Neural Temporal Adaptivity for the Material Point Method Accelerated on the GPU

by

Alexandre Sirois-Vigneux

School of Computer Science
McGill University, Montréal QC
June 2023

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of
Master of Computer Science

Résumé

L'animation basée sur la physique joue un role important dans les environnements virtuels en films et jeux vidéo. Malgré d'importants progrès récents, les algorithmes les plus avancés exécutés sur du matériel de pointe rencontrent toujours des difficultés à atteindre le niveau de réalisme et d'interactivité attendu. Nous proposons une nouvelle méthode à l'intersection de l'apprentissage automatique et de l'infographie pour accélérer la méthode de point de matériau (MPM). Notre approche utilise un réseau de neurones convolutifs 3D pour identifier de manière asynchrone les régions actives de la simulation. Les résultats expérimentaux obtenus témoignent de gains importants en performance dans plusieurs scénarios pratiques sans détérioration dans les pires cas. Notre prototype est entièrement implémenté sur carte graphique, nous permettant de tirer profit de la nature parallèle des algorithmes. Nous espérons que cette adaptabilité temporelle neuronale ouvre la voie au développement d'autres méthodes hybrides sophistiquées.

Abstract

Physically-based animation is crucial for virtual environments in films and video games. Despite recent progress, even the most advanced algorithms running on top-tier hardware face challenges in achieving the desired level of interactivity and realism. We propose a new method at the intersection of machine learning and computer graphics to accelerate the material point method (MPM). Our approach uses a 3D convolutional neural network to asynchronously identify active regions of the simulation. Our experimental results show significant performance improvements in practical scenarios with no deterioration in worst cases. Our prototype is implemented entirely on the GPU, leveraging the algorithms' parallel nature. We hope this neural temporal adaptivity opens new exciting opportunities for more sophisticated hybrid methods.

À mon épouse Audrey,

mes parents Richard et Marie,

et ma soeur Ève

Preface

Before enrolling in academia to study mathematics, computer science, computer graphics, and machine learning, I worked as a technical artist in the visual effects industry for nearly a decade. During this period, I became aware of the capabilities and limitations of production tools, which sparked my desire to gain a more profound understanding of the technical aspects and intricacies of visual effects applications. Consequently, I decided to pursue further education in the fields mentioned above.

My fascination with computer animation, especially physically-based simulation, has been a primary motivator throughout my career. The recent upsurge in the popularity of machine learning has opened up numerous opportunities for research and improvement in computer graphics. Even before the intersection of computer graphics and machine learning became a trending topic, I was thrilled about the possibilities. This thesis aims to investigate an aspect of this intersection and its potential applications to visual effects production.

Acknowledgements

Before we delve into it, I want to take a moment to express my sincere gratitude to those who have made this work at all possible. Their tireless support, encouragement, and guidance have been instrumental in bringing this work to fruition.

First and foremost, I want to thank the love of my life, my wife, Audrey. Throughout the past years, she has been a constant source of inspiration, motivation, and positivity. When doubts and questions arose, she always had the words to keep me grounded and focused on what truly mattered. Her gift for seeing the glass half full in all situations has been an endless source of hope, and I am forever grateful for her love and support. I also want to extend my heartfelt thanks to my father, Richard, for sparking my interest in technology early on and for his everlasting devotion to our long late-night phone calls. His insights and experience have been invaluable, and I will always cherish those memories. My mother, Marie, had consistently shown unconditional support for my work, even when it meant asking many questions in exchange for very few satisfying answers. Her unshakable faith in me has been a constant source of determination. To my younger sister, Ève, thank you for showing interest in my work and being such a great model of strength and perseverance these past few years. Your support and encouragement have meant the world to me.

I also want to express my gratitude to Arnaud Schoentgen, who provided early insights for this project and generously shared his time and expertise. His passion for particle-based simulation has been an inspiration, and I hope we can collaborate further in the future. To my advisor, Pierre Poulin, thank you for the thought-provoking discussions and brainstorming sessions about the potential directions of the project. I also want to acknowledge the superhuman speed at which the different sections of this thesis were reviewed for the last month of redaction; your dedication and attention to detail have been impressive. I also want to thank my advisor, Derek Nowrouzezahrai, for providing the means and financial resources for me to complete this Master's degree. Your trust in my ability to accomplish such an undertaking independently has been appreciated. I am thankful for the opportunity to learn and grow with such liberty. Finally, I want to thank

my former colleagues in the visual effects industry, who have closely monitored my academic evolution on social media and provided support and encouragement along the way. I hope we can soon work together again and push the boundaries of what is shown on the big screen to new heights.

To all of you, thank you for believing in me and for your unwavering support. This work would not have been possible without you, and I am eternally grateful for your contributions to my journey.

Contents

List of Figures

List of Symbols and Acronyms

General

b	A scalar (integer or real)
$\mathbf{b}$	A vector
$\mathbf{B}$	A matrix
$\mathbf{B}$	A tensor

Physically-Based Animation

MPM	Material point method
FLIP	Fluid implicit particle
SPH	Smoothed particle hydrodynamics
PBD	Position-based dynamics
RBD	Rigid-body dynamics
SDF	Signed distance field
MLS	Moving least squares
$\mathcal{S}$	Scene
$\mathcal{P}$	Particle system
p	Particle / Fluid pressure
$\mathcal{G}$	Eulerian background grid
i	Voxel linear index
Δx	Voxel size
t	Time
Δt	Timestep
n	Timestep index

$\mathcal{T}$	Tile grid
τ	Tile linear index
$\mathbf{u}, \mathbf{v}$	Velocity vector
$\mathbf{g}$	Body forces
$\mathbf{x}$	Deformed state / Position vector
$\mathbf{X}$	Rest state
$\boldsymbol{\phi}$	Deformation mapping
ϕ	Signed distance level set
$\boldsymbol{\sigma}$	Cauchy stress tensor (second order)
$\mathbf{P}$	First Piola-Kirchoff stress
ρ	Density
V	Volume
m	Mass
$\mathbf{F}$	Deformation gradient
$\mathbf{F}_E$	Deformation gradient elastic component
$\mathbf{F}_P$	Deformation gradient plastic component
J	Deformation gradient determinant
$\mathbf{C}$	Affine velocity matrix
N	Interpolation function
w	Interpolation weights
Ψ	Energy density function
$\mathbf{f}$	Force vector
λ	Lamé parameter
μ	Lamé parameter / Friction coefficient
E	Young's modulus
ν	Poisson's ratio / Fluid kinematic viscosity
κ	Material hardening under compression
γ	Activation state
δ	Speed threshold
Ω	Domain scale

Machine Learning

ML	Machine learning
MLP	Multilayer perceptron

CNN	Convolutional neural network
ReLU	Rectified linear unit
RF	Receptive field
$\mathbf{f}$	Model
$\boldsymbol{\theta}$	Trainable parameters
$(\mathbf{x}, \mathbf{y})$	Dataset example
$\mathbf{h}$	Hidden layer
$\mathbf{a}$	Layer pre-activation
$\mathbf{W}$	Layer weights
$\mathbf{b}$	Layer biases
σ	Sigmoid function
l	Loss function
Ω	Regularization term
η	Learning rate
$\odot$	Hadamard product (element-wise)
α	Signal multiplier

Software Engineering

CPU	Central processing unit (host)
GPU	Graphical processing unit (device)
GPGPU	General-purpose computing on GPU
CUDA	Compute unified device architecture
VDB	Voxel database
HDA	Houdini digital asset
GUI	Graphical user interface
SoA	Structure of arrays
AoS	Array of structures

Chapter 1

Introduction

The natural world is characterized by constant changes and unpredictability. This dynamism has long captivated human curiosity and creativity, leading to a desire to interact with and influence our environment. In addition to the practical applications of physics-based simulations for science and engineering, the appeal of simulating physical phenomena lies in the ability to experiment and play with such systems inconsequentially. These solvers offer a controlled environment in which we can manipulate variables, repeat events indefinitely, and even explore aspects of reality way beyond our grasp. This drive to recreate and control the world around us has been a fundamental part of human expression throughout history. In the same way painters of the Renaissance sought to achieve realistic depictions of the natural world, modern engineers and artists endeavor to reconstruct virtual replicas of some of the most mesmerizing phenomena occurring on this planet.

As entertainment continues to evolve, it is becoming increasingly clear that virtual worlds will play an integral role in shaping our future. The accurate simulation of physics will be a key factor in creating truly immersive environments to satisfy the demands of audiences who crave dynamic realism. Virtual worlds will act as infinite playgrounds enabling unprecedented levels of creative freedom where cinematic storytelling is no longer held back by technical limitations. Technology should soon exclusively serve as a conduit for great creative minds to transport audiences through their unique visions.

The field of physically-based animation can be divided into two distinct research groups:

those who prioritize achieving perfect realism at any cost, quite literally with regard to time and financial resources, and those who strive to achieve the holy grail of real-time physics simulations. The methods employed by these two groups often vastly differ, but they are not mutually exclusive. As hardware capabilities continue to improve, we frequently observe techniques once only viable for offline applications becoming integrated into real-time technology. There are also instances of the film industry adopting a technology initially developed for real-time use thanks to its scalability and strong physical footing, as demonstrated by the *Bullet Physics Library* [Cou23] for rigid-body dynamics. Ultimately, these two fields should be reconciled when advanced algorithms paired with cutting-edge hardware will allow perfectly believable physics to be simulated in real time, regardless of its complexity. Until that day arrives, compromises must be made to pursue either of these goals.

Specialized solvers have been developed to handle specific materials and improve the efficiency of physics engines. These solvers are built around material-specific shortcuts that often lack general accuracy for other material types. Furthermore, not all materials can be accurately represented with the same topology. For instance, fluid simulations are best described on grids of voxels, while particle systems are more adequate for materials such as sand and other granular media. Despite many past attempts at developing the ultimate multi-physics solver [MMCK14] that is both general and computationally efficient, such solutions have remained imperfect. However, the material point method (MPM) [SZS95] is a promising technique due to its ability to model a wide range of materials and to enable rich interactions between them. Unfortunately, it is not renowned for being lightweight or computationally efficient. This delicate balance between efficiency and generality is not unique to computer graphics but prevalent in computer science as a whole.

Even though offline simulation offers more flexibility, meeting production deadlines requires performance and user interactivity considerations. To achieve this, developers must identify all possible approximations to improve performance without compromising visual accuracy. One popular approach is adaptive discretization, which can be applied spatially and temporally. However, this often leads to more complex algorithms and data structures while also being susceptible to corner cases that perform worse than nonadaptive methods. Ideally, an adaptive approach

should provide a decent average performance improvement, a tolerable worst case, while not being overly challenging to implement.

Algorithm improvements in physically-based animation often stem from critical observations of the discrepancies between what is visually perceived and what is computed. For instance, a stack of bricks in free fall is visually more captivating than a static stack, despite the latter being more computationally expensive to simulate. To address this issue, techniques such as partial deactivation and object merging [CBK20] have been developed to reduce the computational burden on physics engines. However, the challenge is to accurately and seamlessly detect when to transition out of these optimizations without sacrificing visual fidelity. The differences between perceived and actual physical accuracy present endless acceleration opportunities in various computer graphics subfields.

The impressive proliferation of machine learning (ML) advances has enabled formerly insurmountable computer problems to be solved with ease. In the context of supervised learning, a model minimizes its prediction error given a training set by adjusting its internal parameters. Despite the disadvantage of these opaque "black box" models that are difficult to interpret and learn from, this framework can find locally optimal solutions without requiring domain-specific knowledge. In the ideal scenarios, these models can even discover novel, faster, and more precise ways of solving a problem than traditional methods.

Although initially developed for real-time rendering, the graphical processing unit (GPU) has since found extensive use in non-interactive contexts such as scientific computing and physics simulation. The highly parallel nature of many algorithms allows for efficient ports to massively parallel architectures with minimal modification to existing implementations. When adequately executed, such GPU ports often lead to staggering performance improvements, taking us a step closer to harmonizing the online and offline worlds.

1.1 Overview

The main focus of this thesis is the acceleration of a material point method (MPM) solver using a neural temporally adaptive approach on the GPU. The temporal adaptivity is determined at

runtime by querying a 3D convolutional neural network (CNN) within the solver, hence the term "neural". We start by laying down the theoretical foundations of MPM in Section 2.2, drawing parallels with fluid dynamics in Section 2.1. We then delve into the basics of deep neural networks in Section 2.3, as well as general-purpose computing on GPU in Section 2.4. Once these foundational concepts are covered, we outline our methodology in Chapter 3, where we study the prototype developed as part of this work. The aspect of generating the ground-truth data to train our CNN is discussed first in Section 3.1. Then we move to the architectural design of the CNN in Section 3.2. We provide specific details regarding the training of the machine-learning model in Section 3.3 and outline its integration inside the solver in Section 3.4. The introduction of the particle scheduler in Section 3.5 allows the information inferred from the CNN to positively impact the execution of the solver. The chapter ends with practical implementation considerations specific to our prototype in Section 3.6. We later present results obtained with our method and evaluate performance on them in Chapter 4. We conclude by discussing the strengths and weaknesses of the proposed approach in the context of visual effects production as well as potential improvements in future work in Chapter 5. Finally, we revisit our accomplishments from a broader perspective in Chapter 6.

Chapter 2

Background

2.1 Foundations of Fluid Dynamics

From storms to fire to oceans, fluid phenomena are ubiquitous to our environment. Humans have always been fascinated by the motions and patterns described by fluids. Replicating such breathtaking phenomena on a silicon chip has been one of the long-standing goals of the graphics community. The motion of fluids is typically modeled using the well-known *Navier-Stokes equations*, as expressed by Bridson [Bri15] as

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} + \frac{1}{\rho} \nabla p = \mathbf{g} + \nu \nabla \cdot \nabla \mathbf{u}, \quad \text{and} \tag{2.1}$$

$$\nabla \cdot \mathbf{u} = 0. \tag{2.2}$$

In this pair of equations, $\mathbf{u}$ denotes the velocity vector, t the time, $\nabla \mathbf{u}$ the velocity gradient, ρ the fluid density, ∇p the pressure gradient, $\mathbf{g}$ the vector of body forces (such as gravity), $\nabla \cdot \nabla \mathbf{u}$ the velocity *Laplacian*, ν the kinematic viscosity, and $\nabla \cdot \mathbf{u}$ the velocity divergence (Figure 2.1).

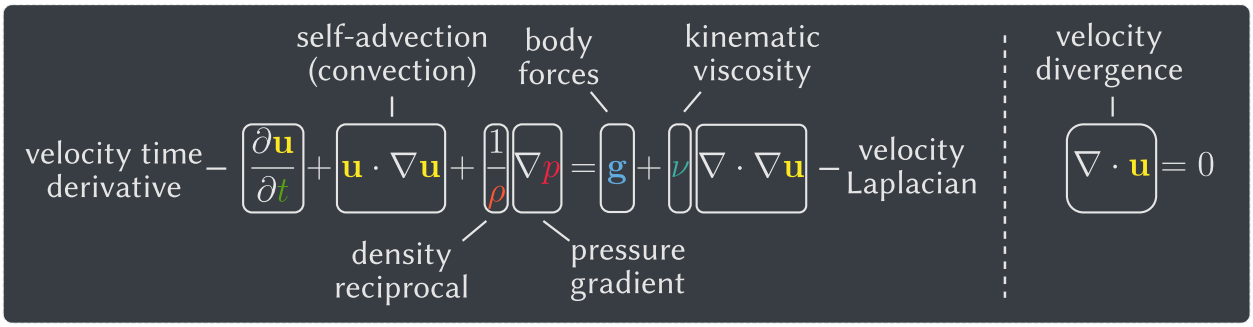


Figure 2.1. Breakdown diagram of the different terms of the *Navier-Stokes equations*.

