particles are not fixed in space, and block timesteps are dynamic,

particle-block relationships are constantly changing. As different timesteps can cause particles to take on slightly different trajectories, particles may be duplicated across blocks. Because of this, when creating the particle neighbor pool for a given block, particles are added in ascending order based on the timestep of their block. If duplicate particles are encountered, the particle with the lowest timestep is kept.

The authors of AsyncMPM state that the method can provide up to a 6x speedup over synchronous MPM. This comes with the significant caveat that the actual speedup to a large extent depends on the scene. The overhead of the method amounts to about 50% of runtime, which means the scene has to be simulated significantly faster in order for the method to be quicker overall. In order to examine whether or not AsyncMPM would yield improvements for real-time scenes, an implementation of the method was written in $C++$, adapted from the high-performant method of Y. Hu et al. written in the Taichi language[Fan+18]. For the scenes tested in chapter 6, AsyncMPM caused performance degradations as a result of the overhead. However, for more complex scenes, e.g. scenes with multiple bodies with vastly different stiffness interspersed over several blocks, the method could potentially improve the runtime.

5.3 Implementation

5.3.1 Grid

Morton Ordering

A key characteristic of MPM is the Eulerian scratch-pad grid, which is written to and erased during each iteration. During both P2G and G2P transfer, each particle must access its 3^d neighborhood of grid cells. Grid accesses can therefore take up a significant portion of runtime.

In order to lower the memory access overhead, it is desirable to store the linear grid cell array in a manner that promotes spatial locality. A naïve method would be to order the grid cells based on a linearization where movement is done across one dimension at a time, i.e. $idx = x + y * grid\_res + z * grid\_res^2$. However, this is only locality-promoting across the x-axis, as small movements in the y- or z-plane correspond to massive offsets.

A more suitable alternative are *Space-filling Curves*. Space-filling curves can map a multidimensional space to a linear space while retaining more of the spatial

locality. These curves work by defining a simple N-dimensional curve that can be recursively subdivided unto itself.

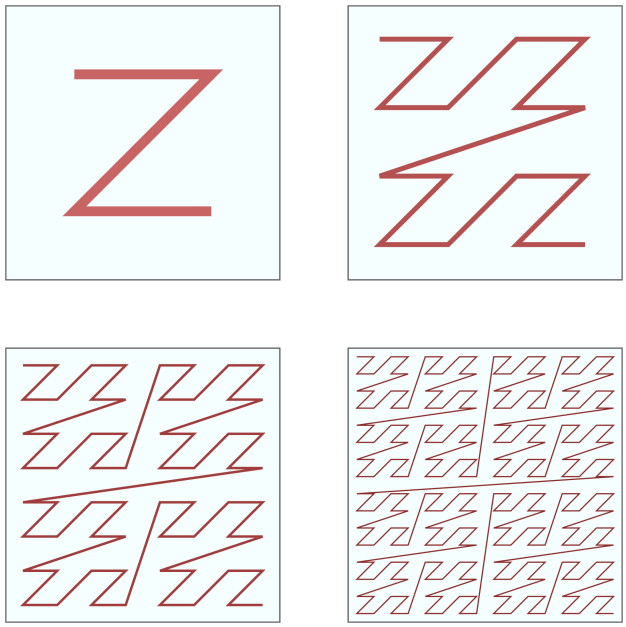

Figure 5.1: Ordering according to a Morton Curve provides improved spatial locality.

One such space-filling curve is the *Morton Curve*. The Morton curve is a simple, Z-shaped curve that provides improved locality. In practice, it works by interleaving the bits of the integers that make up a coordinate using a particular pattern.

SPGrid

SPGrid is an adaptive data structure that allows for compact storage and efficient stream processing of sparsely populated uniform Cartesian grids[Set+14]. One of the key benefits of SPGrid is its ability to enable grid resolutions which would otherwise be prohibitively large. For example, if a grid resolution of 1024^3 was used, and 32 bytes of data were stored per grid cell, ≈ 32 GB of primary memory would be required if a traditional linear array was used. Instead of storing the entire grid in physical memory like this, which would be prohibitively expensive or infeasible without the use of very slow swap-files, the authors instead reserve memory for the entire structure in virtual memory using the Linux function `mmap`.

In order to effectively utilize virtual memory, SPGrid divides the array into 4KB pages which serve as blocks. A block is simply a subsection of the grid containing a set of cells in the form of a rectangular parallelepiped. The 4KB memory allotment per block can be divided in whatever manner is most convenient, e.g. 4x4x4 blocks with 16 channels of 32-bit data.

SPGrid relies upon the fact that sparse data sets by definition contain vast amounts of empty space. As the grid is sparsely populated, only a handful of the 4KB pages will ever be used. Whenever a page is touched, one of two things will occur:

1. **Never touched**: Page fault. Page is reserved in physical memory and recorded in the page table. The page is then zero-filled and made available for use.
2. **Already mapped**: Address translation is handled entirely in hardware.

The authors also provide highly performant methods for sequential and stencil access. Stencil access is done via a bit-wise masked addition of the 1D offsets, foregoing the need for geometric translation between 2D/3D and 1D. The end result is a method that rivals the throughput and parallelism potential of uniform grid methods.

In addition to enabling huge grid resolutions for sparse grids with little overhead, SPGrid also has excellent locality properties. The linearized arrays containing the cell data are encoded using a space-filling Morton Curve.

5.3.2 Particles

During a single step of MPM, particles are accessed at two points: During the P2G substep, and during the G2P substep. The G2P step runs in parallel for each particle, and only requires accessing one particle per thread. During this stage, the majority of memory access overhead comes from grid accesses.

The P2G stage is more complex. Using a naïve scattering approach, where particles write to their 3^d neighborhood, the transfer has to run sequentially to avoid grid write conflicts (see 5.3.3). Furthermore, scattering-based parallelization approaches are generally based on domain decomposition, where subdomains are sequentually executed in parallel. Therefore, optimizing particle memory layout for spatial locality can have a significant impact on performance. A common extension

to MPM involves an initial particle sorting step, where particles are spatially sorted in accordance with access patterns.

Array-of-Structures (AoS) vs Structure-of-Arrays (SoA)

There are two main approaches in terms of memory layout for particle data: *Array of Structures* (AoS) and *Structure of Arrays* (SoA).

The AoS approach stores the particles as a (preferably contiguous) array of structures (`struct` in `C/C++`), where each structure contains all tracked particle quantities. This ensures that locality is maintained when sequentially accessing individual values of a given particle, such as coordinates, velocity, and deformation gradient.

With SoA, we maintain a single structure containing arrays with data for all particles. In other words, we keep separate arrays for each quantity we track on the particles. If this approach is used, performance is dependent on our ability to efficiently utilize all or most data retrieved in cachelines. For example, if we have a 32-bit integer array like `particle_x[]`, which stores the x-coordinates of particles, a single cacheline can retrieve 16 entries. Data for sequentially transferred particles is likely to already be retrieved and stored in the cache. However, this is entirely reliant on whether the particle arrays have been sorted or not. If the arrays are not sorted, cacheline utilization is likely to drop massively. Therefore, an SoA approach is heavily reliant on particles being sorted frequently[Hu+19].

The AoS approach, on the other hand, is guaranteed to have decent utilization of cachelines regardless of particle ordering. Because it excels at random access and is only somewhat slower than SoA at sequential access, AoS is often considered more performant overall [Hu+19].

Recently, [Wan+20] have implemented an *AoSoA* data structure for their GPU implementation of MPM. Their approach intends to combine the best attributes of AoS and SoA. To achieve this, they utilize *binning*. Their implementation uses a modified SPGrid method intended for GPU (GSPGrid), which allows particles to be grouped according to block ownership. Within a block, particles are stored in an SoA manner. The particle structures of each block are then stored in an array, ensuring the best of both worlds. However, it should be noted that their approach is designed for GPU, where coalesced memory accesses are crucial for performance. On the CPU, benchmarks [Hu+19] show that AoS is consistently more performant, even for sequential sorted access.

Coupling of different material models

In order to enable coupling of particles with different material models, a logical approach would be to utilize inheritance. In C++, particles can be stored in a vector containing instances of an abstract `Particle` class, where each instance implements its own methods for stress and strain computation. This allows particles with different parameters and material models to easily interact with one another. However, because the `Particle` vector is abstract, the vector must consist of pointers. This is a natural consequence of the fact that the sub-classes of `Particle` can have any number of members, and thus any size. Because of this, there is no guarantee at all that the initialized particles will be contiguous or even close-by in memory.

Y. Hu et al. have written a high-performance implementation of MPM utilizing the Taichi language, a computer graphics language optimized for computation on sparse data structures such as SPGrid. In their implementation, they circumvent the aforementioned memory locality issue by utilizing *placement syntax*. Using C++'s *placement new* initializer, it is possible to directly give the memory address where an object should be instantiated. Exploiting this, Y. Hu et al. keep an array of of `ParticleContainer`s, which is simply a contiguous section of memory guaranteed to be big enough to hold the biggest particle instance. When initializing new particles, memory is allocated for a `ParticleContainer` and the particle is instantiated at its location using placement new. When retrieving particles, the `ParticleContainer`s are cast to `Particle` pointers using `reinterpret_cast<Particle *>`. This ensures that particles with varying material models can be created and indexed easily while simultaneously exhibiting a high degree of spatial locality.