Equation 2.1 is called the momentum equation, and Equation 2.2 represents the incompressibility condition. It is also possible to rewrite the momentum equation using the material derivative

$$\frac{Dq}{Dt} = \frac{\partial q}{\partial t} + \mathbf{u} \cdot \nabla q, \tag{2.3}$$

where the changes of quantity q at a fixed point in space will be affected by how it evolves over time t and the dot product between how it is changing in space ∇q and how much it is moved (or advected) in space by $\mathbf{u}$. Thus, we arrive at this new formulation

$$\frac{D\mathbf{u}}{Dt} + \frac{1}{\rho}\nabla p = \mathbf{g} + \nu \nabla \cdot \nabla \mathbf{u} \tag{2.4}$$

where $\frac{D\mathbf{u}}{Dt}$ is often referred to as self-advection. To simplify things further, we drop the viscosity term $\nu \nabla \cdot \nabla \mathbf{u}$, which can be ignored for many real-world materials, and we rearrange the terms using the definition of the density $\rho = \frac{m}{V}$ with mass m over volume V to get the form

$$m\frac{D\mathbf{u}}{Dt} = m\mathbf{g} - V\nabla p \tag{2.5}$$

where $\frac{D\mathbf{u}}{Dt}$ acts as the acceleration, $m\mathbf{g}$ the external forces, and $-V\nabla p$ the internal forces, which gives us Newton's $m\mathbf{a} = \mathbf{f}$. This last formulation is not useful in practice but provides intuition to better understand the equations [Bri15].

2.1.1. Lagrangian vs. Eulerian Simulation. The dynamics of a continuum, whether fluid or solid, can be modeled from two perspectives: the Lagrangian and Eulerian viewpoints. The

Lagrangian approach views the continuum as a system of particles moving through space, where each particle carries a set of properties such as mass and velocity. In this context, the particles can be regarded as small elements of fluid. The Eulerian perspective tracks the continuum at fixed positions, most often on a grid, with a static topology; here, the fluid moves past the fixed points rather than advecting them along with itself.

Although less intuitive at first, the Eulerian perspective offers several benefits over the Lagrangian approach. In particular, calculating derivatives is straightforward using finite differences on a regular grid. By contrast, purely Lagrangian techniques such as smoothed particle hydrodynamics (SPH) [Mon92] and position-based dynamics (PBD) [MHHR07] require constant neighborhood queries to evaluate the same quantities and derivatives, which can lead to performance issues due to changing topology. The Eulerian approach is not perfect either since it may be plagued with excessive dissipation, as attributes are repeatedly interpolated to different grid nodes instead of being advected.

One concept that allows us to link the two viewpoints together is the material derivative expressed in Equation 2.3. We can compute the derivative of a fluid quantity q using the chain rule

$$\begin{aligned}\frac{dq(t,\mathbf{x}(t))}{dt} &= \frac{\partial q}{\partial t} + \frac{\partial q}{\partial \mathbf{x}_x}\frac{d\mathbf{x}_x}{dt} + \frac{\partial q}{\partial \mathbf{x}_y}\frac{d\mathbf{x}_y}{dt} + \frac{\partial q}{\partial \mathbf{x}_z}\frac{d\mathbf{x}_z}{dt} \\ &= \frac{\partial q}{\partial t} + \nabla q \cdot \mathbf{u} \\ &= \frac{Dq}{Dt},\end{aligned} \tag{2.6}$$

which yields the material derivative. The Lagrangian term $\frac{dq(t,\mathbf{x}(t))}{dt}$ describes how the quantity q changes with respect to time t without considering its spatial displacement $\mathbf{u}$. In contrast, the Eulerian term $\frac{\partial q}{\partial t}$ represents the derivative evaluated at a fixed point in space accompanied by the corrective term $\nabla q \cdot \mathbf{u}$ accounting for changes due to fluid motion.

In practice, both viewpoints are used in many hybrid methods, such as fluid implicit particle (FLIP) [BKR88] and its extension to solid mechanics, the material point method (MPM) [SZS95], one of the central themes of this thesis.

2.1.2. Point Cloud Surface Extraction. Lagrangian solvers prevent dissipation of the material attributes by keeping track of them on a set of particles. Although this representation has several benefits, it may not be ideal when visualizing the material's volume or utilizing it as a collider. In such cases, a common strategy is to generate a signed distance field (SDF) of the particle system, where the level set $\phi(\mathbf{x}) = 0$ represents the material's surface (Figure 2.2).

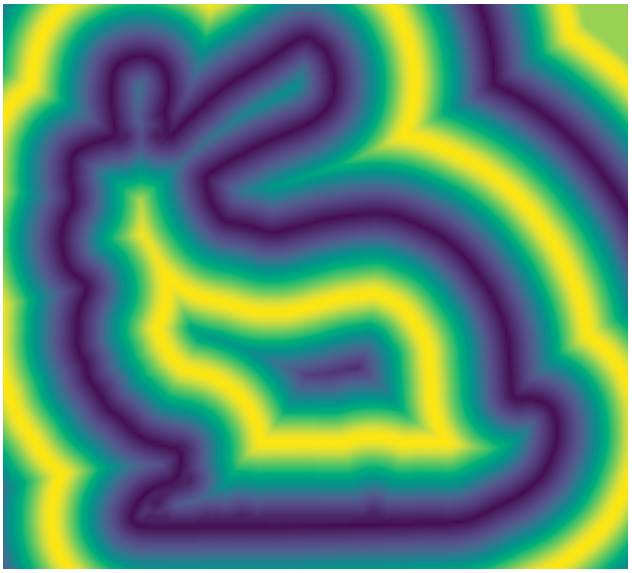


Figure 2.2. 2D array containing signed distance information (SDF); each node of the field contains the distance to the closest point on the surface of the Stanford bunny. Distance inside the object is negative $\phi < 0$ while distance outside is positive $\phi > 0$, hence the "signed" in signed distance field. The distance is visualized with the viridis color mapping in mirrored repeat mode. The purple outline of the bunny corresponds to the the SDF isocontour $\phi = 0$.

The first step in constructing an SDF is to compute the distance to particles for each grid cell containing a particle, while considering the particles' radius. This distance is then propagated through the rest of the grid using a particular order. At this point, the SDF inside the material represents the distance to internal particles, not the surface itself. Therefore, we need to recompute the SDF starting from the cells containing a change in sign by solving the Eikonal equation $||\nabla\phi|| = 1$. Finally, the newly computed information close to the surface is propagated through the rest of the grid, again using a particular order.

The fast marching [Set96, Tsi95] and fast sweeping [Zha05] algorithms are widely used to

propagate SDF information through the grid. The fast marching algorithm runs in $O(n \log n)$ time, where n is the number of grid points (voxels). It requires an additional priority queue to define the order in which grid nodes are visited, but has the advantage of working on sparse grid structures. The fast sweeping algorithm runs in $O(n)$ time and is simpler to implement, but it is limited to dense grids. Hybrid solutions are often used to benefit from the strengths of both techniques. More details on both algorithms can be found in the survey by Jones et al. [JBS06].

When a well-behaved SDF is obtained, it is straightforward to convert it into a triangle mesh using the Marching Cubes algorithm [LC87], which can be more convenient for rendering.

2.1.3. Temporal Adaptivity. Physically-based animation involves numerically integrating a system's equations of motion over time. This integration process is often approximated with a discrete timestep, which can significantly impact a simulation's accuracy and computational efficiency. In computer graphics, we usually try to choose the largest timestep that maintains system stability and reasonable accuracy, often derived from the most nonlinear event in the simulation. However, in certain scenarios, such as tall stacks of rigid bodies, highly compressed fluids, or solids undergoing significant deformation, a tiny timestep may be necessary to prevent the system from entering an invalid state. To address this issue, researchers have developed several temporally adaptive methods that can be divided into three categories: adaptive global time stepping, adaptive local time stepping, and adaptive freezing. We briefly discuss the main characteristics of each approach following the review by Manteaux et al. [MWN+17].

Adaptive global time stepping. One of the most popular techniques used in the context of continuous media like fluids and elastic solids is the Courant-Friedrichs-Lewy (CFL) condition [CFL67]

$$\Delta t \leq \frac{C\Delta x}{||\mathbf{v}||}, \tag{2.7}$$

which states that timestep Δt must be small enough such that the material moving at speed $||\mathbf{v}||$ does not travel more than a certain fraction C of an Eulerian grid cell size Δx per timestep. After each timestep, the CFL condition can be assessed to estimate the next time discretization. However, some simulations may exhibit highly varying timesteps, leading to challenges in main-

taining numerical stability. To address this, Ihmsen et al. [BET+10] developed a mechanism that gradually adjusts the timestep and rewinds the simulation when sudden changes are detected. Additionally, Bridson et al. [BMF03] proposed a time integration scheme that enables verlet and leapfrog integrations to remain second-order accurate even when using adaptive time stepping.

Adaptive local time stepping. Different regions within the simulation domain often have different timestep requirements. For instance, while regions featuring highly nonlinear events, such as complex collisions, may require finer timesteps, others may not. This observation motivates the use of regional time stepping, where the domain is partitioned into blocks that are integrated at their own pace. Explicit integration is usually more suitable for this approach, where the CFL condition can be used as the local time stepping criterion [DG96]. Fang et al. [FHHJ18] proposed a regional time stepping approach with the material point method (MPM), where each block is moved forward in time according to "the speed of sound and the element characteristic length". Their method still requires the presence of so-called ghost or buffer particles to provide particles running at a finer timestep the appropriate boundary conditions from neighboring blocks. This approach also incurs a high overhead cost associated with the scheduler, which limits its efficiency. Nevertheless, it offers excellent energy conservation, which is also true for asynchronous variational integrators (AVIs) [LMOW04]. An alternative approach involves switching to implicit integration only in regions where explicit integration is not accurate enough [FSH11].

Adaptive freezing. Degrees of freedom that are considered unimportant are frozen to focus computational resources where it matters most. This can be considered as taking giant timesteps on temporarily deactivated elements of the simulation. The technique is beneficial in simulations where most objects are static. The reactivation procedure is critical to ensure that deactivation does not influence the simulation outcome. It is commonly used in rigid-body dynamics (RBD) [BET14], where heuristics based on kinematic energy or velocity are used to activate or deactivate bodies [MS01]. Goswami et al. [GPB+11] have developed a simple heuristic to define the active status of particles with SPH. According to their heuristic, particles that are close to a boundary or have a speed above a certain threshold are considered active, while the rest are considered passive if they are not neighbors to fast or boundary particles.

2.1.4. Material Modeling Limitations. While employing a fluid solver to simulate solid-like materials is possible, this strategy is not ideal [SSC+13]. The use of fluid solvers in simulating such materials has been explored by Zhu and Bridson [ZB05], who demonstrated successful simulations of sand as an incompressible fluid, and McAdams et al. [MSW+09], who applied a similar approach to simulate hair collision. Both methods are based on the fluid implicit particle (FLIP) [BKR88] and are not suitable for stiff solids. Additionally, a two-way coupled solver can combine solid and fluid dynamics; however, this does not address the case of materials that continuously vary in phase, such as snow, which can behave simultaneously as a fluid and a solid. The material point method (MPM) [SZS95] extends FLIP to solid mechanics, handling a wide range of materials, including fluids, elastic solids, granular media, and others.

In contrast to fluid dynamics, where the pressure field is the primary quantity of interest for describing material behavior, solid mechanics necessitates an additional variable to characterize deformation. In particular, we must account for the material's response to different types of deformations, such as shear and nonuniform scaling, even in situations where the local volume remains constant. This additional variable is known as stress, which refers to the force exerted on a material as it undergoes deformation. Stress is intimately tied to strain, which describes the material's deformation under the influence of stress.

Stress has multiple definitions, one being the Cauchy stress defined in 3D space, as the nine components second-order tensor

$$\boldsymbol{\sigma} = \begin{pmatrix} \sigma_{1,1} & \sigma_{1,2} & \sigma_{1,3} \\ \sigma_{2,1} & \sigma_{2,2} & \sigma_{2,3} \\ \sigma_{3,1} & \sigma_{3,2} & \sigma_{3,3} \end{pmatrix} \tag{2.8}$$

where each component represents a force per unit area acting in a particular direction, that fully describes the deformation state of a material at a particular point. The rest state corresponds to a 3×3 identity matrix that can be thought of as a frame in 3D space where the local volume change can easily be computed by taking the determinant of the tensor.

2.2 Material Point Method

In the field of continuum mechanics, material deformation can be expressed as the difference between its rest state $\mathbf{X}$ and its deformed state $\mathbf{x} = \boldsymbol{\phi}(\mathbf{X})$, where $\boldsymbol{\phi}$ denotes the deformation mapping between the two configurations. The gradient of $\boldsymbol{\phi}$ with respect to $\mathbf{X}$ is known as the deformation gradient $\mathbf{F} = \partial\boldsymbol{\phi}/\partial\mathbf{X}$. The behavior of $\boldsymbol{\phi}$ is dictated by the Eulerian governing equations

$$\frac{D\rho}{Dt} + \rho\nabla\cdot\mathbf{v} = 0, \quad \text{conservation of mass} \tag{2.9}$$

and

$$\rho\frac{D\mathbf{v}}{Dt} = \nabla\cdot\boldsymbol{\sigma} + \rho\mathbf{g}, \quad \text{conservation of momentum,} \tag{2.10}$$

where ρ is the density, $\mathbf{g}$ the external body forces (such as gravity), $\boldsymbol{\sigma}$ the Cauchy stress (Equation 2.8), and D/Dt the material derivatives (Equation 2.3).

The core idea behind the material point method (MPM) is based on the Lagrangian perspective of tracking physical quantities such as mass m_p, velocity $\mathbf{v}_p$, and deformation gradient $\mathbf{F}_p$ using material particles p. This simplifies the spatial discretization of the material and avoids dissipation during advection. However, this representation is not suitable for computing stress-based forces. To address this, an Eulerian grid is employed as a *scratch pad* to compute the material internal forces. This Eulerian grid discretizes the term $\nabla\boldsymbol{\sigma}$ in accordance with the weak form of the finite element method (FEM).

At each timestep, particles' mass and momentum are transferred to the Eulerian grid. The grid is then solved, followed by the reverse transfer of the velocity information along with its affine velocity matrix from the grid back to the particles before the advection step. These transfers use an affine particle-in-cell (APIC) scheme [JSS+15] and necessitate interpolations between the grid nodes and particles. The finite element perspective is helpful here, wherein the Eulerian grid assumes the role of a mesh in FEM while the particles correspond to quadrature points.

The interpolation function, denoted as N, provides the weights establishing an association between a specific grid node i and a corresponding particle p. The magnitude of these weights should be substantial when i and p are in close proximity while diminishing as the distance

between them increases. The selection of the interpolation function N entails a tradeoff between the desired smoothness, computational efficiency, and the extent of its stencil. It is worth noting that because of cell-crossing instability, the linear kernel commonly employed in FLIP solvers cannot be utilized with MPM. Hence, alternatives such as quadratic and cubic B-Splines are preferred. More specifically, the quadratic B-Spline interpolation function [JST+16] is defined as

$$N(x) = \begin{cases} \frac{3}{4} - x^2 & 0 \leq |x| < \frac{1}{2} \\ \frac{1}{2}(\frac{3}{2} - |x|)^2 & \frac{1}{2} \leq |x| < \frac{3}{2} \\ 0 & \frac{3}{2} \leq |x|. \end{cases} \tag{2.11}$$

This function can be evaluated in d-dimensions for the particle position $\mathbf{x}_p$ and grid node position $\mathbf{x}_i$ of size Δx using the product of the individual one-dimensional evaluations [SKB08] such that

$$N_i(\mathbf{x}_p) = N\left(\frac{1}{\Delta x}(x_p - x_i)\right) N\left(\frac{1}{\Delta x}(y_p - y_i)\right) N\left(\frac{1}{\Delta x}(z_p - z_i)\right). \tag{2.12}$$

We utilize the concise notation $w_{ip} = N_i(\mathbf{x}_p)$, as introduced in [SSC+13], where w_{ip} is the interpolation weight associated with particle p and grid node i.