5.3.3 Parallelization

As explained in chapter 4, the three main stages of standard MPM (and hybrid Lagrangian-Eulerian methods in general) are particle-to-grid (P2G), grid update, and grid-to-particle (G2P). The grid update stage is embarrassingly parallelizable, as each grid cell can be updated independently. Because finding the cell neighborhood for a given particle is trivial, G2P is also embarrassingly parallelizable.

The P2G stage, however, is not. A naïve attempt to run transfers in parallel for each particle would result in a massive amount of write conflicts, as particles with overlapping cell neighborhood would try to write to the same grid cells. The

overhead of atomic operations means such a parallelization sees suboptimal performance gains, if any.

Because of this, attempts at parallelizing MPM mainly focus on the P2G stage. There are two main approaches when attempting to parallelize P2G: *Gathering* and *scattering*[Gao+18].

Gathering

In a gathering approach, the P2G stage runs in parallel for each cell instead of each particle. The cell finds all the particles that influence it and gathers their contributions. This has the effect of eliminating write conflicts, but requires each cell to know which particles have the cell as part of their neighborhood.

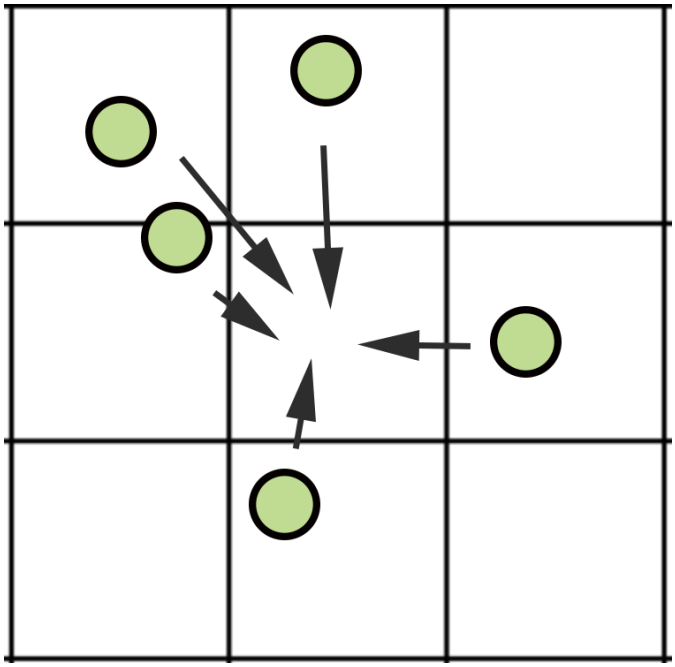

Figure 5.2: In a gathering approach, grid cells/nodes gather contributions from neighboring particles.

[Hua+08] present a parallel CPU-based MPM gathering method based on *domain decomposition*. In this method, the grid is divided into patches, with one patch for each thread. Particles are assigned a local ID within a patch. To do this, particles are first assigned a two-dimensional index (i, j) in an array. Then, each thread is assigned a given segment of particles to count based on their global

indices. The threads count how many times their assigned indices appear in each patch, and a two-dimensional array $pathnp\_h$ keeps track of the number of particles j counted by thread i. The patch counts of each thread are then summed together to yield the final count.

[Chi+09] present a GPU-accelerated gathering method for the GIMP extension of MPM. The method creates a reverse map from cells to particles using a two-dimensional array stored in global memory, where columns correspond to cell nodes and rows correspond to particles. Each nodal array keeps track of the current number of particles added to its neighborhood using an atomic integer. Particles are added to the nodal arrays by using the atomic integer to select the index and incrementing it after insertion. Once this has been constructed, P2G is done using the reverse map to lookup particle-cell relations.

Scattering

In a scattering approach, the P2G stage runs in parallel for each particle, scattering their influences to the neighborhood of grid cells. This makes it trivial to find the relevant particle-cell relations, but requires some way of alleviating write conflicts.

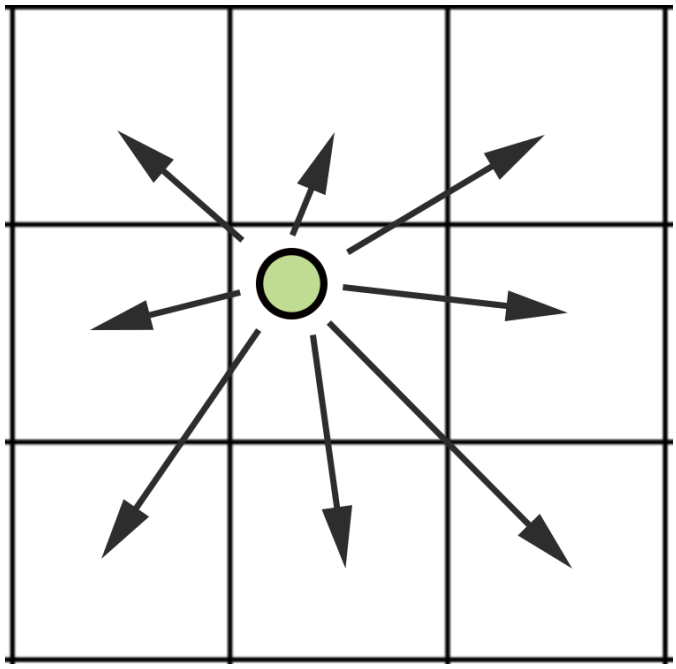

Figure 5.3: In a scattering approach, particles spread their contributions to neighborhood grid cells/nodes.

[Hua+08] present a scattering method that allows for multithreading. The method, called the *array expansion method*, works by expanding the one-dimensional nodal arrays (mass, velocity) with auxiliary arrays for each thread. Each thread writes only to its own array, ensuring no write conflicts occur. After transferring all particle values, the auxiliary arrays can be summed together to yield the final value.

As part of AsyncMPM[Fan+18], Y. Fang et al. propose a scattering method that is based on SPGrid. The grid is divided into blocks of uniform size, e.g. 4x4x8. Performing P2G for particles in a 4x4x8 block requires updating at most the 6x6x10 area covering the block and its immediate neighbors. To avoid write conflicts for blocks with overlapping neighbors, the blocks are partitioned into 2^D sets, where D is the simulation dimensionality. Each set is constructed so that none of the blocks in a given set share any overlap in terms of regions of influence. Once this is in place, blocks in a set can be multithreaded without concern.

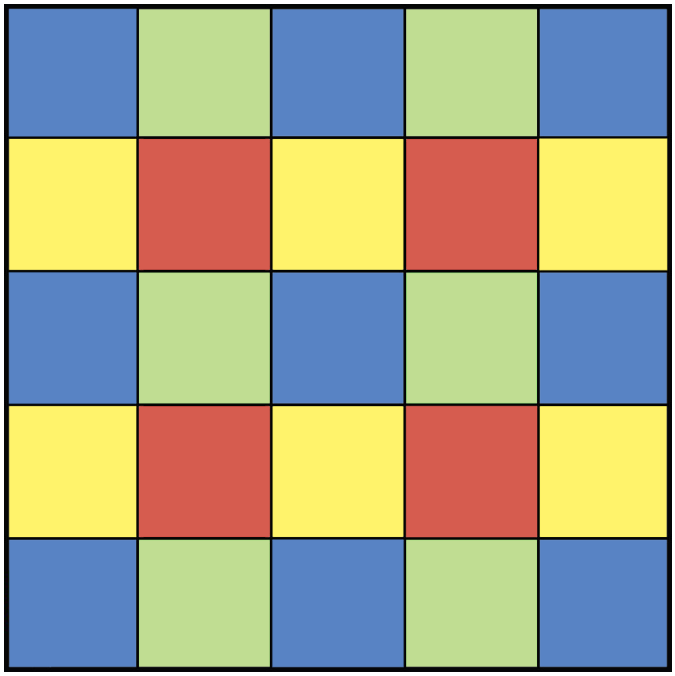

Figure 5.4: The grid is decomposed into non-overlapping groups of blocks. Each color represents a group.

For the method implemented for use in chapter 6, the aforementioned parallelization technique was used.

5.3.4 Particle sampling

In order to simulate a material body, the volume of the body must be represented as a collection of particles. In order to do this, we need some way of sampling the interior volume of the body.

For this purpose, two methods were implemented: The first one randomly samples the volume of the body using a uniform distribution, while the second method distributes samples evenly across the volume.

In order to randomly sample the volume of a closed non-intersecting mesh, one only needs to be able to perform a point in polyhedron test. One way of doing this is to first select an arbitrary point inside the bounding box of the mesh. Then, perform a crossing number test by choosing an arbitrary ray and calculating its ray-triangle intersections with the triangles of the mesh. If the number of intersections is odd, the point is inside the mesh.

Creating an evenly distributed set of samples for a mesh is equivalent to the problem of voxelization. Since there are multiple existing voxelization implementations available, implementing this was straightforward. First, the mesh was transformed into a voxelized representation using the open-source tool binvox [Min21]. Then, the voxelized representation was read in via `C++`, and a particle was spawned for each voxel. In addition to allowing for evenly spaced sampling of bodies, this approach is also easily extendable. The coarseness of the grid can be refined or reduced simply by altering how many particles are spawned per voxel. In addition, different distribution such as blue noise may be emulated by perturbing the spawn location of each particle.

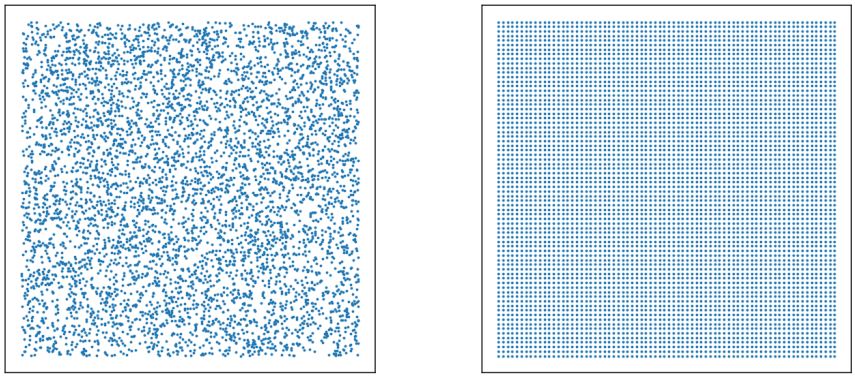

Figure 5.5: Comparison of uniform random (left) and evenly distributed (right) selection.

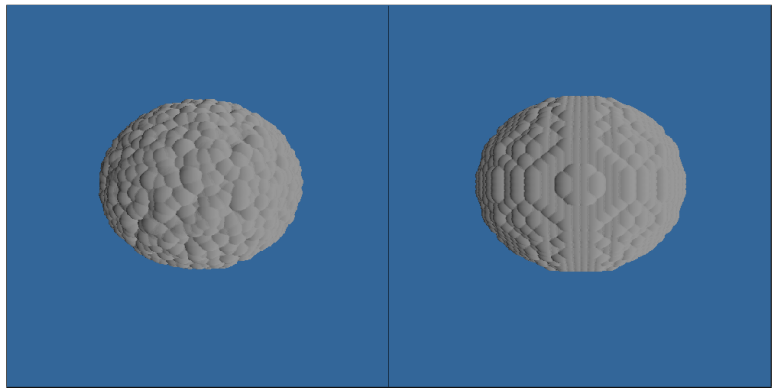

Figure 5.6: Comparison of uniform random (left) and evenly distributed (right) particle sampling.

5.4 Rendering

When aiming to do physical simulation in real time, it is generally desirable to also be able to render the material body in real time as well. Fortunately, a vast body of literature exists for rendering physically simulated bodies.

In order to render the material body, we need to know where the body is located in space. The obvious choice would be to use the particles, as they persistently track the position of the body. However, since we only purge the grid values at the beginning of a step, we are also able to use the rasterized particle positions on

the grid. The grid approach is aptly termed Eulerian rendering, while the particle approach is termed Lagrangian rendering. This section discusses various methods and concerns for each approach.

5.4.1 Eulerian Rendering

When aiming to render a body represented as a 3-dimensional grid, the common approach is to treat the grid as a volume. With this, one can draw on the vast literature of volume rendering techniques. A popular approach is to use direct volume rendering techniques such as volumetric ray marching or splatting. Direct volume rendering techniques are mature and can both provide excellent visual results and/or low runtime, depending on which method is chosen. However, they suffer from the disadvantage of being potentially difficult to integrate into a traditional rasterization-based graphics pipeline, as they are not mesh-based. Furthermore, these methods require interpolating values from grid nodes, which could be costly and potentially inaccurate, depending on the grid resolution.

A mesh-based option is to use an isosurface extractor, which generates a mesh of the simulated body based on values stored in the grid. One such option is the popular Marching Cubes algorithm[LC87]. The Marching Cubes algorithm works by iterating over each cube, i.e. each set of 8 inter-connected nodes and 12 edges that make up a cube. The density values at each node are compared against an isosurface threshold to determine whether they are inside or not. Then, a predetermined partial mesh is selected based on the configuration of nodes that are inside the threshold. Finally, all the partial meshes are sewn together to create the final surface.

A benefit of the Marching cubes class of algorithms is that they mesh fairly well with SPGrid. When implementing Marching Cubes, we are only interested in cells where one or more nodes belong to the material body. For sparse grids, the overhead of filtering out empty cells can become prohibitively expensive. However, with SPGrid we already keep track of which blocks are active. Therefore, we need only run isosurface extraction on active blocks, lowering the runtime significantly. Furthermore, Marching Cubes is easily parallelizable and can run on the GPU, which allows us to utilize more of the CPU for simulation.

Rendering methods based on Eulerian grids can be both easy to implement and run at real-time levels (depending on the grid resolution). In addition, many of these methods are highly parallelizable and have been successfully implemented on the GPU. Unfortunately, the application of these methods to standard MPM

has a significant drawback. As the grid is populated on each step by all simulated particles, coupled simulations (i.e. simulations with different bodies interacting with each other) are difficult to mesh. As an example, consider a ball dropping into a pool of water. While the two bodies are separate, they are extracted as different meshes. However, as the ball enters the water, the meshes connect and blend together, making it difficult to distinguish the two bodies from one another. This problem is exacerbated by the fact that the grid resolution must be low enough to enable real-time meshing. A potential solution would be attempt to distinguish material bodies innately in the simulation, e.g. by flagging particles belong to different bodies and tracking separate density arrays on grid cells for each body. However, this would incur significant overhead, and would fundamentally not be scalable.

5.4.2 Lagrangian Rendering

Lagrangian Rendering methods render a set of particles as a cohesive surface. Just as with Eulerian rendering, both mesh-based and direct rendering approaches exist. However, mesh-based approaches generally work by first rasterizing the particles to a grid, before using some Eulerian extraction method, e.g. Marching Cubes. While this can be used in conjunction with particle flagging to solve the issue of separating the surfaces of coupled bodies, it is inefficient compared to fully Eulerian methods, as it has the performance of multiple-pass Marching Cubes with the added overhead of grid rasterization.

For direct rendering approaches there exists a plethora of methods intended to render SPH simulations, a particle-based Lagrangian fluid simulation method. As these methods generally only rely on simple particle data such as position, they are often applicable to MPM rendering as well. For the purposes of rendering the scenes in chapter 6, the method described in [XZY17], *Real-Time High-Quality Surface Rendering for Large Scale Particle-based Fluids*, was used. This method combines ray-marching with splatting in order to efficiently render bodies consisting of hundreds of thousands of particles at real-time rates. As no open-source implementation existed at the time of writing, a new implementation was written for the purposes of this thesis. The method was also adapted for visualization of hyperelastics.

The method involves the following steps:

1. Render particles (e.g. using point primitives) into a depth buffer.

2. In screen-space, march rays towards the material body, using the depth buffer location obtained in the previous step as the starting point for the ray.

3. While the isosurface has not been found, march the ray and sample the density of the local neighborhood of particles near the sampling point in order to obtain a density estimate.

4. When the isosurface point has been discovered, estimate the normal using nearby particle values.

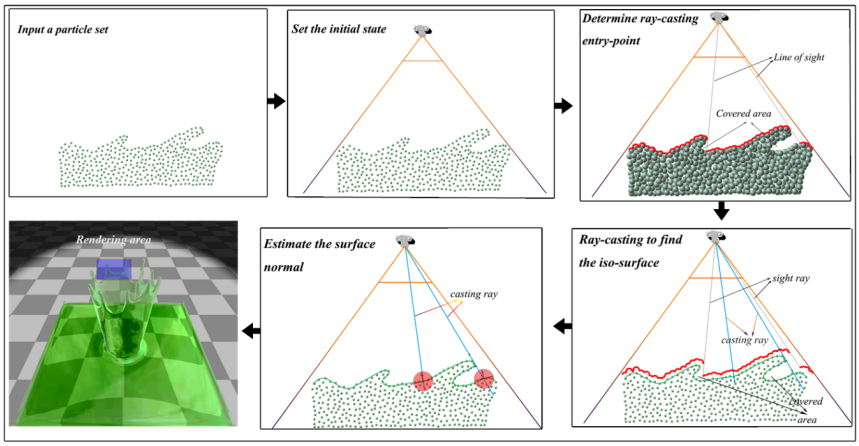

Figure 5.7: Overview of steps in the implemented method. Picture from [XZY17].