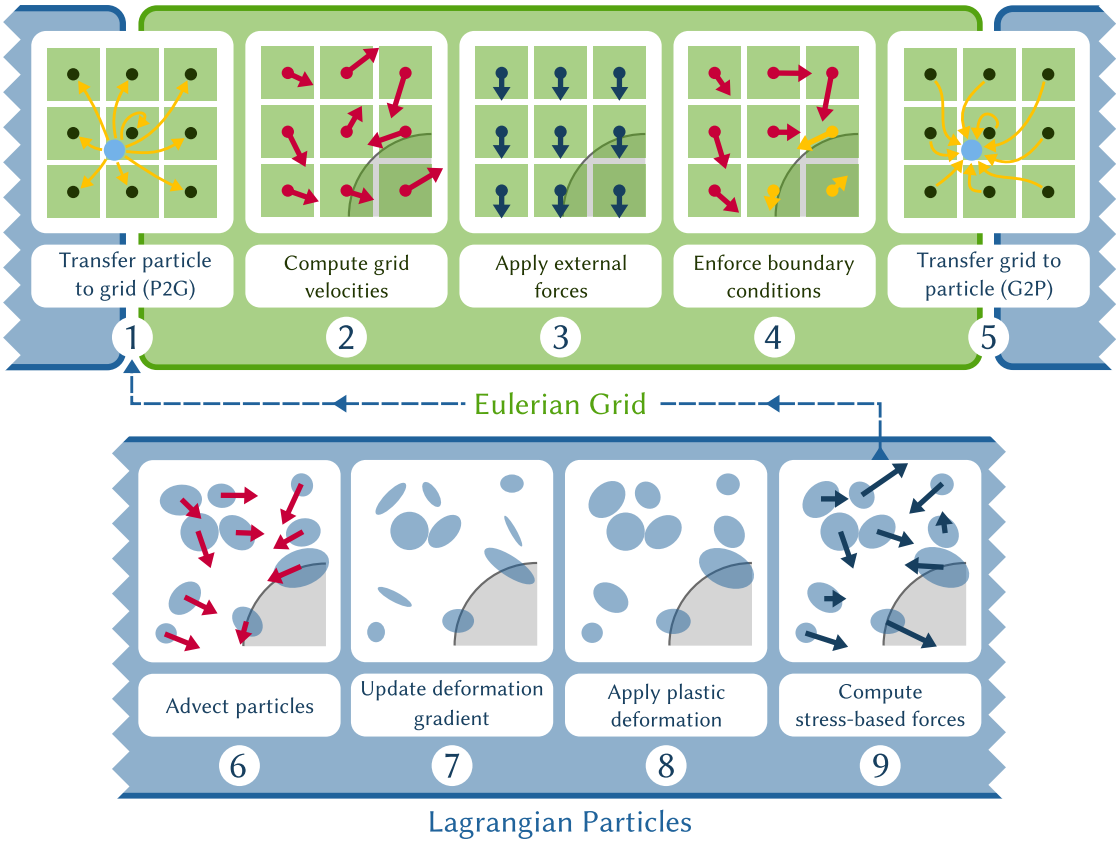


Figure 2.3. Overview of the material point method (MPM) algorithm.

2.2.1. The Algorithm. Below, we present the main steps of the MPM algorithm using explicit integration accompanied by its visual counterpart (Figure 2.3). Although similar to traditional MPM [SZS95], the algorithm described here is the moving least squares MPM (MLS-MPM) [HFG+18], which offers superior computational efficiency over the original method. Explicit integration is used to keep the complexity to a minimum and to take advantage of the fusion of the affine momentum and particle force contribution, as shown in Equation 2.24. The two algorithmic improvements of MLS-MPM will be further discussed as we delve into the theory.

(1) **Transfer particle to grid (P2G):** Particles transfer their mass and fused affine momentum and particle force to the Eulerian background grid (specific to MLS-MPM) using the affine particle-in-cell (APIC) scheme (see Equation 2.24).

(2) **Compute grid velocities:** The grid velocity field is obtained by dividing the momentum

by the mass with $\mathbf{v}_i = m_i \mathbf{v}_i / m_i$, where i is the voxel linear index in the grid.

(3) **Apply external forces:** External body forces $\mathbf{g}$ such as gravity are integrated into the grid velocities according to $\mathbf{v}_i^{n+1} = \mathbf{v}_i^n + \Delta t \mathbf{g}$, where n represents the timestep index.

(4) **Enforce boundary conditions:** Boundary conditions for the domain closed walls and the colliders are enforced as described in Section 2.2.7.

(5) **Transfer grid to particle (G2P):** Grid velocities are transferred back to the particles $\mathbf{v}_p^{n+1} = \sum_i w_{ip}^n \mathbf{v}_i^{n+1}$and the affine velocity matrix $\mathbf{C}_p$ is reconstructed following Equation 2.23.

(6) **Advect particles:** Particles are advected using their new velocity obtained through Symplectic Euler time integration $\mathbf{x}_p^{n+1} = \mathbf{x}_p^n + \Delta t \mathbf{v}_p^{n+1}$.

(7) **Update deformation gradient:** The deformation gradient is updated using the affine velocity matrix $\mathbf{C}_p$ (specific to MLS-MPM) according to Equation 2.26.

(8) **Apply plastic deformation:** When required, plastic deformation is applied by factoring the deformation gradient $\mathbf{F}$ in an elastic $\mathbf{F}_E$ and plastic $\mathbf{F}_P$ component (see Equation 2.29). This is only performed on materials that can get permanently displaced under sufficient forces.

(9) **Compute stress-based forces:** Compute the internal forces opposing the material deformation based on the elastic component of the deformation gradient $\mathbf{F}_E$ (see Equation 2.21).

2.2.2. Constitutive Models. A constitutive model is a mathematical model that describes the relationship between the stress and strain of a material. We start by introducing the First Piola-Kirchoff stress and the Cauchy stress, respectively defined as

$$\mathbf{P} = \frac{\partial \Psi}{\partial \mathbf{F}} \qquad \text{and} \qquad \boldsymbol{\sigma} = \frac{1}{J}\mathbf{P}\mathbf{F}^\top = \frac{1}{\det(\mathbf{F})}\frac{\partial \Psi}{\partial \mathbf{F}}\mathbf{F}^\top \tag{2.13}$$

where $J = \det(\mathbf{F})$. They are both fundamental quantities used to describe the constitutive behavior of materials. The potential energy density function Ψ fully specifies a constitutive model.

In the case of elasto-plastic materials, the fixed corotated constitutive model is commonly employed [JST+16] and admits the following potential energy density function

$$\Psi(\mathbf{F}) = \mu \sum_{i=1}^{d} (\sigma_i - 1)^2 + \frac{\lambda}{2}(J-1)^2 \tag{2.14}$$

where σ_i are the singular values on the diagonal of matrix $\mathbf{\Sigma} \in \mathbb{R}^{d\times d}$, which is part of the singular value decomposition $\mathbf{F} = \mathbf{U\Sigma V}^\top$. The Lamé parameters μ and λ relate the strain-stress relationship to the material properties E (Young's modulus) and ν (Poisson's ratio) such that

$$\mu = \frac{E}{2(1+\nu)}, \quad \lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}. \tag{2.15}$$

The First Piola-Kirchoff stress is obtained by differentiating Ψ with respect to the deformation gradient $\mathbf{F}$ according to

$$\mathbf{P}(\mathbf{F}) = \frac{\partial \Psi}{\partial \mathbf{F}}(\mathbf{F}) = 2\mu(\mathbf{F} - \mathbf{R}) + \lambda(J-1)J\mathbf{F}^{-\top} \tag{2.16}$$

where $\mathbf{R} = \mathbf{UV}^\top$ and $\mathbf{S} = \mathbf{V\Sigma V}^\top$ are respectively the rotational and scaling components of the polar decomposition $\mathbf{F} = \mathbf{RS}$.

Liquids are modeled using the constitutive model from Tampubolon et al. [TGK+17] defined as

$$\frac{\partial \Psi}{\partial \mathbf{F}}(\mathbf{F}) = k\left(1 - \frac{1}{J^\gamma}\right) \tag{2.17}$$

where k is the bulk modulus of water and γ is a term that more severely penalizes deviation from incompressibility.

2.2.3. Stress-based forces. Given that a material has a well-defined potential energy function Ψ, MPM internal forces can be derived from the total potential energy

$$\int_{\Omega^0} \Psi(\mathbf{F}_p) d\mathbf{X} \tag{2.18}$$

where Ω^0 corresponds to the undeformed state of the material, which admits the following spatial discretization on the Eulerian grid

$$\Phi = \sum_p V_p^0 \Psi_p(\mathbf{F}_p). \tag{2.19}$$

The elastic forces at each grid node are defined as the negative gradient of the total potential energy such that

$$\mathbf{f}_i(\mathbf{x}_i^n) = -\frac{\partial \Phi}{\partial \mathbf{x}_i}(\mathbf{x}_i^n) = -\sum_p V_p^0 \left(\frac{\partial \Psi_p}{\partial \mathbf{F}}(\mathbf{F}_p^n(\mathbf{x}_i^n)) \right) (\mathbf{F}_p^n)^\top \nabla w_{ip}^n = -\sum_p V_p^n \boldsymbol{\sigma}_p^n \nabla w_{ip}^n \tag{2.20}$$

if we recall that $V_p^n = J_p^n V_p^0$ and $\boldsymbol{\sigma}_p^n = \frac{1}{J_p} \frac{\partial \Psi}{\partial \mathbf{F}}(\mathbf{F}_p^n(\mathbf{x}_i^n))(\mathbf{F}_p^n)^\top$. However, in MLS-MPM, it is possible to avoid the computation of ∇w_{ip}^n via

$$\mathbf{f}_i^n = \mathbf{f}_i(\mathbf{x}_i^n) = -\sum_p V_p^n (\mathbf{D}_p^n)^{-1} \boldsymbol{\sigma}_p^n w_{ip}^n (\mathbf{x}_i^n - \mathbf{x}_p^n) \tag{2.21}$$

where $\mathbf{D}_p^n = \sum_i w_{ip}^n (\mathbf{x}_i^n - \mathbf{x}_p^n)(\mathbf{x}_i^n - \mathbf{x}_p^n)^\top$ but simplifies to the constant $\frac{1}{4}(\Delta x)^2$ when $N_i(\mathbf{x})$ is the quadratic interpolation function [JST+16].

2.2.4. APIC Transfers. The transfers between the particles and the Eulerian grid use an affine particle-in-cell (APIC) scheme [JSS+15]. In traditional MPM, particle momentum is transferred to the grid using

$$m_i^n \mathbf{v}_i^n = \sum_p w_{ip}^n m_p (\mathbf{v}_p^n + \mathbf{C}_p^n (\mathbf{x}_i^n - \mathbf{x}_p^n)) \tag{2.22}$$

where $\mathbf{C}_p^n = \mathbf{B}_p^n (\mathbf{D}_p^n)^{-1}$ and the affine velocity matrix is constructed from the grid to the particles with

$$\mathbf{B}_p^{n+1} = \sum_i w_{ip}^n \mathbf{v}_i^{n+1} (\mathbf{x}_i^n - \mathbf{x}_p^n)^\top. \tag{2.23}$$

In contrast, with MLS-MPM, we can fuse the scattering of the affine momentum and particle stress contribution, so the momentum transfer to the grid becomes

$$m_i^n \mathbf{v}_i^n = \sum_p w_{ip}^n m_p (\mathbf{v}_p^n + \mathbf{Q}_p^n (\mathbf{x}_i^n - \mathbf{x}_p^n)) \tag{2.24}$$

where $\mathbf{Q}_p^n = \Delta t V_p^n m_p^{-1} \mathbf{D}_p^{-1} \boldsymbol{\sigma}_p^n + \mathbf{C}_p^n$. This limits the computational effort to a single matrix-vector product per inner loop and does not require the computation of the weights' gradient ∇w_{ip}^n as previously established.

2.2.5. Deformation Gradient Update. The deformation gradient is updated with

$$\mathbf{F}_p^{n+1} = \left(\mathbf{I} + \Delta t \frac{\partial \mathbf{v}^{n+1}}{\partial \mathbf{x}} (\mathbf{x}_p^n) \right) \mathbf{F}_p^n \tag{2.25}$$

where, with traditional MPM, $\frac{\partial \mathbf{v}^{n+1}}{\partial \mathbf{x}} (\mathbf{x}_p^n) = \sum_i \mathbf{v}_i^{n+1} \nabla N_i (\mathbf{x}_p^n)^\top$. Interestingly, MLS-MPM achieves the same update by setting $\frac{\partial \mathbf{v}^{n+1}}{\partial \mathbf{x}} (\mathbf{x}_p^n) = \mathbf{C}_p^{n+1}$ using the existing affine velocity matrix simplifying the update to

$$\mathbf{F}_p^{n+1} = (\mathbf{I} + \Delta t \mathbf{C}_p^{n+1}) \mathbf{F}_p^n. \tag{2.26}$$

2.2.6. Plasticity. In order to model plastic deformation as observed with snow, soil, and other material types, we factor the deformation gradient into two terms $\mathbf{F} = \mathbf{F}_E \mathbf{F}_P$ where $\mathbf{F}_E$ and $\mathbf{F}_P$ respectively represent the elastic and plastic components of the deformation. For materials with plasticity, the elastic deformation gradient $\mathbf{F}_E$ will oppose deformation up to a certain point. Once the material surpasses this threshold, it stays deformed even after the forces that caused the deformation are withdrawn. The difference between the actual rest state and this new deformed equilibrium is captured by the plastic deformation gradient $\mathbf{F}_P$.

Let θ_c and θ_s define the two thresholds for compression and stretching such that the singular values of $\mathbf{F}_E$ must remain within the range $[1 - \theta_c, 1 + \theta_s]$. Any deformation outside this range will be classified as part of the plastic component of the deformation gradient $\mathbf{F}_P$. More formally, we have that

$$\begin{aligned} \mathbf{F}_E^{n+1} &= \mathbf{F}^{n+1} (\mathbf{F}_P^n)^{-1} \\ &= \mathbf{U}_E^{n+1} \boldsymbol{\Sigma}_E^{n+1} (\mathbf{V}_E^{n+1})^\top. \end{aligned} \tag{2.27}$$

Let $\tilde{\mathbf{F}}_E^{n+1} = \mathbf{U}_E^{n+1} \tilde{\boldsymbol{\Sigma}}_E^{n+1} (\mathbf{V}_E^{n+1})^\top$ such that $\tilde{\boldsymbol{\Sigma}}_E^{n+1}$ is a clamped version of $\boldsymbol{\Sigma}_E^{n+1}$ according to

$$\tilde{\sigma}_{Ei} = \text{clamp}(\sigma_{Ei}, 1 - \theta_c, 1 + \theta_s), \quad i = 1, \ldots, d. \tag{2.28}$$

In order to maintain a valid decomposition $\mathbf{F}^{n+1} = \mathbf{F}_E^{n+1}\mathbf{F}_P^{n+1}$, we set

$$\mathbf{F}_E^{n+1} = \tilde{\mathbf{F}}_E^{n+1} \quad \text{and} \quad \mathbf{F}_P^{n+1} = (\tilde{\mathbf{F}}_E^{n+1})^{-1}\mathbf{F}^{n+1}. \tag{2.29}$$

The hardening and softening of material under compression and stretching is modeled using $J_P = \det(\mathbf{F}_P)$ and ξ, the user-defined hardening parameter, where the original Lamé parameters defined in Equation 2.15 are modified according to

$$\mu(\mathbf{F}_P) = \mu_0 e^{\xi(1-J_P)}, \quad \lambda(\mathbf{F}_P) = \lambda_0 e^{\xi(1-J_P)} \tag{2.30}$$

before being used to compute the stress-based forces in Equation 2.16.

2.2.7. Collision and Friction. Collisions are trivially tested using a signed distance field (SDF) representation of the colliders with $\phi \leq 0$. Let $\mathbf{n} = \nabla\phi$ be the gradient of the SDF. The collision is computed in object space using the collider velocity $\mathbf{v}_{co}$ such that the relative velocity on the grid is given by $\mathbf{v}_{rel} = \mathbf{v} - \mathbf{v}_{co}$. If $v_n = \mathbf{v}_{rel} \cdot \mathbf{n} \geq 0$, no collision response is applied since the particle is already separating from the collider. Otherwise, we define the velocity locally tangent to the collider as $\mathbf{v}_t = \mathbf{v}_{rel} - v_n\mathbf{n}$.