The main bottleneck for direct rendering methods for SPH (and for SPH itself) is the computation of fixed-radius nearest neigbhors. When rendering the body, we need to find the subset of particles that have some distance $< d$, a fixed number, from the sample point. The obvious approach would be to iterate through all particles, computing their distance to the sample point. However, even when the method is massively parallelized on the GPU for each ray, this quickly becomes prohibitively expensive. In order to achieve real-time rates with high particle counts, an improved method needs to be used. The common choice is use some lookup structure that stores particles based on which cell they belong to. With this, a sample point only needs to lookup the particles belonging to its neighborhood of cells, which massively reduces runtime.

As GPU data structures need to be limited in their complexity, the particle locations were stored on the GPU as a linear array. This array was sorted according to cell ownership using a Morton Ordering (see 5.3.1) to promote spatial locality. For each populated cell, a 32-bit integer was bit-packed to store both the linear

array offset as well as the number of particles in a cell. Then, in order to maintain the low memory overhead of SPGrid, the GPU hashtable structure used in [Alc+09] (implemented in the open-source library CUDPP) was used to translate the linearized indices into the bit-packet offsets. This ensured that sparse grids of arbitrary resolution could be used without causing unmanagable memory overhead.

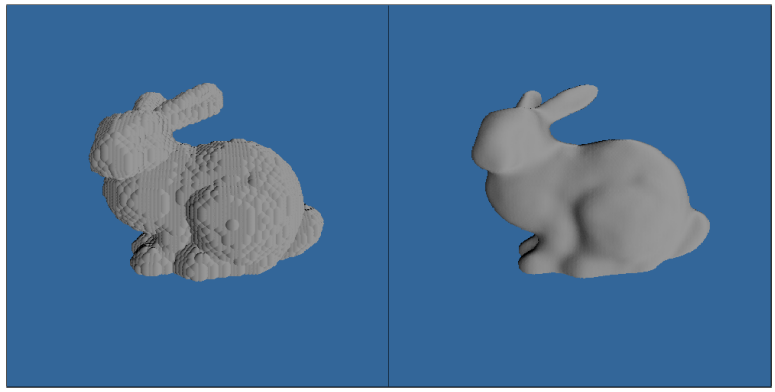

Figure 5.8: Left: Primitive rendered for each particle. Right: Particle-based surface rendering. Particle neighborhoods are sampled to compute isosurface and estimate normal.

While this rendering approach can provide good visual results, depending on the number of particles and the steps taken when raymarching, it suffers from many of the same drawbacks as other direct volume rendering methods. In particular, it is fairly cumbersome to enable light interaction between bodies rendered using the method and meshes drawn using a traditional rasterization pipeline. This makes phenomena such as shadows and indirect lighting difficult to implement.

5.5 Choice of Implementation

For the purposes of evaluating the practicability of real-time, a high-performance CPU-based MPM implementation was written in `C++` in accordance with the practices outlined in this chapter. Linear algebra was computed using the high-performant Eigen library[GJ+10]. Both the APIC transfer scheme as well as MLS-MPM were implemented. For the grid data structure, SPGrid was utilized, with a grid resolution of 128. Spatial locality of particles was enabled using the methods outlined in 5.3.2. In addition, block-level caching was added for the P2G stage, an implementation detail borrowed from Y. Hu et al.'s high-performance MPM solver [Fan+18].

The grid update stage and the G2P stage were made to run in parallel on each grid cell, enabling massive parallelization. The P2G stage was parallelized in accordance with the domain decomposition method outlined in 5.3.3. For the purposes of evaluation the choice of timestep as done experimentally, not through the CFL condition. This was done in order to determine the highest possible timestep, while the CFL condition would underestimate. Finally, the Lagrangian Rendering algorithm described in 5.4.2 was used was used.

Chapter 6

Evaluation

In this section, different parameterizations and implementation choices will be evaluated in order to determine their suitability for real time MPM. Two main metrics are used for evaluating the performance of a scene: The *largest stable timestep*, and the *ratio of runtime to timestep*.

When aiming for real time simulation, the largest stable timestep (see 5.2) is a key limiting factor. As the time required to compute a single iteration of the MPM algorithm is independent of the size of size of the timestep, a higher timestep directly leads to a faster runtime. The stability of a scene may vary depending on multiple different factors, some of which are evaluated in this chapter.

The ratio of runtime to timestep is the ratio of time spent computing a timestep relative to the size of that timestep. It is simply given as:

$$performance = \frac{computational\ time}{\Delta t} \tag{6.1}$$

From this, it is clear to see that increasing the timestep leads to a linear increase in runtime. However, it should be noted that for real-time this increase mainly matters while Δt is smaller than the desired frame-rate, as increasing it beyond this would lead to an inaccurate speed-up of the simulation.

In the remainder of this chapter, a comparative evaluation will be done for the following:

- Particle Seeding methods: Uniform random vs evenly sampled (see 5.3.4).

- Constitutive models: Neo-Hookean vs Fixed Corotated (3.11 and 3.17.3).
- Particle sampling density: $\Delta x = 1.0$, $\Delta x = 0.5$ and $\Delta x = 0.3$.
- Varying Particle count: Comparison of deformation quality for the same model with different particle counts.
- Coupling of bodies: Hard-hard, Hard-soft and soft-soft coupling.

Two sets of material parameters (see 3.15 and 3.16) have been used for evaluation. The first set exhibits comparatively stiffer material behavior, while the second is comparatively softer:

	Parameter set 1	Parameter set 2
Young's Modulus	100 000	10 000
Poisson's Ratio	0.3	0.2
μ	$\approx$ 38 462	$\approx$ 4167
λ	$\approx$ 57 692	$\approx$ 2778

6.1 Particle Seeding

Two methods of particle seeding were described in 5.3.4: Random uniform distribution, and evenly spaced samples. These methods are evaluated based on a qualitative assessment of their deformation, as well as their experimentally determined largest stable timestep for the given scene. No measure of performance is given. While the two methods may have different evolution of deformation and stable timestep, partially as a result of different volume estimations, their runtime is only marginally different depending on differences in deformation.

The following scene was used for testing:

Description	A gravity-affected sphere bouncing on the ground.
Material Model	Varying
Distribution	Varying
Particle count n_p	16 048
Density Δx	0.5

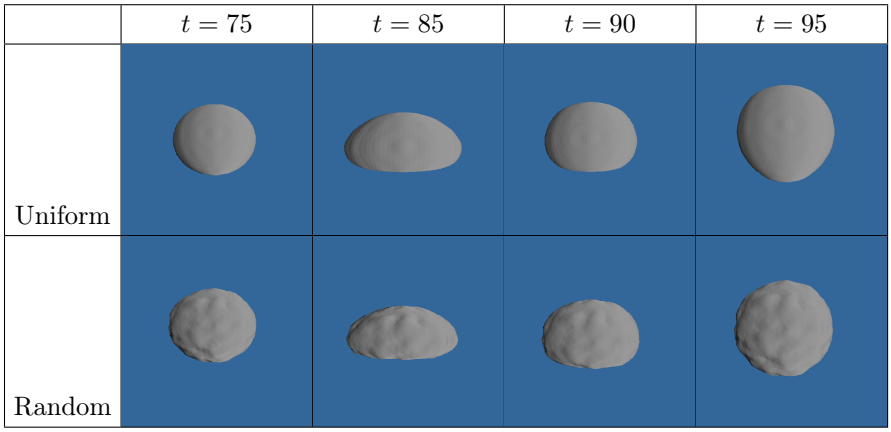

Figure 6.1: Comparison of deformation at different stages in time.
Material Model: Neo-Hookean with parameter set 1.

The evolution of deformation in time can be seen in figure 6.1. Barring the visual quality of the surface, which stems from the rendering algorithm used, the visual differences can be observed to be relatively minor.

The following largest stable timesteps were estimated for each of the material models:

Material model	Distribution	max Δt
Neo-Hookean 1	Uniform	0.006
Neo-Hookean 1	Random	0.005
Neo-Hookean 2	Uniform	0.012
Neo-Hookean 2	Random	0.015
Corotated 1	Uniform	0.006
Corotated 1	Random	0.005
Corotated 2	Uniform	0.019
Corotated 2	Random	0.018

As expected, the softer material set permits a higher timestep. Notably, the uniform distribution has a slightly larger permissible timestep for the stiffer set of material parameters. The random distribution is more stable for the soft Neo-Hookean model, but not for the soft Corotated model. Overall, the uniform distribution appears to be slightly more stable for the given scene, but not drastically so.

6.2 Constitutive Models

In this section, a comparative evaluation of the Neo-Hookean and Corotated Constutive model is done. These methods are described in 3.11 and 3.17.3. The other two methods described (Linear and St. Venant-Kirchoff) are not evaluated due to their issues with large deformation scenarios, which are of interest in real-time interactive scenarios.