Using μ as the friction coefficient, we apply static friction if $||\mathbf{v}_t|| \leq -\mu v_n$, in which case we set $\mathbf{v}'_{rel} = 0$, where $'$ designates that collision response was applied. If $||\mathbf{v}_t|| > -\mu v_n$, we apply dynamic friction with

$$\mathbf{v}'_{rel} = \mathbf{v}_t + \mu\mathbf{v}_n\frac{\mathbf{v}_t}{||\mathbf{v}_t||} \tag{2.31}$$

where $\mathbf{v}_n$ is the velocity locally normal to the collider. In both cases, the velocity is returned to world space according to $\mathbf{v}' = \mathbf{v}'_{rel} + \mathbf{v}_{co}$.

When dealing with animated colliders, there are typically two approaches to consider. The dynamic approach supports deforming geometries where the animation is composed as a sequence of distinct SDFs and their associated velocity field. In this case, it is possible to interpolate between two consecutive SDFs and velocities at time $t + \gamma\Delta t$ with $\gamma \in (0, 1)$ using the formulae

introduced by Selle et al. [SLF08].

$$\begin{aligned} \phi(\mathbf{x}, t+\gamma\Delta t) &= (1-\gamma)\phi(\mathbf{x}-\gamma\Delta t\mathbf{v}_{co}, t) + \gamma\phi(\mathbf{x}+(1-\gamma)\Delta t\mathbf{v}_{co}, t+\Delta t) \\ \mathbf{v}_{co} &= (1-\gamma)\mathbf{v}(\mathbf{x}, t) + \gamma\mathbf{v}(\mathbf{x}, t+\Delta t) \end{aligned} \tag{2.32}$$

Another approach is to use a static SDF and provide a sequence of rigid transformations. In this case, the transformations are interpolated to compute the position and orientation of the collider at any given time, and no velocity field is required as it can be computed from the transformations directly.

2.3 Deep Neural Networks

Deep feedforward networks, also known as multilayer perceptrons (MLPs), play a crucial role in the field of deep learning, serving as the foundational building blocks for some of the most advanced models developed to date. Essentially, an MLP $\mathbf{f}$ seeks to approximate a target function $\mathbf{f}^*$ by iteratively adjusting its learned parameters $\boldsymbol{\theta}$ during the training process, with the ultimate goal of achieving an output $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}; \boldsymbol{\theta})$ that closely approximates the ground-truth evaluation $\mathbf{y} = \mathbf{f}^*(\mathbf{x})$.

An MLP $\mathbf{f}$ is composed of multiple layers that transform input $\mathbf{x}$ into a higher-level representation. Each layer's output serves as input to the next layer, resulting in a series of nested functions $\mathbf{f}(\mathbf{x}) = \mathbf{f}^{(k)}(\mathbf{f}^{(k-1)}(\dots(\mathbf{f}^{(1)}(\mathbf{x}))\dots))$, where the depth of network k is the number of nested functions. In supervised learning, the network is trained on pairs of input examples and their corresponding ground-truth outputs $(\mathbf{x}, \mathbf{y})$ to minimize the difference between $\mathbf{y}$ and prediction $\hat{\mathbf{y}}$. The training examples only impose constraints on the input and output layers, allowing the network to construct its own latent representation of the input data through its internal layers, called hidden layers $\mathbf{h}^{(k)}$. This enables the network to capture complex and abstract patterns that are not directly observable in the input data.

The individual layers in an MLP consist of multiple units, also known as neurons, as they initially drew inspiration from neuroscience. In a fully-connected layer, each neuron receives the previous layer's output as input and computes a single scalar value, known as its activation. Even

though we use the term "neural", it is important to note that artificial neural networks bear little resemblance to their biological counterparts. Instead, they should be viewed as mathematical models or algorithms that allow function approximation with generalization.

Unlike linear models, MLPs are capable of learning nonlinear functions of their input. Each neuron establishes a mapping $y = \phi(\mathbf{x}, \boldsymbol{\theta})$, where y is a new data representation. If ϕ is learned and belongs to a broad class of nonlinear functions, we allow the algorithm to find the proper mapping through nonconvex optimization. Practitioners can apply constraints to the family of functions ϕ to incorporate domain knowledge into the model, also known as inductive bias. Various specialized models have emerged to solve domain-specific problems, such as the convolutional neural network (CNN) that will be discussed in Section 2.3.3.

Despite their power, not all aspects of an MLP are learned. Successful training of a neural network involves many design decisions, such as selecting the optimizer, the objective function l, and the output format, as it is the case with other machine-learning models. These architecture parameters that are kept static during training are called hyperparameters. For MLPs specifically, those include the network depth (number of layers), the width of each layer (number of neurons), and the choice of activation function for each layer. Training is carried out using the back-propagation algorithm (Section 2.3.2), which allows the gradients of the network to be efficiently and progressively computed in reverse order, starting from the objective function to the parameters of the network.

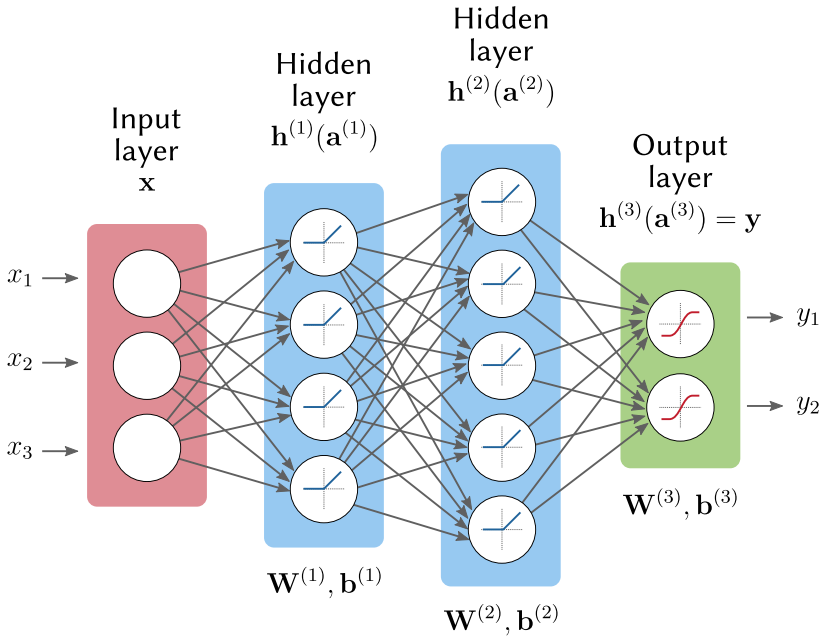


Figure 2.4. Instance of a small feedforward neural network's architecture, which takes three input values x_i and generates two outputs y_j. It consists of two hidden layers, the first with four neurons and the second with five. The weights and biases of each layer k are denoted by $\mathbf{W}^{(k)}$ and $\mathbf{b}^{(k)}$, respectively. The output of the linear transformation of a layer is referred to as the pre-activation $\mathbf{a}^{(k)}$, while the result after applying the activation function $\mathbf{h}^{(k)}$ is the layer's output or activation. Note that while both hidden layers use the same activation function (a ReLU in this example), the output layer typically uses a different one (a hyperbolic tangent in this example), as shown here.

2.3.1. Architecture. Let us consider a small MLP (Figure 2.4) to get familiar with the architecture and its formal notation. The set of trainable parameters is defined as

$$\boldsymbol{\theta} = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \mathbf{W}^{(3)}, \mathbf{b}^{(3)}\} \tag{2.33}$$

where $\mathbf{W}^{(k)}$ is the weight matrix, such that each row represents the weights of a single neuron, and $\mathbf{b}^{(k)}$ is a vector of biases containing a single scalar per neuron to offset the result of the affine transformation $\mathbf{a}^{(k)}$ applied to the previous layer output $\mathbf{h}^{(k-1)}(\mathbf{x})$ such that

$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}_j^{(k)} + \sum_j \mathbf{W}_{i,j}^{(k)} \mathbf{h}^{(k-1)}(\mathbf{x})_j = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}(\mathbf{x}). \tag{2.34}$$

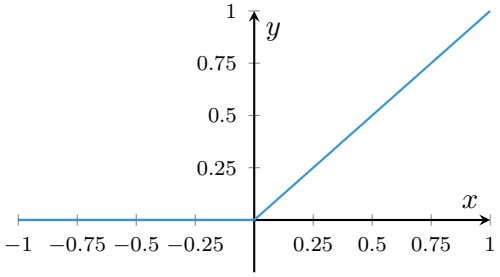
This result, also called the pre-activation of layer k, is then followed by an activation function g (usually nonlinear), which ultimately yields the output of the hidden layer

$$\mathbf{h}^{(k)}(\mathbf{x}) = g(\mathbf{a}^{(k)}(\mathbf{x})). \tag{2.35}$$

Figure 2.4 highlights the fact that the output layer's activation function often differs from the one used for the hidden units. The sigmoid and softmax functions are commonly used in binary and multi-class classification problems, whereas a linear function is preferred for regression tasks. The choice of activation function for the output layer is heavily influenced by the loss function used to optimize the model during training, as discussed in Section 2.3.1.

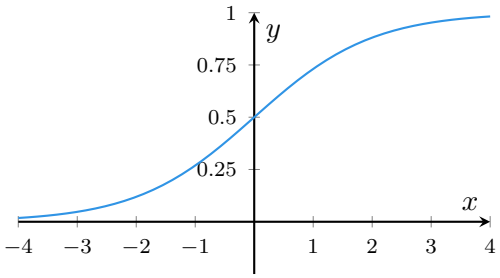
The concept of capacity is another important topic that refers to the level of complexity that can be modeled or captured by a given network. A linear model, for example, has limited capacity as it can only capture a linear relationship between inputs. In contrast, the universal approximation theorems [HSW89] show that an MLP with a single hidden layer can model a function of arbitrary complexity given an unlimited number of neurons. However, in practice, many hyperparameters can significantly affect the model's capacity. It is essential to carefully consider the specific task at hand to set it appropriately, as excessive capacity will lead to other challenges. On complex tasks, we strive for high capacity at low computational cost. While no rigorous theoretical results currently support these findings, empirical evidence shows that deeper networks tend to offer a better tradeoff than wider networks. Nevertheless, training these deep networks presents challenges, as discussed in Section 2.3.2.

Activation Functions. The purpose of the activation function is to introduce nonlinearity in the neurons' output, which is vital for neural networks to learn complex patterns and relationships from their input. Some standard activation functions include the rectified linear unit (ReLU) [NH10] function



$$ReLU(x) = \max(0, x), \tag{2.36}$$

which is often used in hidden layers and has the advantage of being computationally efficient, as well as the sigmoid function σ



$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.37}$$

that maps any input value to a value between 0 and 1, which is useful for binary classification problems.

2.3.2. Learning. The general framework to train a neural network can be expressed as

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{T} \sum_{t} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), \mathbf{y}^{(t)}) + \lambda \Omega(\boldsymbol{\theta}) \tag{2.38}$$

where we minimize the loss function l, also known as the empirical risk, by adjusting the parameters $\boldsymbol{\theta}$ of the model $\mathbf{f}$ on the T example pairs $(\mathbf{x}^{(t)}, \mathbf{y}^{(t)})$. The loss function can optionally be combined with a regularization term $\lambda \Omega(\boldsymbol{\theta})$ to impose constraints on the learned parameters.

The general algorithm to train a neural network first initializes the parameters. The bias $\mathbf{b}$ can be set to zero, but the weights need to be initialized based on a precise heuristic such as [GB10]

$$\mathbf{W}_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right) \tag{2.39}$$

where U is a uniform distribution and m, n are the sizes of the input and output of the layer

so that $\mathbf{W} \in \mathbb{R}^{n \times m}$. Then, gradient descent iterations can be applied where we compute the gradient of the loss function l with respect to the parameters for each example $(\mathbf{x}^{(t)}, \mathbf{y}^{(t)})$ of the whole training set of size m such that

$$\Delta_{\boldsymbol{\theta}} = \frac{1}{m} \sum_{t=1}^{m} \nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), \mathbf{y}^{(t)}) + \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta}). \tag{2.40}$$

Regrettably, the aforementioned procedure exhibits linear scaling with respect to the volume of data present in the training set. As the quantity of data increases, the efficiency of the algorithm diminishes accordingly. Consequently, the vast majority of deep learning algorithms employ a technique known as stochastic gradient descent (SGD) to mitigate this issue. SGD restricts the number of data instances considered for gradient calculation to a fixed count N, which is significantly smaller than the total dataset size m. This subset of data, commonly referred to as a minibatch, enables the approximation of the gradient $\Delta_{\boldsymbol{\theta}}$ by uniformly sampling from the entire training set such that

$$\tilde{\Delta}_{\boldsymbol{\theta}} = \frac{1}{N} \sum_{t=1}^{N} \nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), \mathbf{y}^{(t)}) + \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta}). \tag{2.41}$$

Adopting this approach offers a twofold advantage. Firstly, it improves computational efficiency by reducing the burden associated with processing the entire training set. Secondly, the introduction of noise resulting from the estimation of the true gradient helps to prevent the algorithm from becoming trapped in local minima. The approximated gradient is then used to perform an optimization step downhill with

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \tilde{\Delta}_{\boldsymbol{\theta}} \tag{2.42}$$

where $\eta \in (0, 1]$ is usually a very small scalar corresponding to the learning rate. In order to apply this algorithm, we must have a well-defined loss function l and a procedure to compute its gradient $\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), \mathbf{y}^{(t)})$ with respect to the parameters of the network.

Loss Functions. During training, the network makes predictions from the inputs in the training set. The loss measures how well the network performs by comparing its output to the cor-

responding ground-truth values. This result is later used to update the network's parameters to improve its accuracy.

The choice of loss functions is derived from the problem being solved. For instance, the mean squared error (MSE)

$$l(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{2.43}$$

is a common loss function used in regression problems, where N is the number of training samples in a minibatch. As another example, the cross-entropy loss

$$l(\hat{\mathbf{y}}, \mathbf{y}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} y_{i,j} \log \hat{y}_{i,j} \tag{2.44}$$

is commonly used in classification problems with $y_{i,j} = 1$ if example i belongs to class j and $y_{i,j} = 0$ otherwise.

REGULARIZATION. Overfitting typically occurs when a model fits the training data too closely and fails to generalize to new, unseen data. This can happen when a model has too much capacity compared to the size of the training set. The regularization penalty term $\lambda\Omega(\boldsymbol{\theta})$ added to the loss function during training encourages the model to learn simpler, more generalizable representations. Some of the most popular techniques include L1/L2 regularization, dropout, and early stopping.

L1 regularization adds a penalty term proportional to the absolute value of the model parameters, which encourages the model to learn sparse representations where many parameters are close to zero. L2 regularization adds a penalty term proportional to the square of the model parameters, forcing the model to learn small parameter values, resulting in smoother decision boundaries and regressions.

Dropout randomly ignores a fraction of the activations during training. This constructs an alternative/smaller network for each training batch, preventing the network from relying on a small subset of neurons or specific paths to perform well.

Early stopping allows high-capacity models to be adequately trained on relatively small datasets. This is achieved by monitoring the performance on a validation set separate from the

data used for training, so the optimization procedure can be stopped as soon as the validation loss/accuracy stops improving.

Back-Propagation. The algorithm that makes neural networks trainable in a reasonable amount of time is called back-propagation. The procedure is based on the well-known chain rule from calculus where given a function

$$f(x) = p(q_1(x) + q_2(x)) \tag{2.45}$$

we can compute its derivative with

$$\frac{df(x)}{\Delta x} = \frac{df(x)}{dq_1(x)}\frac{dq_1(x)}{\Delta x} + \frac{df(x)}{dq_2(x)}\frac{dq_2(x)}{\Delta x}. \tag{2.46}$$

In multivariate calculus, if $\mathbf{x}$ is a vector in $\mathbb{R}^m$, $\mathbf{q}$ maps from $\mathbb{R}^m$ to $\mathbb{R}^n$ and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. This generalizes to the form

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}_i} = \sum_j \frac{\partial f(\mathbf{x})}{\partial \mathbf{q}_j(\mathbf{x})}\frac{\partial \mathbf{q}_j(\mathbf{x})}{\partial \mathbf{x}_i}, \tag{2.47}$$

which can be further rewritten in vector form as

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left(\frac{\partial \mathbf{q}(\mathbf{x})}{\partial \mathbf{x}}\right)^\top \nabla_{\mathbf{q}} f(\mathbf{x}) \tag{2.48}$$

where $\frac{\partial \mathbf{q}(\mathbf{x})}{\partial \mathbf{x}}$ is the $n \times m$ Jacobian matrix of $\mathbf{q}$ [GBC16].

To train the network, we must update the parameters $\boldsymbol{\theta}$ of our model $\mathbf{f}$. For instance, if we want to compute the gradient of the loss with respect to $\mathbf{W}^{(3)}_{i,j}$ (as in Figure 2.4), we need to compute

$$\frac{\partial}{\partial \mathbf{W}^{(k)}_{i,j}} l(f(\mathbf{x}), y) = \frac{\partial l(\mathbf{f}(\mathbf{x}), \mathbf{y})}{\partial \mathbf{a}^{(k)}(\mathbf{x})_i}\frac{\partial \mathbf{a}^{(k)}(\mathbf{x})_i}{\partial \mathbf{W}^{(k)}_{i,j}} \tag{2.49}$$

where $\frac{\partial \mathbf{a}^{(k)}(\mathbf{x})_i}{\partial \mathbf{W}^{(k)}_{i,j}}$ is relatively straightforward to compute. The other term $\frac{\partial l(\mathbf{f}(\mathbf{x}),\mathbf{y})}{\partial \mathbf{a}^{(k)}(\mathbf{x})_i}$ indirectly depends on $\partial \mathbf{a}^{(k)}(\mathbf{x})_i$, meaning that it also needs to be unpacked through another iteration of the

chain rule, resulting in

$$\frac{\partial l(\mathbf{f}(\mathbf{x}), \mathbf{y})}{\partial \mathbf{a}^{(k)}(\mathbf{x})_i} = \frac{\partial l(\mathbf{f}(\mathbf{x}), \mathbf{y})}{\partial \mathbf{h}^{(k)}(\mathbf{x})_i} \frac{\partial \mathbf{h}^{(k)}(\mathbf{x})_i}{\partial \mathbf{a}^{(k)}(\mathbf{x})_i} \tag{2.50}$$

where $\frac{\partial \mathbf{h}^{(k)}(\mathbf{x})_i}{\partial \mathbf{a}^{(k)}(\mathbf{x})_i}$ is simply evaluated as the derivative of the activation function g' at layer k and $\frac{\partial l(\mathbf{f}(\mathbf{x}), \mathbf{y})}{\partial \mathbf{h}^{(k)}(\mathbf{x})_i}$ can, in this case, also be evaluated directly since $k = 3$ corresponds to the output layer of the network. Note that computing the gradients of the parameters in that order will require many repeated computations as each $\mathbf{W}_{i,j}$ with the same i shares the same $\frac{\partial l(\mathbf{f}(\mathbf{x}), \mathbf{y})}{\partial \mathbf{a}^{(k)}(\mathbf{x})_i}$. The algorithm must, therefore, proceed in the reverse order, hence the "back" in back-propagation.

The main idea is to compute the gradients backward, starting from the loss function all the way to the parameters of the network. As a reference, we provide the equations of the algorithm for a network of L layers without derivations [Lar14]. The first step is to compute the gradient of the loss function with respect to the activation of the last layer

$$\nabla_{\mathbf{h}^{(L)}} l(\mathbf{f}(\mathbf{x}), \mathbf{y}). \tag{2.51}$$

Then for the output layer $k = L$, we compute the gradient of the loss with respect to the pre-activation

$$\nabla_{\mathbf{a}^{(k)}} l(\mathbf{f}(\mathbf{x}), \mathbf{y}) = \nabla_{\mathbf{h}^{(k)}} l(\mathbf{f}(\mathbf{x}), \mathbf{y}) \odot [\dots, g'(\mathbf{a}^{(k)}(\mathbf{x})_j), \dots] \tag{2.52}$$

where $\odot$ is the Hadamard product (element-wise) and $[\dots, g'(\mathbf{a}^{(k)}(\mathbf{x})_j), \dots]$ is a vector containing the derivatives of the activation function g'. Subsequently, we can compute the gradients of the hidden layer parameters

$$\begin{aligned} \nabla_{\mathbf{W}^{(k)}} l(\mathbf{f}(\mathbf{x}), \mathbf{y}) &= \nabla_{\mathbf{a}^{(k)}} l(\mathbf{f}(\mathbf{x}), \mathbf{y}) \mathbf{h}^{(k-1)}(\mathbf{x})^\top \\ \nabla_{\mathbf{b}^{(k)}} l(\mathbf{f}(\mathbf{x}), \mathbf{y}) &= \nabla_{\mathbf{a}^{(k)}} l(\mathbf{f}(\mathbf{x}), \mathbf{y}) \end{aligned} \tag{2.53}$$

and finally, compute the gradient of the previously hidden layer to finish an iteration

$$\nabla_{\mathbf{h}^{(k-1)}} l(\mathbf{f}(\mathbf{x}), \mathbf{y}) = (\mathbf{W}^{(k-1)})^\top (\nabla_{\mathbf{a}^{(k)}} l(\mathbf{f}(\mathbf{x}), \mathbf{y})). \tag{2.54}$$

These steps are repeated for $k = L - 1$ to 1, where the last equation is ignored when $k = 1$ such

that $\mathbf{h}^{(k-1)} = \mathbf{x}$.

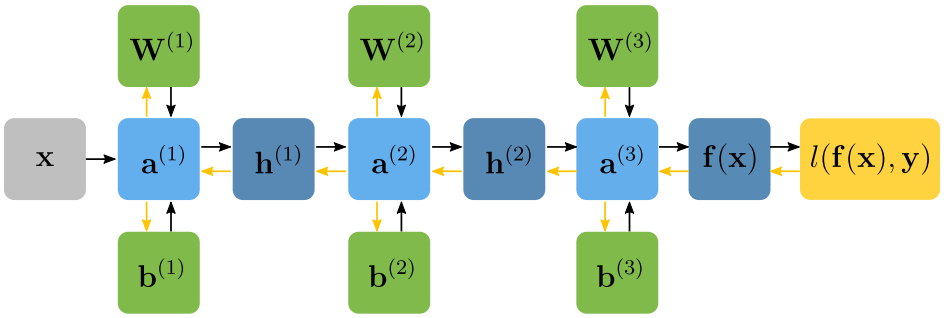


Figure 2.5. The flow graph of both the forward and backward propagation passes in a neural network of three layers. The forward path, depicted in black, begins from the input $\mathbf{x}$ and propagates to the loss function l. Conversely, the backward path or gradient flow, depicted in yellow, starts from the loss function and propagates to the parameters $\mathbf{W}^{(k)}$ and $\mathbf{b}^{(k)}$ of each layer k.

A neural network's forward and backward passes can be visualized in Figure 2.5 through an acyclic computational graph, also known as a flow graph. In the forward pass, the input is propagated through the network to compute the output as well as the intermediate results for each node of the computational graph. The backward pass involves reversing the graph and computing the gradients using the results stored during the forward pass. The optimization step (Equation 2.42) is performed when parameters are encountered.

2.3.3. Convolutional Networks. Convolutional Neural Networks (CNNs) [LCJB+89] are a specialized type of feedforward network designed to process data that are organized in a grid-like structure, such as audio signals (1D), images (2D), and volumetric data (3D). The architecture gets its name from the convolution operation, which is a key component of the network. While the mathematical definition of convolution differs slightly from the operation performed in CNNs (technically called cross-correlation), we assume they are the same for simplicity. CNNs have demonstrated remarkable success in many applications, particularly in computer vision tasks such as object detection and segmentation. This section introduces the convolution operation and pooling mechanism, both essential aspects of CNNs. Figure 2.6 provides a typical CNN architecture for reference.

One of the fundamental concepts behind CNNs is called parameter sharing. The convolution

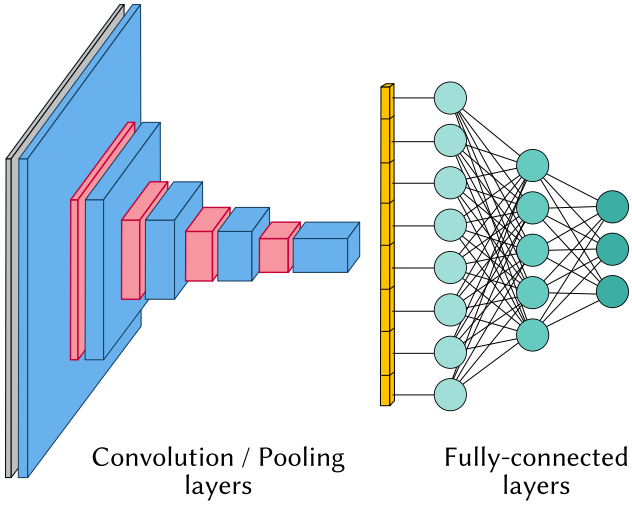


Figure 2.6. Architecture of a CNN. The input (colored gray) goes through a series of convolution layers (colored blue) and pooling layers (colored red) before being flattened in a 1D tensor (colored yellow). This is then passed to three fully-connected layers (colored turquoise) before reaching the network's end. The pooling layers are scaling down the feature maps (tensor width and height) while the convolution layers are extending the feature space (tensor depth) to build a meaningful latent representation of the input that the fully-connected layers will ultimately have to reason about.

operation is performed by sliding a small kernel over the input array, producing a new feature map. Parameter sharing is achieved by using the same kernel for all locations of the input array, which means that the same set of parameters is used to compute each output element. This approach is efficient because it dramatically reduces the number of parameters in the model, making it easier to train and less prone to overfitting. Furthermore, this enables CNNs to learn local features that are translation invariant.

CONVOLUTION. Let us assume we have a continuous noisy signal x that we want to smooth out to obtain its average behavior s at any given time t. To achieve this, we can use a weight function w, defined as $w = 1/2\epsilon$ if $t - \epsilon < t < t + \epsilon$ and $w = 0$ everywhere else, known as a box filter. Here, ϵ is a real number that ensures w is a proper density function that integrates to 1 such that

$$s(t) = \int_{-\infty}^{\infty} x(\tau) w(t - \tau) d\tau \tag{2.55}$$

will give a weighted average of the signal x around time t. This corresponds to the definition of the convolution operator in continuous space, also denoted $s(t) = (x * w)(t)$ where $*$ is the convolution operator. In practice, we must discretize time to obtain this new form

$$s(t) = \sum_n x(n)w(t-n). \tag{2.56}$$

This operator can be extended to an arbitrary number of dimensions. A discrete 2D convolution of the kernel K applied to image I takes the form

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n). \tag{2.57}$$

Note here that we use the commutative property of the convolution to offset I with i, j instead of K. We have also changed the subtraction to an addition with the m, n indices to omit the kernel flipping that is part of the convolutions' definition from mathematics. Technically, we perform cross-correlation here, but we will still refer to this operation as a convolution, as previously stated. In the context of CNNs, the convolution can be parameterized to achieve different results. It is possible to pad the input's border to perform what is known as a "same" convolution, maintaining the resolution throughout the layer (Figure 2.7 (a)). The kernel size can also be varied to consider more or less of the input from a single convolution, hence impacting the receptive field of the layer.

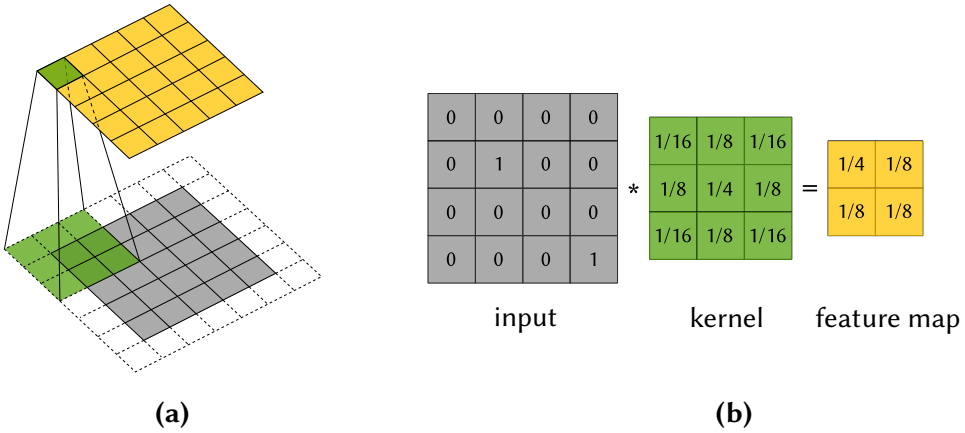


Figure 2.7. (a) A 3×3 kernel (colored green) is convolved with the data below (colored gray), and the resulting feature map is written above (colored yellow). Note that the input has a padding of zeros, which prevents the input's resolution from shrinking in the output, called "same" convolution. (b) In contrast, when convolution is performed without padding, the input's resolution is reduced by $K_{width} - 1$ in each dimension of the feature map, called "valid" convolution.

POOLING. Adding successive convolutional layers can increase neurons' receptive fields. However, this often requires large kernels or deep networks, which can be computationally inefficient. Pooling is a computationally efficient alternative that can increase the receptive field by partitioning the input data into small regions that are aggregated together.

Pooling is commonly applied to the convolution layer's activation in CNNs. It replaces each unit's output with a summary statistic of its local neighborhood, resulting in translation invariance. For example, a pooling operation with a 2×2 window is applied to a 4×4 input, resulting in a 2×2 output where each window performs pooling on its assigned values without overlapping the neighboring windows (Figure 2.8). By reducing the spatial resolution of the feature maps, the network becomes more computationally efficient. This spatial reduction is usually followed by an increase in the number of feature maps to allow the network to build a more expressive latent representation. Depending on the pooling type, it can also act as a feature selection mechanism where important characteristics are retained, but fine details are discarded. The two most common pooling types are max and average pooling, as shown in Figure 2.8.

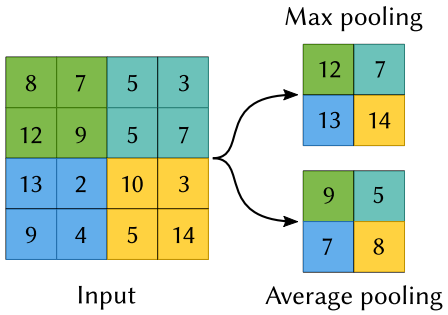


Figure 2.8. Max and average pooling operations applied to a 4×4 input. Each 2×2 pooling window computes its four assigned values' maximum or average. The windows do not overlap as they are shifted with a stride of two.

A similar effect can be achieved with a learned convolutional layer using a stride larger than one. For instance, applying a same convolution with a stride of two divides by two the spatial resolution of the feature map.

2.4 General-Purpose Computing on GPUs

The advent of graphics processing units (GPUs) for executing tasks traditionally reserved to central processing units (CPUs) has profoundly impacted algorithm design. These massively parallel architectures can handle vast amounts of data concurrently, making them highly effective for a multitude of tasks, including image processing, physics simulations, and machine learning. High-level specialized programming languages such as *CUDA* (Compute Unified Device Architecture) and *OpenCL* (Open Computing Language) can be turned into machine code to take advantage of GPU capabilities. This section provides an overview of the fundamental principles underlying the *CUDA/C++* programming language, inspired by the book by Sanders and Kandrot [SK10].

2.4.1. Data Transfers. The GPU, also known as the device, possesses its own memory. To perform computations, data must first be transferred from the CPU (or host) to the device. These memory transfers can be expensive and should be kept to a minimum. For instance, an algorithm composed of multiple phases, some easily parallelizable and others more suitable for sequential processing, should typically avoid using both the host and device for computation. Depending on

the data volume, frequent copies between host and device can rapidly neutralize the performance benefits gained by migrating the algorithm to the GPU.