The following scene was used for testing:

Description	A gravity-affected bunny bouncing on the ground.
Material Model	Varying
Distribution	Uniform
Particle count n_p	52 730
Density Δx	0.5

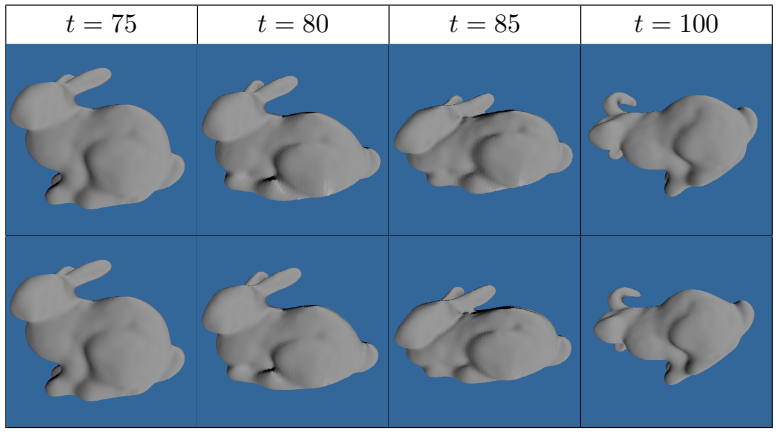

Figure 6.2: Comparison of deformation at different stages in time using hard material parameters. Top: Neo-Hookean. Bottom: Corotated Constitutive Model.

Figure 6.2 shows a comparison in deformation using the hard material parameters. As can be seen, the differences in visual quality are essentially negligible.

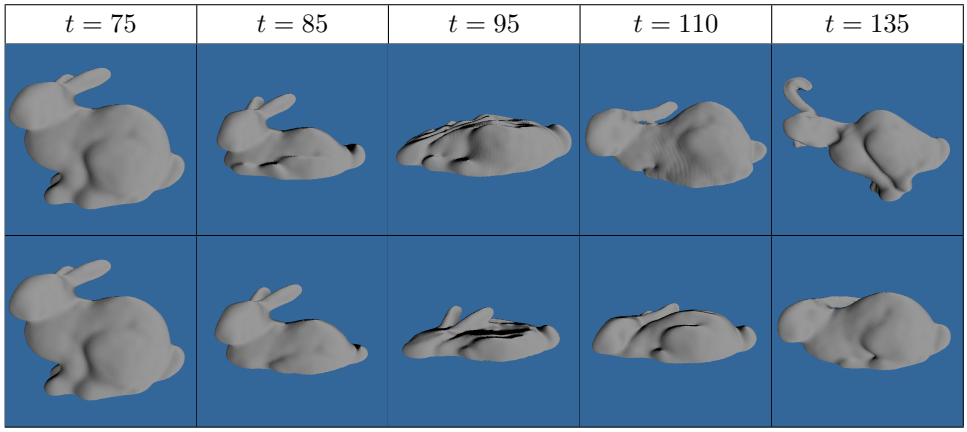

Figure 6.3: Comparison of deformation at different stages in time using soft material parameters. Top: Neo-Hookean. Bottom: Corotated Constitutive Model.

For the soft set of material parameters, shown in figure 6.3, the differences in deformation are more pronounced. Coming out of the bounce at timestep $t = 110$ and $t = 135$, the Neo-Hookean model exhibits more extreme elastic behavior, straying from the intial configuration of the material body. The corotated model, on the other hand, better maintains and recovers its initial configuration during and after collision.

The large stable timesteps and runtime for each model were determined to be as follows for the scene:

Material model	max Δt	Runtime
Neo-Hookean 1	0.003	275.96%
Neo-Hookean 2	0.012	85.72%
Corotated 1	0.003	341.26%
Corotated 2	0.011	146.74%

As expected, each model is more stable for the softer set of material parameters. In addition, their stable timesteps are comparatively close. However, the corotated model is significantly more expensive. This comes as a result of the comparatively expensive polar decomposition required for the stress computation of the corotated model.

Comparing the average time spent during each stage in a single iteration, we see that the overhead primarily comes from the P2G step:

Material Model	P2G	Grid Update	G2P
Neo Hookean 1	6.317 ms	0.249 ms	2.1564 ms
Corotated 1	9.908 ms	0.246 ms	2.1498 ms

6.3 Varying particle density

Particle density refers to the number of particles that are spawned per grid cell. This section evaluates the relation between particle density and deformation/performance. Particle density is measured in terms of the distance Δx between particles.

The following scene was used for testing:

Description	A gravity-affected bunny bouncing on the ground.
Material Model	Neo-Hookean
Distribution	Uniform
Particle count n_p	52 730
Density Δx	Varying

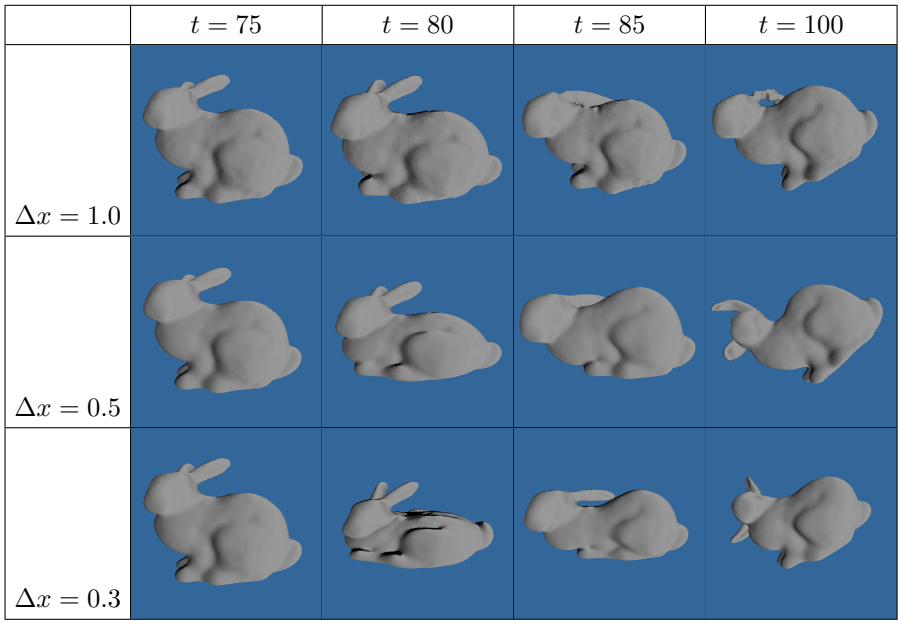

Figure 6.4: Comparison of deformation at different stages in time with varying particle density. Material Model: Neo-Hookean with parameter set 1.

As seen in figure 6.4, there are significant differences in deformation depending on particle density. In particular, a more dense particle distribution results in softer behavior. With $\Delta x = 1.0$, the deformation is stiffer, with tearing occuring at the ears of the model. A density of $\Delta x = 0.3$ results in softer, more pronounced deformation, while a density of $\Delta x = 0.5$ lands somewhere in between.

Stable timesteps for the scene were estimated to be:

Δx	Material model	max Δt
1.0	Neo-Hookean 1	0.001
1.0	Neo-Hookean 2	0.005
0.5	Neo-Hookean 1	0.003
0.5	Neo-Hookean 2	0.012
0.3	Neo-Hookean 1	0.005
0.3	Neo-Hookean 2	0.014

The $\Delta x = 1.0$ density, which had stiffer behavior, intuitively also has a lower stable timestep. The difference between $\Delta x = 0.5$ and $\Delta x = 0.3$ is less pronounced, in particular for the softer material model.

6.4 Varying particle count

Of particular interest for real-time is the difference in visual quality of deformation for the same model with different particle counts. Particle count directly impacts performance, so real-time simulation relies on finding a good compromise of visual quality and runtime.

Description	A gravity-affected bunny bouncing on the ground.
Material Model	Neo-Hookean
Distribution	Uniform
Particle count n_p	Varying
Density Δx	0.5

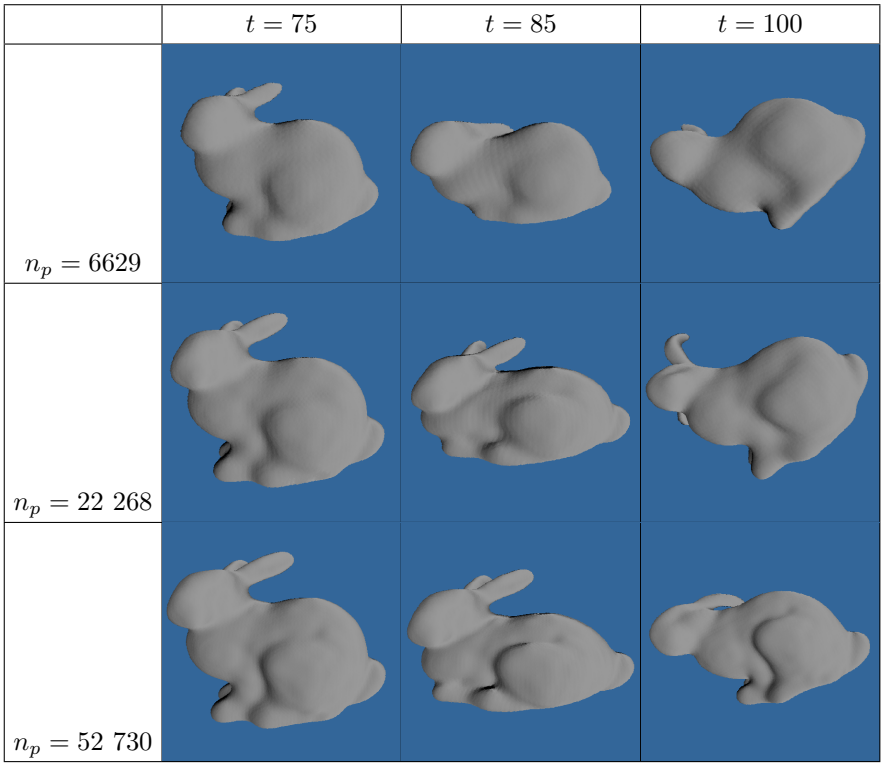

Figure 6.5: Comparison of deformation at different stages in time with varying particle count.

Figure 6.5 shows deformation at different stages in time for each particle count. From the initial, undeformed configuration we can see that a higher particle count translates to more detail using the rendering algorithm described in 5.4.2. At timesteps $t = 85$ and $t = 100$, there is a significant difference in deformation between the model with the lowest particle count and the others. Minor model pieces such as the ears are less detailed in their deformation, and compression is less comprehensive. However, the difference in visual quality between $n_p = 22\ 268$ and $n_p = 52\ 730$ is not as pronounced.

The largest stable timestep and performance ratio were found to be:

n_p	Material model	max Δt	Runtime
6629	Neo-Hookean 1	0.002	173.9%
6629	Neo-Hookean 2	0.009	40.72%
22 268	Neo-Hookean 1	0.002	317.64%
22 268	Neo-Hookean 2	0.011	59.52%
52 730	Neo-Hookean 1	0.003	275.96%
52 730	Neo-Hookean 2	0.012	85.72%

Average time spent during each stage in a single iteration were found to be:

n_p	P2G	Grid Update	G2P
6629	2.827 ms	0.091 ms	0.035 ms
22 268	4.604 ms	0.144 ms	0.879 ms
52 730	6.317 ms	0.249 ms	2.1564 ms

While the higher particle count results in slower runtime at all stages, it also yields a higher stable timestep. In addition, as the higher particle count model covers a larger area of the grid, it is also able to bettwe utilize the P2G domain decomposition parallelization strategy.

Interestingly, the highest particle count model using material parameter set 1 is faster than the second largest, despite its particle count being more than twice as high. This is likely a result of two things: One, it is able to take a 50% higher timestep, and two, it is able to better utilize domain the decomposition-based parallelization of the P2G stage.

6.5 Coupling of Bodies

One of the key benefits of MPM is easy coupling of different material models. For the purposes of testing this, three scenes were set up: One with a hard-hard (material parameter set 1) coupled interaction, one with a soft-soft (material parameter set 2) coupled interaction, and one with a hard-soft coupled interaction.

Description	A gravity-affected sphere colliding with a sphere at rest.
Material Model	Neo-Hookean
Distribution	Uniform
Particle count n_p	32 096
Density Δx	0.5

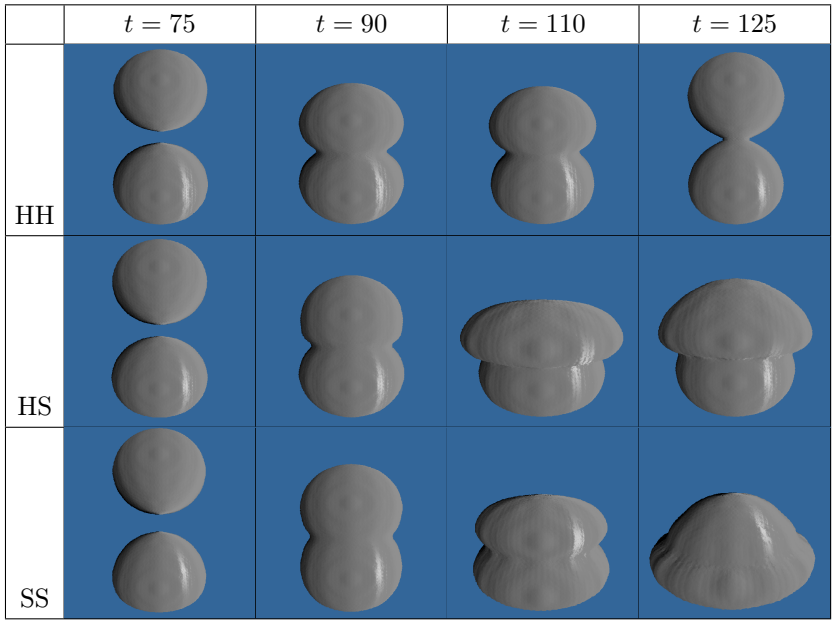

Figure 6.6: Comparison of deformation at different stages in time for various coupled interactions. HH: Hard-hard. HS: Hard-soft (Soft sphere on top). SS: Soft-soft.

The largest stable timesteps and performance ratio were found to be:

Coupling	max Δt	Runtime
Hard-Hard	0.006	88.32%
Hard-Soft	0.006	90.41%
Soft-Soft	0.017	52.05%

As expected, the scene with soft-soft coupling had the largest stable timestep. Interestingly, there was no distinction between the timestep of the hard-hard and the hard-soft scene. In other words, the largest stable timestep was dependent on the stiffest material in the scene, and was not negatively affected by coupling of two bodies.

Performance wise, the soft-soft scene was significantly faster as a consequence of its higher permissible timestep. The hard-hard and hard-soft scenes expectedly had similar performance, with some slight difference as a consequence of different deformation.

Chapter 7

Conclusion and Further Work

Using the implementation detailed in chapter 5 we were able to achieve real-time simulation rates for several scenes. Based on the evaluation in chapter 6, the following conclusions are drawn with regards to the feasibility of real time MPM on the CPU:

In terms of choice of particle seeding, there was little difference in terms of stable timestep when comparing the uniform random and evenly spaced distributions. The evenly spaced method was preferred for subsequent evaluations due to its comparative ease of implementation, but as different scenes likely have somewhat different stability behavior it is difficult to recommend one over the other.

For constitutive models, the Neo-Hookean was significantly faster. However, it did produce more extreme elastic deformations in the scene using the second set of material parameters. Because it had a significantly faster runtime, however, it is overall preferred for real time applications, as it is still able to produce physically probable results.

Varying particle density provided an interesting set of tradeoffs. Increasing particle spacing resulted in stiffer behavior with more tearing, but also resulted in a lower stable timestep. For real-time scenes where it is desirable to simulate stiffer hyperelastics that exhibit tearing phenomena, increasing particle spacing might permit better performance than increasing material parameters. Increasing particle density resulted in softer elastic behavior, indicating that a there exists an optimal tradeoff between material parameter stiffness and particle spacing for a given scene.

When simulating the same body with different particle counts, we were able to achieve quite convincing model deformation with a particle count as low as 6629. While there were clear visual improvements to deformation with a higher particle count, these improvements were also diminishing. Furthermore, as the low particle count model was small and unable to fully utilize P2G parallelization, it would be feasible to simulate multiple low-particle count bodies in a scene.

In simulating coupling between bodies with different material models, we confirmed that the largest stable timestep was entirely bound by the stiffest body in the scene. For scenes containing coupling between several bodies with different material models, the high-overhead temporally adaptive method AsyncMPM (see 5.2.3) may be able to produce runtimes that approach real time performance.

In conclusion, the current state of the art for CPU-based MPM is able to effectively simulate hyperelastic bodies in real time, although with reduced physical accuracy. With necessary concessions to particle count, effective parallelization of P2G, and a sufficiently soft material model, a variety of interesting scenes can be simulated. Furthermore, if concessions to runtime are acceptable, a whole host of additional scenes can be simulated in slow motion with real time framerates. This result has interesting potential applications in domains that can feature interactive physical simulation, in particular video games.

There are many additional avenues to explore with regards to real-time MPM. The scope of this thesis has been limited to fairly simple hyperelastic models. Much of the power and subsequent interest of MPM in computer graphics comes from its extensibility and the variety of materials that can be simulated. Enabling more advanced models (e.g. models that involve plasticity or phase-changing phenomena such as melting) to be simulated in real-time could have very promising applications.

Recent research has also produced promising results for real time GPU-based simulations[Gao+18] [Wan+20]. CPU-based MPM has desirable properties for real time applications like video games because it leaves the GPU available for rendering. However, a high-performant GPU MPM implementation coupled with a low-overhead rendering method could produce results exceeding those possible on CPU.

Appendices

Appendix A

Stress

In continuum mechanics, the quantity of stress is used for describing the forces acting on internal points of a material body. We can in turn use this to describe the movement and deformation of the body by relating stress to strain.

In order to understand stress, it is useful to mention the types of forces that can act on a continuum body. To this end we distinguish between *body forces* and *surface forces*. A body force can be thought of as any force that does not arise from physical contact between bodies. The standard example is gravity.