It is essential to pay close attention to how data is laid out in device memory since it can drastically impact performance. To optimize memory access patterns, we usually strive for something called coalesced memory access, which happens when a group of threads accesses consecutive memory locations. This allows all threads to retrieve their respective data within a single memory transaction, leading to improved memory bandwidth utilization.

2.4.2. Warp Divergence. The device is capable of launching thousands of threads concurrently, with each thread executing the same kernel function on different data. Threads are organized into blocks, which are then organized into a grid. The SIMT (Single Instruction Multiple Threads) execution model governs how threads perform computation. Each warp, counting 32 threads, must execute instructions in lockstep such that an instruction executed by a partial warp will force the rest of the warp to wait idle for the instruction to finish. This is called thread divergence, which can potentially cause a $32\times$ slowdown in the worst case if each warp thread performs a different instruction. Note that it is possible to implement the same algorithm with varying degrees of divergence, which is a determinant factor of good kernel design.

2.4.3. Thread Cooperation. The size of a block is restricted to a specific amount of threads depending on the specific GPU architecture. It allows threads within the same block to collaborate on a common objective, oblivious of threads in other blocks.

SHARED MEMORY. Shared memory acts as a software-managed cache or *scratch pad*, representing a memory space unique to each block, which allows all threads to cooperate. While global device memory also permits cooperation, using shared memory allows much faster read and write access. For example, one can perform a reduction by first loading all data from global memory to shared memory, so each block can perform a subtask of the reduction in shared memory, before copying the result back to global memory. Many algorithms can significantly benefit from this kind of manual caching. However, to avoid potential race conditions and nondeterministic results, we must synchronize the block threads at critical points in the algorithm. For the reduction

of an array of size N, this synchronization must occur after each of the $\log N$ "folding" iterations ($N/2$, $N/4$, ..., 1) to prevent a read-before-write event from occurring.

2.4.4. Read-Only Memory. There are two types of read-only memory that can accelerate algorithms bound by memory access: constant memory and texture memory.

Constant memory is equipped with on-chip caching; it enables broadcast reads across multiple threads of a half-warp, resulting in improved memory throughput when multiple threads access the same data.

Texture memory, like constant memory, has on-chip caching; it performs remarkably well when data is accessed based on spatial locality. In contrast to typical multi-dimensional arrays stored in memory such that read-access performance varies per axis, texture memory offers uniform read access speed across all dimensions. Depending on the application, texture memory can provide significant performance gains over global memory use. Additionally, texture memory supports hardware-accelerated linear sampling for 1D, 2D, and 3D textures, further accelerating image and volume processing applications.

2.4.5. Graphic Interoperability. *CUDA* can read and modify buffers owned by graphic APIs such as *OpenGL* and *DirectX*. The main advantage is to avoid a roundtrip to host memory. This is especially crucial for physical systems where millions of particles are simulated before being rendered interactively at every frame. Having to copy the data to the host each time would make the solver almost as slow as its CPU counterpart.

2.4.6. Atomics. Certain algorithms require multiple threads to simultaneously modify the same memory address within a data structure. This sounds hazardous but can be accomplished safely by using atomic operations that are guaranteed to be performed in a single read-modify-write operation. This means that no other thread is allowed to read or write to this memory address while another atomic operation is conducted. The concept of atomic operations is similar to mutexes used in multi-threaded programming. It should be noted that atomic operations can quickly become a bottleneck in kernel execution, as threads must wait idle to access the same data. For example, performing a reduction by having thousands of threads incrementing the same memory

address would be wildly inefficient. Nevertheless, using atomics to combine a reasonable number of independently computed partial sums would have an unnoticeable impact on performance. Therefore, it is essential to consider the number of threads that will compete for access, and, if necessary, distribute the load across multiple independent atomic operations instead of a single one.

Chapter 3

Predicting Active Particles

The accurate simulation of physical phenomena can entail significant computational costs. Even seemingly simple scenarios, such as a static snow pile, may demand considerable resources just to stay stable and motionless.

At each timestep of the simulation, external forces such as gravity act upon the material, while internal forces such as volume conservation and resistance to deformation oppose them. From a numerical perspective, this delicate balance can rapidly become unstable. Such systems can experience unwanted oscillations, forcing explicit integrators to take tiny timesteps to preserve equilibrium. This ensures that the opposing forces propagate through the material rapidly enough to maintain stability and avoid movement.

Regrettably, the significant effort invested in preserving the immobility of a snow pile often goes unrecognized by viewers, as the scene appears lifeless and static. Nonetheless, this demanding computational task is necessary as it is impossible to predict when the material will undergo motion and suddenly become visually compelling. It goes without saying that this idle state is quite inefficient and resource-intensive.

Thus, we introduce a novel approach that categorizes particles as active ($\gamma = 1$) or passive ($\gamma = 0$) based on whether they should move. To achieve this, we use a convolutional neural network (CNN) trained on synthetic simulation data produced by our GPU-accelerated material point method (MPM) solver. Our approach enables the solver to allocate computational resources to the regions where motion is expected in the next timestep while leaving the passive particles

untouched. Notably, our methodology is not limited to MPM and can be implemented to accelerate other particle-based and hybrid simulation frameworks, such as Position-Based Dynamics (PBD) [MHHR07], Smoothed Particle Hydrodynamics (SPH) [Mon92], and Fluid Implicit Particle (FLIP) [BKR88], among others.

Developing hand-crafted rules that can accurately and efficiently detect when particles should be set as active or passive would be impractical due to the complex and varied behaviors that particle systems can exhibit. Merely establishing a suitable threshold based on particle speed may initially appear viable. However, we also need to account for the spatial extent of the stencil surrounding each particle and its corresponding falloff function. This consideration ensures that a barely moving particle, located far from another, does not inadvertently trigger its activation. Conversely, we can argue that a cluster of slowly moving particles collectively possesses sufficient influence to activate a distant particle, while the precise definition of a cluster remains ambiguous.

Hence, it becomes evident that the relationship between a particle's neighborhood and its activation can be multifaceted and complex. Implementing simple heuristics that conservatively activate particles could induce ripple effects due to sagging, leading to the unnecessary activation of all particles within the domain. Moreover, attempting to capture all possible scenarios through a comprehensive set of rules is often insurmountable.

Instead, machine learning (ML) offers a more viable solution to overcome these issues. By training a model on a sufficiently large dataset of MPM simulations featuring ground-truth information, the model may learn to discern the underlying patterns and behaviors that dictate particle activation without manual heuristics. Additionally, trained models can adapt to new scenarios and generalize to different conditions, reducing the effort and time required in the long run.

3.1 Generating the Ground Truth

Obtaining accurate and trustworthy data is a crucial initial phase for any ML endeavor, and our project is no exception. We aim to predict which particles will surpass a given speed threshold δ considering the current and the subsequent timestep. The following section explains how we

collected and curated our data to achieve this goal.

3.1.1. A Peek into the Future. It was determined through empirical experimentation that an appropriate threshold for particle speed, separating active ($\gamma_p = 1$) from passive ($\gamma_p = 0$) particles, is given by the expression:

$$\delta = \frac{1}{400}\Omega \tag{3.1}$$

where $\Omega = \sqrt[3]{V_{\mathcal{G}}}$ is what we call the domain scale, with its underlying grid $\mathcal{G}$ and its volume $V_{\mathcal{G}}$. Intuitively this means that for a domain of size $\mathcal{D} = (1, 1, 1)$ rendered at a resolution of 1920×1080 and a frame rate of 24 fps, with the camera closely framing the domain's width, a particle would move less than $\frac{1}{5}$ of a pixel per frame to be labeled passive. To improve generalization, we make this threshold dependent on the scene's scale Ω, making large scenes more tolerant of small motions and vice versa. This heuristic increases the portability of our system and reduces manual intervention compared to an absolute threshold.

The value of δ may initially seem arbitrary, but it only serves as the default behavior of our system. As discussed in Section 3.6.3, users can adjust the tolerance to particle speed on a per-scene basis without needing to retrain the model.

An active particle must meet the following criterion at time t:

$$\gamma_p^t = 1 \iff (||\mathbf{v}_p^t|| > \delta) \vee (||\mathbf{v}_p^{t+1}|| > \delta). \tag{3.2}$$

On the one hand, determining if $||\mathbf{v}_p^t|| > \delta$ is straightforward as the data is already available. However, determining if a particle will exceed our speed threshold in the next timestep, $||\mathbf{v}_p^{t+1}|| > \delta$, is more challenging.

One evident and perhaps disappointing approach is to simulate the particle to the next timestep to observe if it indeed exceeds δ. The generation of our dataset will therefore depend on a little temporal dance involving storing the initial state of the particle system at time t, advancing to time $t+1$ to identify the particles that should be designated as active, then returning to the initial time t now with the essential knowledge of what should be simulated. From there, we retrieve the particle's state from time t, save it to disk along with its activation information, and move

forward to time $t+1$, simulating only the particles set as active. An overview of this process is presented in Algorithm 1.

The method described above guarantees that the data produced closely approximates the distribution that our trained model will encounter at inference time. Labeling some particles as passive inevitably affects the behavior of their neighbors. Thus, by actively using the previous output as input to the next timestep, we incorporate perturbations that approximate the data faced at runtime. This technique employs a rationale similar to the noise annealing technique proposed by Sanchez-Gonzalez et al. [SGGP+20].

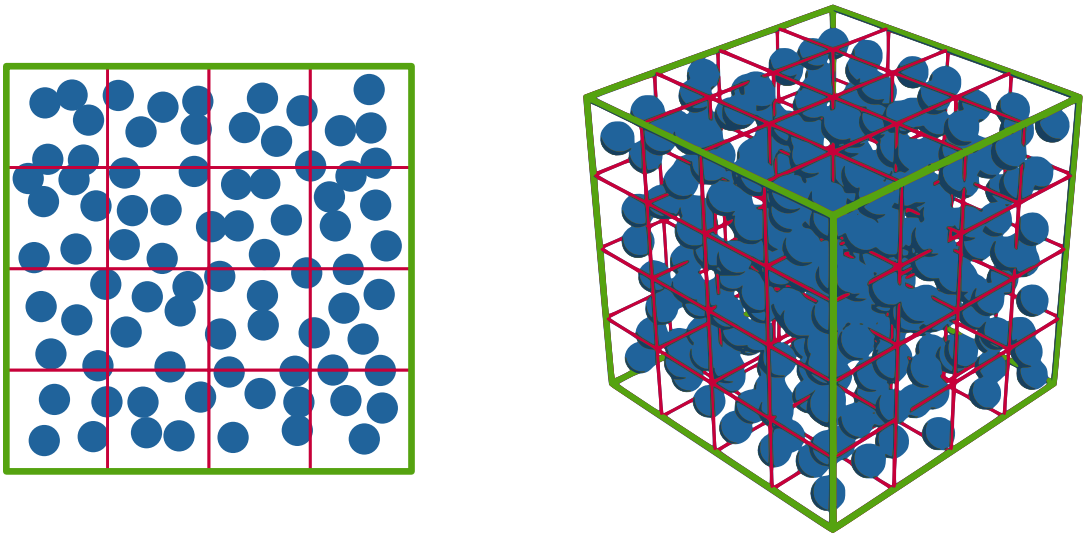
Algorithm 1 GROUNDTRUTHGENERATION()

for $sim = 1$ **to** 50 **do**
  $\mathcal{S} \leftarrow$ INITIALIZERANDOMSCENE( ) ▷ Initialize sources, colliders, and materials per sim
  for $frame = 1$ **to** 400 **do**
    for $substep = 1$ **to** 160 **do**
      $\tilde{\mathcal{P}} \leftarrow \mathcal{P}$ ▷ Store the particles' state at time t
      $\gamma_{\mathcal{P}} \leftarrow 1$ ▷ Make all particles active; $\gamma_p = 1$
      $\mathcal{P} \leftarrow$ STEPSIMULATION$(\mathcal{P}, \mathcal{S})$ ▷ solve to $t+1$
      $\gamma_{\mathcal{P}} \leftarrow \{\mathbf{v}_{\tilde{p}} > \delta \vee \mathbf{v}_p > \delta \,|\, \tilde{p} \in \tilde{\mathcal{P}} \text{ and } p \in \mathcal{P}\}$ ▷ Find ground truth γ using δ
      $\gamma_{\mathcal{T}} \leftarrow \gamma_{\mathcal{P}}$ ▷ Promote γ from particles to tiles
      $\mathcal{P} \leftarrow \tilde{\mathcal{P}}$ ▷ Restore the particles' state from time t
      if $substep = 1$ **then**
        WRITESAMPLETODISK$(\mathcal{P}, \gamma_{\mathcal{T}})$ ▷ Write the sample to disk once per $frame$
      $\gamma_{\mathcal{P}} \leftarrow \gamma_{\mathcal{T}}$ ▷ Promote γ from tiles to particles
      $\mathcal{P} \leftarrow$ STEPSIMULATION$(\mathcal{P}, \mathcal{S})$ ▷ solve to $t+1$, with proper activation

3.1.2. Tiled Spatial Partitioning. Instead of categorizing individual particles as active or passive, we propose to partition the simulation domain into tiles $\tau \in \mathcal{T}$ of 4^3 Eulerian grid cells (Figure 3.1). For instance, a simulation domain consisting of 128^3 Eulerian grid cells would comprise 32^3 tiles, effectively reducing the dimensionality by a factor of 64. This grouping provides several benefits, which we will elaborate on in Section 3.2 when discussing the neural network architecture. For now, it suffices to note that the activation attribute γ is assigned to each tile τ denoted γ_τ, and every particle is associated to a single tile index τ_p such that $\gamma_p = \gamma_{\tau_p}$ is the particle activation. A tile τ is considered active if it contains at least one active particle p, as

expressed formally by the following condition:

$$\gamma_\tau = 1 \iff \exists p \in \mathcal{P} \,|\, \tau_p = \tau \wedge \gamma_p = 1. \tag{3.3}$$

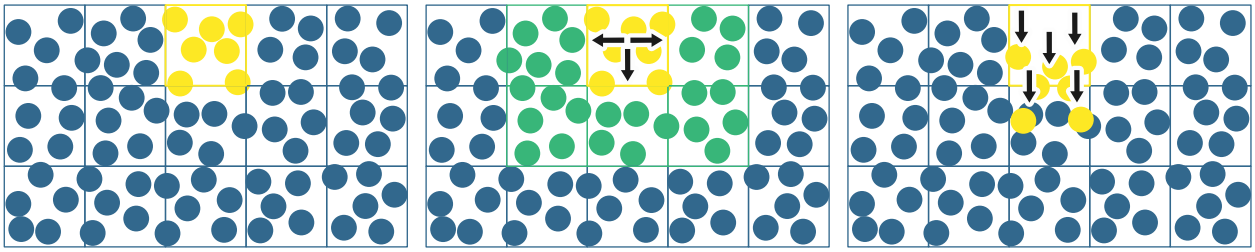


Tile in 2D: 4×4 Eulerian grid cells

Tile in 3D: $4 \times 4 \times 4$ Eulerian grid cells

Figure 3.1. The simulation domain is subdivided into tiles (colored green) consisting of 4^3 Eulerian grid cells (colored red). Each tile maintains the active attribute, and every particle (colored blue) is associated with exactly one tile.

3.1.3. Naive Passive Tiles. Eliminating the passive tiles from computation can significantly enhance performance of several scenes. However, this naive approach creates another important issue. For instance, imagine a vertical stack of two tiles where the bottom tile is passive, and the top tile is active. If we remove the bottom tile entirely from the simulation, the top tile will move downward due to gravity, lacking any resistance from the absent bottom tile (Figure 3.2). On the other hand, if we retain all passive tiles in the simulation, we will almost return to the original computational complexity. Although we can gain a little by ignoring the passive particles in the GridToParticle( ) kernel where the integration occurs, the ParticleToGrid( ) and GridVelocityUpdate( ) kernels remain just as slow as they used to be.