Surface forces are forces that do arise from physical contact. Intuitively, we think of these as forces arising from collision between bodies, or as a result of self-collision. However, to understand stress we need to further classify these types of collisions as *external surface forces*, or forces that act on the bounding surface of the body. This is to distinguish between external and *internal surface forces*. Internal forces are forces that act along some imaginary surface (oftentimes a cross-section) within the interior of the body.

Stress is a measure of the internal forces that the neighbouring particles of a material exert on each other.

To formalize and quantify this, it is useful to first define the traction vector (sometimes known as the stress vector). Simply put, the traction vector is force per unit area F/A for some surface (internal or external) in the region B of the material body [Kel13]. To see why this concept is useful, consider the example of a heavy object hanging from a rope. As Galileo observed, it is not the weight of the object that determines if the rope will break, but the weight divided by the

cross-sectional area of the rope, i.e. force per unit area.

In order to define the stress tensor σ, we need Cauchy's postulate. We define Γ as an arbitrary, oriented surface in B. Γ has a unit normal field $\hat{n} : \Gamma \to V$, where V is the set of all unit vectors. The unit normal field gives us the normal $\hat{n}(x)$ for every point x on the surface.

Cauchy's postulate states that all surfaces Γ through a point x that share unit normal $\hat{n}$ at x have the same traction vector.

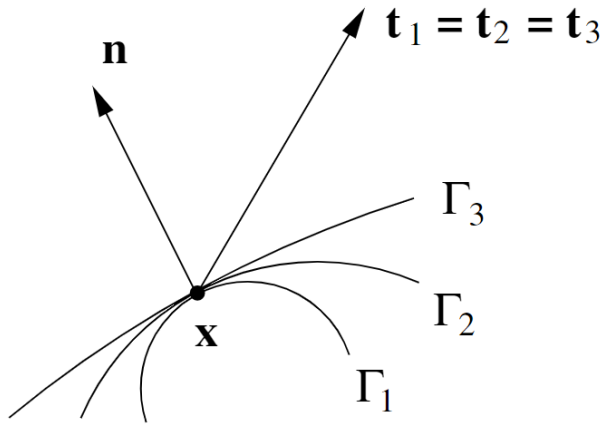

Figure A.1: Cauchy's Postulate lets us describe stress via stress tensor σ. Picture from [GS08].

Furthermore, as detailed in [GS08], it can be shown that $t(-n, x) = -t(n, x)$, i.e. the traction t of a surface with normal n has an equal and opposite counterpart $-t$ for the mirror surface with normal $-n$.

Given this, we can describe the traction of any surface through a point x by its normal n and the **stress tensor** σ:

$$t(n, x) = \sigma(x)n \tag{A.1}$$

Where σ is a second order tensor which can be represented by a $[d, d]$ matrix (where d is dimensionality). The entries of the stress tensor can be thought of as the traction magnitudes for a infinitesimal square/cube at point x:

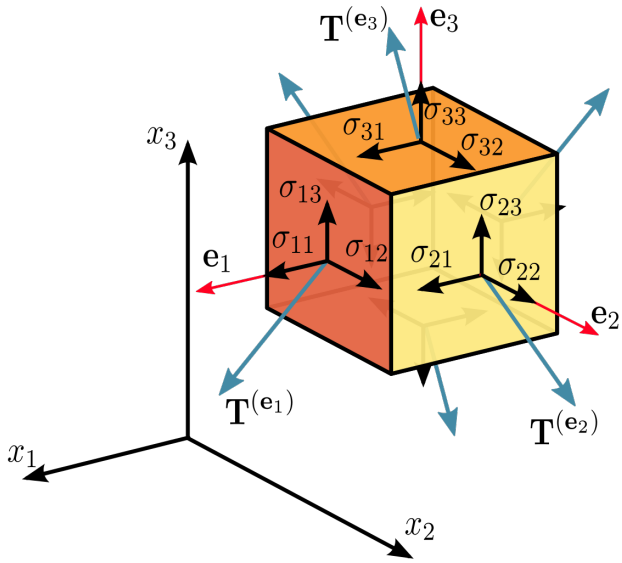

Figure A.2: Visual illustration of the components of the stress tensor in 3D space.

Appendix B

Cell-Crossing Instability

The original Material Point method proposed by Sulsky et. al. [SCS94] used simple linear hat functions like the one in figure 2.3. While these linear shape functions are simple and cheap to compute, they are also only C^0 continuous. This causes an issue known as the *cell-crossing instability*/grid-crossing instability.

A cell-crossing instability occurs when a particle crosses a position where the derivative of a shape function is discontinuous. Examining the linear Lagrangian shape functions of the original MPM in figure 2.3, we see that they are linearly piecewise continuous. When a particle crosses over to another cell, the derivative of the basis function changes sign and a discontinuity occurs.

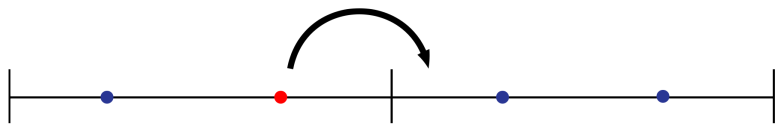

Figure B.1: A cell crossing instability can occur when a particle crosses between cells.

Since the traditional method of calulating the stress forces affecting a node involves the derivative of the shape function (see 4.4), the internal force of a given node will suddenly change once a particle crosses over. Assuming the particles in figure B.1 contribute equal stress to the node, the system is in equilibrium. However, once a particle crosses over, the internal forces abruptly shift. This error can cause massive un-physical spikes in stress, which in turn can lead to inaccurate physical results and instability.

Multiple methods have been developed to prevent cell-crossing errors. Some of the most common methods are MPM with B-Spline shape function (described in 4.2), the generalized interpolation material point method (GIMP) and its derivatives, and recently the Total Lagrangian MPM (TLMPM)[Vau+20]. In computer graphics research, B-spline MPM [Sto+13] and GIMP [Gao+17] are fairly common.

Bibliography

[Alc+09] Dan A. Alcantara et al. “Real-Time Parallel Hashing on the GPU”. In: *ACM Trans. Graph.* 28.5 (Dec. 2009), pp. 1–9. ISSN: 0730-0301. DOI: 10.1145/1618452.1618500. URL: https://doi.org/10.1145/1618452.1618500.

[Arb20] Peter Arbenz. *The Poisson Equation*. ETH Zurich, June 2020. URL: http://people.inf.ethz.ch/arbenz/FEM17/pdfs/0-19-852868-X.pdf.

[BK04] S. Bardenhagen and Edward Kober. “The Generalized Interpolation Material Point Method”. In: *CMES - Computer Modeling in Engineering and Sciences* 5 (June 2004).

[BR86] J.U. Brackbill and H.M. Ruppel. “FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions”. In: *Journal of Computational Physics* 65.2 (1986), pp. 314–343. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(86)90211-1. URL: http://www.sciencedirect.com/science/article/pii/0021999186902111.

[Bri08] R. Bridson. *Fluid Simulation for Computer Graphics*. Ak Peters Series. Taylor & Francis, 2008. ISBN: 9781568813264. URL: https://books.google.no/books?id=gFI8y87VCZ8C.

[CFL67] R. Courant, K. Friedrichs, and H. Lewy. “On the Partial Difference Equations of Mathematical Physics”. In: *IBM Journal of Research and Development* 11.2 (1967), pp. 215–234. DOI: 10.1147/rd.112.0215.

[Chi+09] Wei-Fan Chiang et al. “GPU Acceleration of the Generalized Interpolation Material Point Method”. In: (2009).

[Dre07] Ulrich Drepper. “What Every Programmer Should Know About Memory”. In: (2007). URL: https://akkadia.org/drepper/cpumemory.pdf.

[Fan+18] Yu Fang et al. "A Temporally Adaptive Material Point Method with Regional Time Stepping". In: *Computer Graphics Forum* 37.8 (2018), pp. 195–204. DOI: https://doi.org/10.1111/cgf.13524. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13524. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13524.

[Fu+17] Chuyuan Fu et al. "A polynomial particle-in-cell method". In: *ACM Transactions on Graphics* 36 (Nov. 2017), pp. 1–12. DOI: 10.1145/3130800.3130878.

[Gao+17] Ming Gao et al. "An Adaptive Generalized Interpolation Material Point Method for Simulating Elastoplastic Materials". In: *ACM Trans. Graph.* 36.6 (Nov. 2017). ISSN: 0730-0301. DOI: 10.1145/3130800.3130879. URL: https://doi.org/10.1145/3130800.3130879.

[Gao+18] Ming Gao et al. "GPU Optimization of Material Point Methods". In: *ACM Trans. Graph.* 37.6 (Dec. 2018). ISSN: 0730-0301. DOI: 10.1145/3272127.3275044. URL: https://doi.org/10.1145/3272127.3275044.

[GJ+10] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[GS08] Oscar Gonzalez and Andrew M. Stuart. *A First Course in Continuum Mechanics*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2008. DOI: 10.1017/CBO9780511619571.

[Har63] Francis H Harlow. *The particle-in-cell method for numerical solution of problems in fluid dynamics*. Tech. rep. Los Alamos Scientific Lab., N. Mex., 1963.

[HER55] F.H. Harlow, M. Evans, and R.D. Richtmyer. *A Machine Calculation Method for Hydrodynamic Problems*. LAMS (Los Alamos Scientific Laboratory). Los Alamos Scientific Laboratory of the University of California, 1955.

[Hoe13] Rama C. Hoetzlein. *Fast Fixed-Radius Nearest Neighbors: Interactive Million-Particle Fluids*. 2013.

[Hu+18a] Yuanming Hu et al. "A Moving Least Squares Material Point Method with Displacement Discontinuity and Two-Way Rigid Body Coupling". In: *ACM Trans. Graph.* 37.4 (July 2018). ISSN: 0730-0301. DOI: 10.1145/3197517.3201293. URL: https://doi.org/10.1145/3197517.3201293.

[Hu+18b] Yuanming Hu et al. "ChainQueen: A Real-Time Differentiable Physical Simulator for Soft Robotics". In: (2018). arXiv: 1810.01054 [cs.RO].

[Hu+19] Yuanming Hu et al. “On hybrid lagrangian-eulerian simulation methods: practical notes and high-performance aspects”. In: *ACM SIGGRAPH 2019 Courses*. ACM. 2019, p. 16.

[Hua+08] P. Huang et al. “Shared Memory OpenMP Parallelization of Explicit MPM and Its Application to Hypervelocity Impact”. In: *CMES - Computer Modeling in Engineering and Sciences* 38 (Dec. 2008).

[Jan21] JangaFX. *EmberGen: Real-time Fluid Simulations For Fire, Smoke and Explosions*. 2021. URL: https://jangafx.com/software/embergen/.

[Jas76] Zbigniew D. Jastrzebski. “Nature and properties of engineering materials”. In: (Jan. 1976). URL: https://www.osti.gov/biblio/7341495.

[Jia+15] Chenfanfu Jiang et al. “The Affine Particle-in-Cell Method”. In: *ACM Trans. Graph.* 34.4 (July 2015). ISSN: 0730-0301. DOI: 10.1145/2766996. URL: https://doi.org/10.1145/2766996.

[Jia+16] Chenfanfu Jiang et al. “The material point method for simulating continuum materials”. In: *ACM SIGGRAPH 2016 Courses*. 2016, pp. 1–52.

[Kel13] Piaras Kelly. “Solid Mechanics”. In: *Part II, Lecture notes, The University of Auckland* (2013).

[Kot21] Grant Kot. *Liquid Crystal*. 2021. URL: https://www.grantkot.com/.

[LC87] William E. Lorensen and Harvey E. Cline. “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’87. New York, NY, USA: Association for Computing Machinery, 1987, pp. 163–169. ISBN: 0897912276. DOI: 10.1145/37401.37422. URL: https://doi.org/10.1145/37401.37422.

[Liu15] Chien Liu. “Discretizing the Weak Form Equations”. In: (Feb. 2015). URL: https://www.comsol.com/blogs/discretizing-the-weak-form-equations.

[Mac+14] Miles Macklin et al. “Unified Particle Physics for Real-Time Applications”. In: *ACM Trans. Graph.* 33.4 (July 2014). ISSN: 0730-0301. DOI: 10.1145/2601097.2601152. URL: https://doi.org/10.1145/2601097.2601152.

[Min21] Patrick Min. *binvox*. http://www.patrickmin.com/binvox. 2004 - 2021.

[Mon92] J. J. Monaghan. “Smoothed particle hydrodynamics.” In: 30 (Jan. 1992), pp. 543–574. DOI: 10.1146/annurev.aa.30.090192.002551.

[Mül+05] Matthias Müller et al. “Meshless Deformations Based on Shape Matching”. In: *ACM Trans. Graph.* 24.3 (July 2005), pp. 471–478. ISSN: 0730-0301. DOI: 10.1145/1073204.1073216. URL: https://doi.org/10.1145/1073204.1073216.

[Nea04] Andrew Nealen. “An As-Short-As-Possible Introduction to the Least Squares, Weighted Least Squares and Moving Least Squares Methods for Scattered Data Approximation and Interpolation”. In: (2004).

[PH90] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990. ISBN: 1558800698.

[SB12] Eftychios Sifakis and Jernej Barbic. “FEM Simulation of 3D Deformable Solids: A Practitioner’s Guide to Theory, Discretization and Model Reduction”. In: *ACM SIGGRAPH 2012 Courses*. SIGGRAPH ’12. Los Angeles, California: Association for Computing Machinery, 2012. ISBN: 9781450316781. DOI: 10.1145/2343483.2343501. URL: https://doi.org/10.1145/2343483.2343501.

[Sch14] Bettina Schieche. “The Strength of the Weak Form”. In: (Apr. 2014). URL: https://www.comsol.com/blogs/strength-weak-form/.

[SCS94] D. Sulsky, Z. Chen, and H.L. Schreyer. “A particle method for history-dependent materials”. In: *Computer Methods in Applied Mechanics and Engineering* 118.1 (1994), pp. 179–196. ISSN: 0045-7825. DOI: https://doi.org/10.1016/0045-7825(94)90112-0. URL: http://www.sciencedirect.com/science/article/pii/0045782594901120.

[Set+14] Rajsekhar Setaluri et al. “SPGrid: A Sparse Paged Grid Structure Applied to Adaptive Smoke Simulation”. In: *ACM Trans. Graph.* 33.6 (Nov. 2014). ISSN: 0730-0301. DOI: 10.1145/2661229.2661269. URL: https://doi.org/10.1145/2661229.2661269.

[SKB08] Michael Steffen, Robert Kirby, and Martin Berzins. “Analysis and reduction of quadrature errors in the material point method (MPM)”. In: *International Journal for Numerical Methods in Engineering* 76 (Nov. 2008), pp. 922–948. DOI: 10.1002/nme.2360.

[Sof21] Valve Software. *Steam Hardware Survey*. May 2021. URL: https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam.

[SSS20] Yunxin Sun, Tamar Shinar, and Craig Schroeder. "Effective time step restrictions for explicit MPM simulation". In: *Computer Graphics Forum* 39.8 (2020), pp. 55–67. DOI: https://doi.org/10.1111/cgf.14101. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14101. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14101.

[Sta01] Jos Stam. "Stable Fluids". In: *ACM SIGGRAPH 99* 1999 (Nov. 2001). DOI: 10.1145/311535.311548.

[Sto+13] Alexey Stomakhin et al. "A Material Point Method for Snow Simulation". In: *ACM Trans. Graph.* 32.4 (July 2013). ISSN: 0730-0301. DOI: 10.1145/2461912.2461948. URL: https://doi.org/10.1145/2461912.2461948.

[Sul20] Tim Sullivan. "A brief introduction to weak formulations of PDEs and the finite element method". In: (June 2020). URL: https://warwick.ac.uk/fac/sci/hetsys/studentinformation/induction/mathsinduction/pde/pde.pdf.

[SZS95] Deborah Sulsky, Shi-Jian Zhou, and Howard L. Schreyer. "Application of a particle-in-cell method to solid mechanics". In: *Computer Physics Communications* 87.1 (1995). Particle Simulation Methods, pp. 236–252. ISSN: 0010-4655. DOI: https://doi.org/10.1016/0010-4655(94)00170-7. URL: http://www.sciencedirect.com/science/article/pii/0010465594001707.

[Tie+17] Roel Tielen et al. "A High Order Material Point Method". In: *Procedia Engineering* 175 (2017). Proceedings of the 1st International Conference on the Material Point Method (MPM 2017), pp. 265–272. ISSN: 1877-7058. DOI: https://doi.org/10.1016/j.proeng.2017.01.022. URL: http://www.sciencedirect.com/science/article/pii/S187770581730022X.

[Vau+20] Alban de Vaucorbeil et al. "Material point method after 25 years: theory, implementation and applications". In: Advances in Applied Mechanics (2020).

[Wan+20] Xinlei Wang et al. "A Massively Parallel and Scalable Multi-CPU Material Point Method". In: *ACM Trans. Graph.* 39.4 (July 2020). ISSN: 0730-0301. DOI: 10.1145/3386569.3392442. URL: https://doi.org/10.1145/3386569.3392442.

[XZY17] Xiangyun Xiao, Shuai Zhang, and Xubo Yang. “Real-Time High-Quality Surface Rendering for Large Scale Particle-Based Fluids”. In: *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D ’17. San Francisco, California: Association for Computing Machinery, 2017. ISBN: 9781450348867. DOI: 10.1145/3023368.3023377. URL: https://doi.org/10.1145/3023368.3023377.

[ZB05] Yongning Zhu and Robert Bridson. “Animating Sand as a Fluid”. In: *ACM Trans. Graph.* 24.3 (July 2005), pp. 965–972. ISSN: 0730-0301. DOI: 10.1145/1073204.1073298. URL: https://doi.org/10.1145/1073204.1073298.

NTNU

Kunnskap for en bedre verden