(a) Tile activation time t **(b)** With boundary tiles time $t + \Delta t$ **(c)** No boundary tiles time $t + \Delta t$

Figure 3.2. (a) Tiles are assigned an activation of active (colored yellow) or passive (colored blue) for the next timestep. (b) Passive tiles directly adjacent to active tiles are labeled as boundary (colored green); they are partially processed by the solver to provide proper boundary conditions to active tiles. (c) The naive approach, without boundary tiles, where passive tiles are ignored by the solver, sends active tiles in free fall under gravity.

To ensure that active tiles receive the appropriate boundary conditions, we must introduce a new type of tile: the boundary tile ($\gamma = 2$). All active tiles will have their adjacent neighbor tiles (8 in 2D, 26 in 3D) running, regardless of their activation attribute (Figure 3.2). Thus, boundary tiles are passive tiles scattered to the Eulerian grid, like active tiles, but not integrated forward in time. Fortunately, identifying these boundary tiles can be accomplished in parallel on the GPU using a gather operation on each tile as a post-process after activation.

One crucial aspect that necessitates careful consideration is the physical attributes of the boundary tiles. In the ParticleToGrid( ) kernel, the boundary tiles are scattered to the Eulerian grid. It is, therefore, vital to nullify their velocity $\mathbf{v}$ and affine velocity matrix $\mathbf{C}$ attributes to prevent them from imprinting non-zero velocity information onto the Eulerian grid. Neglecting this procedure can result in undesirable outcomes such as volume loss, drift, and other unpleasant artifacts that we aim to circumvent (Figure 3.3).

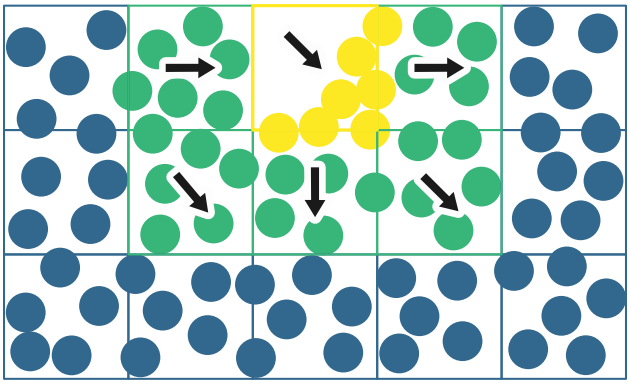


Figure 3.3. The boundary tiles (colored green) have been set passive without neutralizing their particles' velocity and affine velocity matrix. This induces subtle drift, and compression artifacts to the active tiles (colored yellow) as the solver will consider the boundary tiles in motion. The tiny artifacts encountered in practice were significantly exaggerated for illustration purposes only. The black arrows depict the average velocity direction of each tile.

3.1.4. Synthetic Dataset. The training dataset comprises 50 simulations randomly generated, each consisting of 400 frames. Each frame is a $(\mathbf{X}, \mathbf{Y})$ pair, where $\mathbf{X} \in \mathbb{R}^{128\times128\times128\times2}$ and $\mathbf{Y} \in \mathbb{R}^{32\times32\times32}$ have resolutions of $(128, 128, 128, 2)$ and $(32, 32, 32)$, respectively. The two floating-point channels in $\mathbf{X}$ represent the mass m and speed $||\mathbf{v}||$ of the particles scattered to the Eulerian grid. The mass is not directly utilized for training; rather, it is employed to derive a binary mask, which is later used to compute the mean and standard deviation across the entire training set as well as to limit the region where the loss is calculated during training. Although not independent, each 4^3 block in $\mathbf{X}$ and its corresponding label in $\mathbf{Y}$ can be regarded as a data point. Consequently, the training set consists of $655\ 360\ 000$ data points, totaling 340 GB (Figure 3.4).

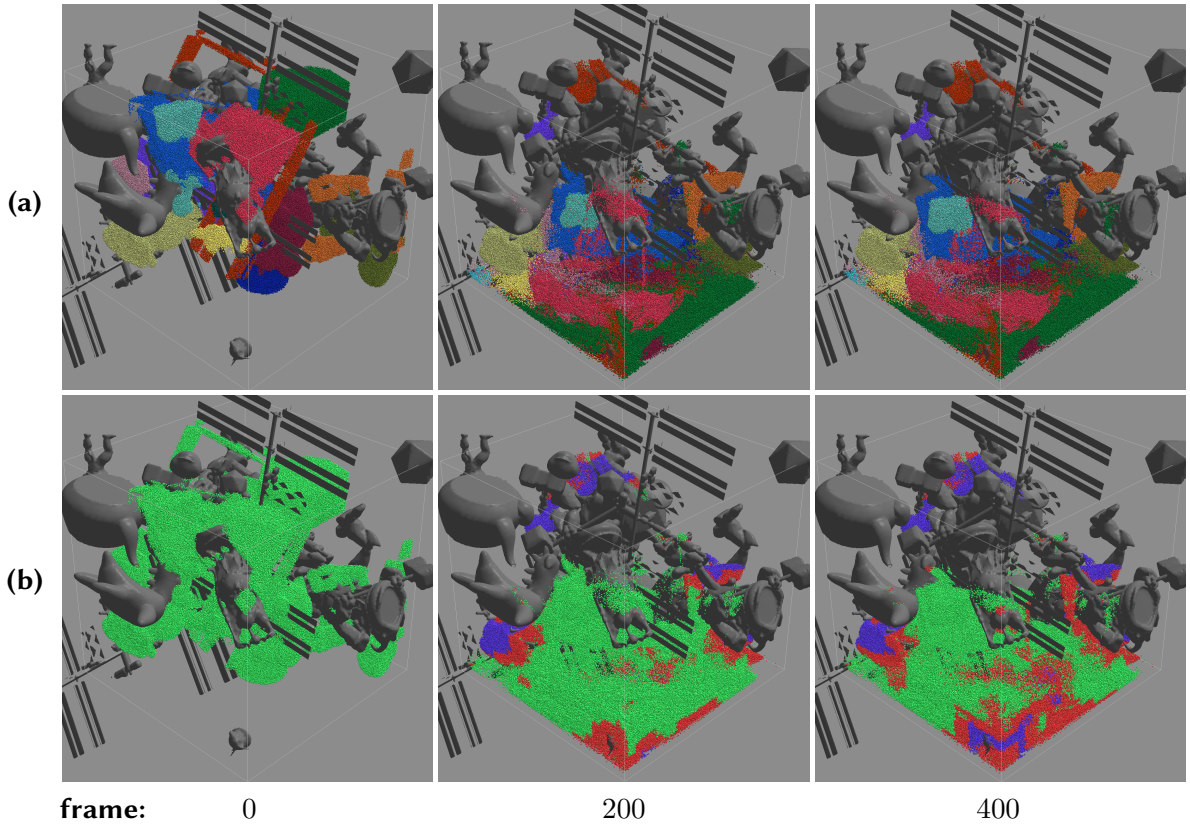


Figure 3.4. One randomly generated scene from our synthetic dataset is shown at frame 0, 200, and 400. (a) Different sources/materials are assigned a unique random color, while the particles' activation is shown in (b): active (green), passive (purple), and boundary (red). The domain is of dimension 10^3 and resolution 128^3. All gray meshes represent VDB signed distance field (SDF) colliders. The particles are all initialized as active with random initial velocities uniformly sampled per component from $(-10, 10)$ and are progressively made passive as the simulation progresses.

In order to perform cross-validation, we have created a separate validation dataset using carefully extracted data from five demo scenes, that are presented in Section 4. The advantage of this approach is that we can evaluate the accuracy of our model on data that is more representative of scenes encountered in production. Additionally, this allows us to assess the similarity between the distribution of the randomly generated scenes used in the training set compared to those hand-crafted for the validation set. If the two distributions are sufficiently close, we expect to observe an increase in accuracy on the validation set as we minimize the loss on the training data. Our validation set comprises 3550 frames, totaling 60.35 GB, which represents approximately

15% of the combined datasets.

The randomly generated scenes all feature dimensions of $10 \times 10 \times 10$ centered at the origin and a resolution of $128 \times 128 \times 128$. These scenes comprise a varying amount of particle sources and colliders sampled from a collection of 17 SDF *OpenVDB* files [MLJ+13] mainly obtained from the *OpenVDB*'s download web page.

While it is common for colliders to be animated in realistic scenes, the ones in our generated dataset are static. This works because we are combining particle and collider speeds in the same channel. As a result, our model is trained to respond to nearby speeds, regardless of its origin.

The duration of 400 frames was determined based on the need for an evenly distributed set of active and passive tiles across the entire training set. Failure to consider this factor may result in an imbalanced dataset, where the number of passive tiles far exceeds the number of active ones or the other way around. Such an imbalance could allow a suboptimal model to score highly in accuracy by always predicting the class with the most occurrences. Since the particles begin with an initial velocity, more active tiles are expected at the start of the simulation. As it progresses and things settle down, the number of passive tiles will increase. After conducting several experiments, we found that a duration of 400 frames strikes an appropriate balance, resulting in a nearly perfect split between active and passive tiles.

The material and physical properties of particles are varied in our training set to enhance our model's ability to handle diverse configurations. The table below presents a summary of the parameters that were randomized on a per-scene basis to build our training set.

Parameter		Uniform sampling range
Source and collider		
VDB position	$\mathbf{x}$	$(-5, 5)$ per axis
VDB rotation	—	$(0, 2\pi)$ per axis
VDB uniform scale	—	$(0.7, 3.8)$
Source only		
VDB count	—	[1,20]
Initial velocity	$\mathbf{v}$	$(-10, 10)$ per axis
Material type	—	$\{liquid,\ chunky,\ solid\}$
Density	ρ	$(100.0, 2000.0)$
Young's modulus	E	$10000.0\rho \times (0.9, 1.1)$
Poisson's ratio	ν	$(0.1, 0.3)$
Hardening	κ	$(5.0, 15.0)$
Collider only		
VDB count	—	[1,40]
Friction	μ	(0.0, 1.0)

Table 3.1. Summary of the parameters randomized for our training set generation.

3.2 Neural Network Design

3.2.1. Architecture. As explained in Section 2.2, MPM necessitates a grid as a *scratch pad* to execute the velocity update step of the algorithm. Therefore, it is natural to exploit this well-organized data structure and build our neural network around it. CNNs are known for their efficiency on grid-like data structures such as images. The convolution operation can be extended to an arbitrary number of dimensions, and so can the CNN architecture. A 3D CNN is highly suitable for our application since we aim to leverage the data's spatial locality to infer tile activation.

Our model takes this grid representation of the simulation state as input of fixed resolution 128^3 and output a predicted activation of size 32^3 at the tile level. The MPM particles possess numerous attributes that could contribute to determining tile activation. However, passing all these attributes to the CNN would create a bottleneck, potentially counteracting any performance gain achieved through tile deactivation.

To identify the channels necessary to our model, we began by providing all channels that

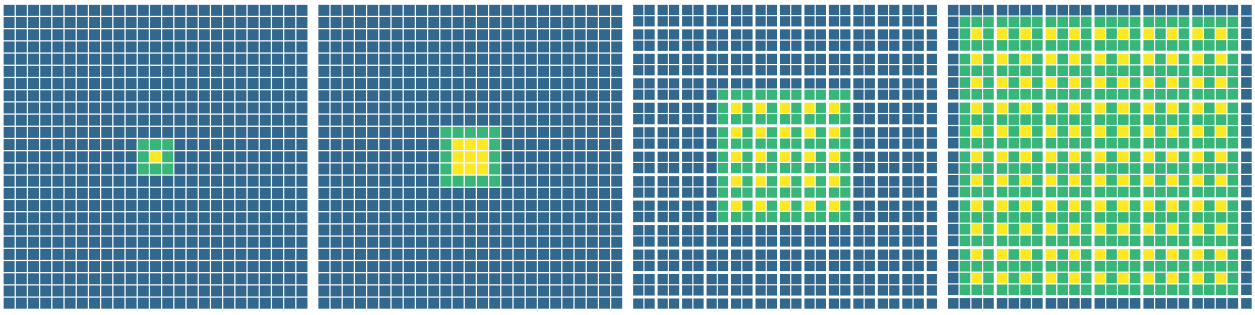
could potentially be employed during inference. We then gradually pruned them through an ablation study. The original set of input channels consisted of: mass m, determinant of the deformation gradient $|\mathbf{F}|$, determinant of the affine velocity $|\mathbf{C}|$, speed $||\mathbf{v}||$, colliders' SDF ϕ, colliders' speed $||\mathbf{v}_{co}||$, and λ, μ the Lamé parameters from continuum mechanics. Through this investigation, we determined that $||\mathbf{v}||$ was the channel that primarily contributed to the model's performance. The affine velocity determinant $|\mathbf{C}|$ also delivered promising results as it is closely related to the velocity. However, it turned out to be less reliable due to its high volatility, often approaching the limits of floating point representation. Consequently, we restricted our model's input to the particles' speed, significantly reducing the volume of data processed during a forward pass.

After conducting a hyperparameter grid search, we have identified the optimal balance between speed and accuracy for our model. We found that using four hidden convolutional layers, each with a kernel size of 3^3, yielded the best performance. Rather than using pooling operations to reduce the input size between layers, we opted to use strided convolution [KSH17] with a stride of two, which effectively decreases the input dimension by two in every dimension. This approach allows the model to learn a more expressive downscaling method rather than relying on ad hoc average or max operations. Additionally, all layers are padded with zeros and employ a ReLU activation function, except for the final layer, which uses a sigmoid activation function commonly used in binary classification problems.

THE POWER OF DEPTH. To make accurate predictions, each tile must be provided with sufficient information about its surroundings. Looking at the above architecture, we might be concerned that such small convolution kernels might miss important information not directly adjacent to a particular tile. It is no coincidence that our network counts four layers, nor that the downscaling operations are performed in the first two. Let us unfold the convolutions backward to assess what the network truly sees (Figure 3.5).

We begin with layer four's 3^3 convolution that produces a receptive field (RF) of 3^3. Moving to layer three, two voxels are added in each dimension for a total of 5^3. At layer two, we reach the first strided convolution, which effectively more than doubles the RF to 11^3. Finally, another

strided convolution at layer one results in an RF of 23^3. The network is, therefore, able to aggregate information from a relatively sizeable surrounding volume, providing plenty of information for tile-level predictions.



(a) Conv 4: RF = 3^3 **(b)** Conv 3: RF = 5^3 **(c)** Conv 2: RF = 11^3 **(d)** Conv 1: RF = 23^3

Figure 3.5. Backward analysis of our CNN receptive field (RF) in 2D. (a) A 3^3 convolution is performed, resulting in an RF of 3^3. (b) The RF is increased to 5^3 after another convolution is performed at layer three. (c) The RF is more than doubled to 11^3 after a strided convolution at layer two. (d) The final RF of 23^3 is achieved after the last strided convolution. The convolution kernels' center is colored yellow, while the rest is colored green. The spaces clustering the cells in groups of 2×2 and 4×4 in (c) and (d) respectively identify the tiles through the change in resolution.

To put it another way, every tile can access a total of 23^3 voxels of input data centered around itself, which is equivalent to having more than two tiles of padding on all sides. However, it is worth noting that this is only possible thanks to the network's depth. We can confirm this by introducing Δ as the resolution width of the input tensor fed to our network. The number of multiplications computed by our CNN can be expressed as follows:

$$g(\Delta) = 3^3 \left(2 \left(\frac{\Delta}{2} \right)^3 + 22 \left(\frac{\Delta}{4} \right)^3 \right). \tag{3.4}$$

Let us now consider an alternative network architecture consisting of a single convolutional layer with an equivalent receptive field. The total number of multiplications required to compute the output of this network can be expressed as:

$$h(\Delta) = 23^3 \left(\frac{\Delta}{4} \right)^3. \tag{3.5}$$

There exists a $\beta \in \mathbb{R}$ such that $h(\Delta) = \beta g(\Delta)$. Let us solve for β.

$$\begin{aligned} h(\Delta) &= \beta g(\Delta) \\ 23^3 \left(\frac{\Delta}{4}\right)^3 &= 3^3 \left(2\left(\frac{\Delta}{2}\right)^3 + 22\left(\frac{\Delta}{4}\right)^3\right)\beta \\ \frac{23^3}{4^3}\Delta^3 &= \frac{38 \cdot 3^3}{4^3}\Delta^3\beta \\ \beta &= \frac{23^3}{38 \cdot 3^3} \\ \beta &\approx 11.85. \end{aligned} \tag{3.6}$$

Thus, regardless of the input resolution Δ, our proposed architecture achieves an equivalent receptive field almost $12\times$ more efficiently than a single-layer CNN, even when accounting for the increase in channel counts that usually comes with deeper networks.

3.2.2. Loss Function. We utilize the binary cross-entropy with logits as our loss function, with a batch size of two frames comprising 32^3 tiles each. The loss function is defined as

$$l(\tilde{\mathbf{y}}, \mathbf{y}) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \cdot \log(\sigma(\tilde{y}_i)) + ((1 - y_i) \cdot \log(1 - \sigma(\tilde{y}_i)))] \tag{3.7}$$

where $\tilde{y}$ denotes the output of our model without the sigmoid activation function $\sigma(\cdot)$ applied, and $N = 65\,536$.

To ensure numerical stability, we incorporate the computation of the sigmoid activation function with the binary cross-entropy using the *log-sum-exp* (LSE) trick. This trick shifts the exponential argument by a constant to prevent overflow or underflow. The LSE is based on the identity

$$
\begin{aligned}
z &= \log\left(\sum_{i=1}^{N} \exp(x_i)\right) \\
\exp(z) &= \sum_{i=1}^{N} \exp(x_i) \\
\exp(z) &= \exp(x^\star) \sum_{i=1}^{N} \exp(x_i - x^\star) \\
z &= x^\star + \log\left(\sum_{i=1}^{N} \exp(x_i - x^\star)\right)
\end{aligned}
\tag{3.8}
$$

where, in our problem, we define $x^\star$ as the maximum value among all x_i. It is worth noting that it is possible to introduce a weight term in the loss function to address the issue of class imbalance between the positive and negative classes. However, we choose not to adopt this approach, as we have full control over the generation of our datasets. Instead, we strive to maintain a roughly equal representation of both classes, which is expected to promote the best generalization performance.

3.2.3. Regularization. Effective regularization techniques are essential when the capacity of our network significantly surpasses the information available in the training set. Despite the existence of many advanced regularization methods, increasing the amount of data remains the most effective approach, provided that such data is readily available. Fortunately, since we generate the data, we have access to an unlimited supply of it. It is noteworthy that for our model to be lightweight enough for online inference, it is constrained to a relatively small capacity. Therefore, we find ourselves in an ideal situation where it is virtually impossible for our model to overfit the training set, as it can only capture the underlying core features contained in the supplied noisy data and lacks the capacity to capture the noise.

3.2.4. Hyperparameters. To optimize our model, we employed the Adam optimizer with a learning rate of $\eta = 0.0005$ and a moving average parameters of $\beta = (0.9, 0.999)$. Although the epoch count is set to 1000, in practice, the training process is early stopped as soon as convergence is reached.

Table 3.2 summarizes the hyperparameters of our model and the values tested during the grid

search.

Hyperparameter	Grid search space	Optimal value
Kernel size	3 / 5 / 7	3
Channel count multipliers	1,3,1,1 / 2,5,2,1 / 4,10,4,1	2,5,2,1
Network depth	3 / 4 / 5	4
Learning rate	0.00025 / 0.0005 / 0.001	0.0005
Learning rate scheduling	0.99 / 0.995 / 0.999	0.995
Batch size	1 / 2 / 4	2
Dataset normalization	true / false	true

Table 3.2. Summary of our hyperparameter grid search including the resulting optimal values.

3.3 Training the Network

To ensure maximum flexibility and ease of experimentation, we implemented the training code in *Python 3.9* using the *PyTorch 1.9.0* library [PGM+19]. Training was conducted on *Nvidia Quadro RTX 8000* GPUs. Our dataset, described in Section 3.1.4, consisted of approximately 400 GB of simulation data divided into training and validation partitions of 85% and 15%, respectively.

3.3.1. Problem-Specific Considerations. To achieve convergence, we utilize a learning rate scheduling strategy, which updates the learning rate at each epoch t according to the expression $\eta_{t+1} = 0.995\eta_t$. This process continues until early stopping criteria are met, which typically occurs at around epoch 600.

We compute global statistics across the entire training set to normalize each input channel of the CNN. Although the mass channel is not used as input, it is used as a mask to consider only voxels containing particle information in order to prevent a bias toward zero. Thus, the input is always normalized using the computed mean and standard deviation prior to being processed by the network. This prevents our model from learning disproportionately large weights, which can lead to poor generalization. The calculated statistics are saved as a JSON file and will be loaded by the solver for inference.

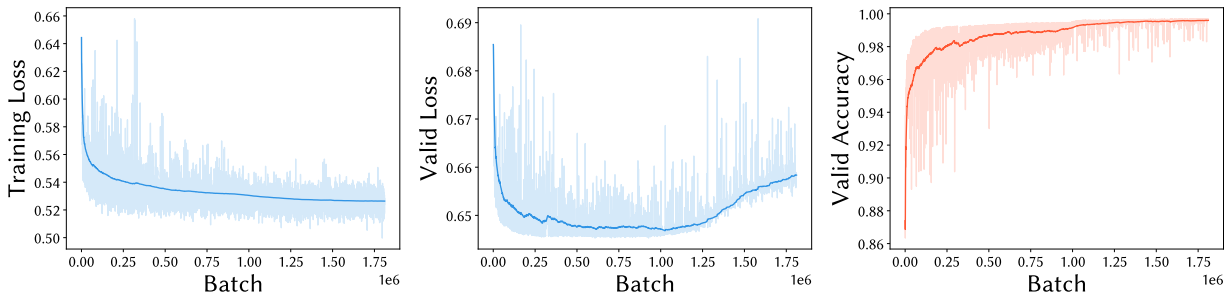
As previously indicated, our model's capacity is constrained for efficiency reasons, so we aim

to focus training efforts solely where they are most effective. Thus, the binary mask derived from the mass channel is utilized as a stencil on our network's predictions to prevent it from being penalized for incorrect predictions in empty regions, which ultimately are ignored by our solver. This same masking technique is also applied during cross-validation.

Early stopping is employed during training such that whenever our model achieves a new high score in validation accuracy, the model is saved to disk, replacing the previous version. Typically, the accuracy score ceases to improve after about 600 epochs of training, which takes roughly 36 hours (Figure 3.6).

As described in Section 3.1.4, the speed of particles and colliders is combined in the same channel. In earlier experiments, we attempted to train the model by isolating the speed of moving colliders in a separate channel, but this approach proved to be ineffective. The model learned to ignore the channel containing colliders' speed since it could consistently get the same information by looking at the particles speed near moving colliders. Consequently, the model tended to rely solely on the channel with particles speed, which provided more reliable results in general. Unfortunately, this approach prevented a moving collider from activating passive particles at inference time if particles were already all passive. This issue is easily addressed by combining both speed channels through a max operation.

When the training is done, we generate a JSON file containing the weights and biases of the model, which can be utilized by our solver for inference, as described in Section 3.4.



(a) **(b)** **(c)**

Figure 3.6. We present the training and validation performance of our model. (a) Specifically, the plot illustrates the training loss as a function of the optimization step. The cross-entropy loss on the training set decreases monotonically on average throughout the 1.75 million optimization steps, corresponding to 647 epochs. (b) The plot depicts the same loss function evaluated on the validation set. The loss starts to increase at around 1 million steps, which usually indicates overfitting on the training data. (c) However, our best validation accuracy of 99.7% is achieved after 1.674 million optimization steps or 598 epochs. Since accuracy is the primary objective, the increase in the validation loss (b) should not raise any concern.

3.4 Inference with the Trained Network

3.4.1. Forward Pass as Part of the Solve. In order to maximize performance and flexibility, we decided to directly implement the fundamental building blocks of a CNN in *CUDA/C++* instead of relying on the *Pytorch C++ API* to query the neural network. Thus, we can mirror our *PyTorch* implementation in the solver while retaining complete control over the implementation details.

The rigorous development process involves maintaining a comprehensive suite of unit tests to verify the correct implementation of each functionality. This approach enables us to ensure that our implementation harmonizes seamlessly with the memory layout of our data and exploits all available optimizations.

There are several reasons why we opted to develop our version of the CNN instead of relying on *TorchLib*. Firstly, *PyTorch* typically expects the input to have a fixed size to perform convolution as a tensor product. However, in our case, we dynamically reshape the domain as the simulation progresses, necessitating constant changes in the input size. Although not currently

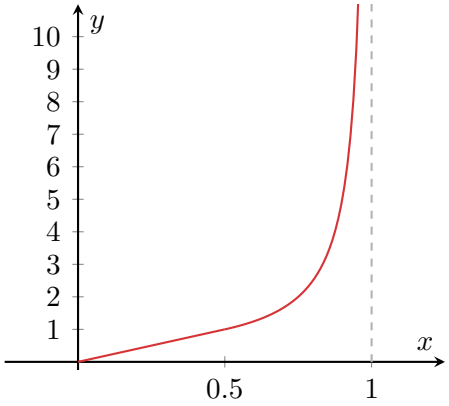
integrated into the system, the MPM background grid may evolve into a sparse dynamic structure, further complicating the integration with the *PyTorch C++ API*. Additionally, at the time of writing, dynamically linking against the massive 4 GB *TorchLib* library is the only available option due to numerous issues with the static build. Considering that our prototype is about 10 MB in size and only uses a tiny fraction of the *PyTorch* library, it is illogical to package it with such a large dependency. Moreover, the *PyTorch C++ API* documentation is still lacking, making it difficult to experiment with.

In contrast, our implementation exhibits unparalleled flexibility and optimization potential. It can handle different resolutions, provided that they are divisible by four in all dimensions, which is required for tile alignment. It is also not restricted to static Eulerian grids, suggesting that transitioning to a sparse structure like *SPGrid* [SABS14] should be relatively intuitive, as our design was constructed with this prospect in mind.

To query the CNN during simulation, we first load the trained parameters and statistics from disk. Afterward, we scatter the particles' mass m and momentum magnitude $m||\mathbf{v}||$ to a tensor object, following a procedure similar to the GRIDTOPARTICLE( ) kernel (Algorithm 2 at page 56). Subsequently, we iterate on the voxels of the tensor, dividing the momentum magnitude by the mass to get the speed when $m > 0$, in a similar manner to the GRIDVELOCITYUPDATE( ) kernel (Algorithm 2). In the same kernel, we also sample the speed of the moving colliders $||\mathbf{v}_{co}||$ and combine them with the speed of the particles through a max operation. We then proceed to normalize the input channels using the previously loaded statistics from disk and divide the outcome by the domain scale Ω according to the heuristic introduced in Section 3.1. The mass channel is then stripped out by copying all the other channels (only the speed, in this case) to a smaller tensor that will be handed to the CNN to perform its forward pass on the data. Finally, we iterate over each valid tile (tile containing at least one particle regardless of its activation) to extract the prediction from the output tensor. At this point, the activation attribute γ is promoted to the particle level for the solver to calculate a single timestep forward before repeating the cycle.

As noted earlier, it is possible to influence how the trained neural network perceives the amount of motion in the scene, despite being trained with the fixed tolerance as described in Section 3.1. Similar to dividing the speed by Ω, it is also possible to multiply it by an arbit-

rary constant before passing it to the network. Therefore, we expose a user-defined parameter bounded to the $[0, 1)$ range with a default value of 0.5. This parameter is then passed through the function $f : [0, 1) \to [0, \infty)$, defined as



$$f(x) = \begin{cases} 2x & \text{if } x \in [0, 0.5] \\ \frac{1}{2-2x} & \text{if } x \in (0.5, 1.0) \end{cases} \tag{3.9}$$

and then as a signal multiplier α to the speed passed as input to our model. This feature can be leveraged in various ways. It can amplify the signal, making the model more sensitive to subtle motions such as ripples on the water surface. It can also be used to freeze parts of the simulation more aggressively, to keep things stacked without drifting or to enable faster preview of the simulation. This level of control post-training lessens the burden of choosing the perfect tolerance δ to generate the dataset used for training, and yields a more robust system that can adapt to various situations. The entire procedure per timestep is outlined in Algorithm 2 below.

Algorithm 2 CNNINFERENCE()

$\mathcal{X} \leftarrow$ SCATTERTOTENSOR$(m_{\mathcal{P}}, ||\mathbf{v}_{\mathcal{P}}||)$ ▷ Particles' mass and momentum norm is scattered to a tensor
$||\mathbf{v}_{\mathcal{X}}|| \leftarrow ||\mathbf{v}_{\mathcal{X}}||/m_{\mathcal{X}}$
$||\mathbf{v}_{\mathcal{X}}|| \leftarrow \max\{||\mathbf{v}_{\mathcal{X}}||, ||\mathbf{v}_{co}||\}$ ▷ Colliders' speed is maxed with the tensor's speed
$||\mathbf{v}_{\mathcal{X}}|| \leftarrow ||\mathbf{v}_{\mathcal{X}}|| \cdot \alpha/\Omega$
$\mathcal{Y} \leftarrow$ CNNFORWARD$(\mathcal{X})$
$\gamma_{\mathcal{T}} = \lfloor \mathcal{Y} + \frac{1}{2} \rfloor$ ▷ Output from the network is extracted per tile
$\gamma_{\mathcal{T}} \leftarrow$ SETBOUNDARYTILES$(\gamma_{\mathcal{T}})$ ▷ Boundary tiles are detected based on neighbors
$\gamma_{\mathcal{P}} \leftarrow \gamma_{\mathcal{T}}$ ▷ Activation is promoted from tiles to particles
if $\gamma_{\mathcal{P}} = 1, 2$ **then**
    $\mathcal{G} \leftarrow$ PARTICLETOGRID$(\mathcal{P})$ ▷ Only applied to active and boundary particles
if $m_{\mathcal{G}} > 0$ **then**
    $\mathcal{G} \leftarrow$ GRIDVELOCITYUPDATE$(\mathcal{G})$
if $\gamma_{\mathcal{P}} = 1$ **then**
    $\mathcal{P} \leftarrow$ GRIDTOPARTICLES$(\mathcal{G})$ ▷ Only applied to active particles

3.5 Particle Scheduler

The scheduler is an essential component in our system, enabling efficient particle processing based on their activation attribute γ. Specifically, the scheduler consists of an array that stores the indices of all particles in the simulation, sorted according to their activation attribute. We call this array the schedule, as it defines the order in which the particles are processed. This approach facilitates tracking the particle distribution across the three activation classes and enables launching kernels on the GPU with the appropriate thread count configuration. Without the scheduler, many threads would be launched with passive particles only to be terminated as soon as they reach the kernels.

3.5.1. Non-Coalesced Access vs. Sorting. One potential issue with the scheduler is the additional level of indirection it introduces. Prior to its use, each warp of 32 threads would be assigned to a consecutive sequence of particles continuously laid out in memory, where a single memory transaction could retrieve all the data required by the warp. However, with passive particles, only one particle out of the 32 might be active, making the entire warp idle and waiting for this solitary thread to finish. In contrast, with the scheduler, a single warp will persistently be assigned

non-passive particles. However, they might be dispersed far apart in memory, resulting in non-contiguous access that requires multiple memory transactions, which can significantly degrade the overall application performance (Figure 3.7).

To tackle the issue of non-coalesced memory access when using the scheduler, we can re-arrange the particle attribute arrays on the device according to the order prescribed by the scheduler. By doing so, memory access can be coalesced, thereby minimizing the number of memory transactions required. However, this approach comes at the cost of sorting the structure of arrays containing particle data. In cases where the number of particles is substantial, sorting the arrays in parallel might not be feasible due to the associated memory overhead. As a result, the arrays must be processed one at a time, further reducing the procedure's efficiency.

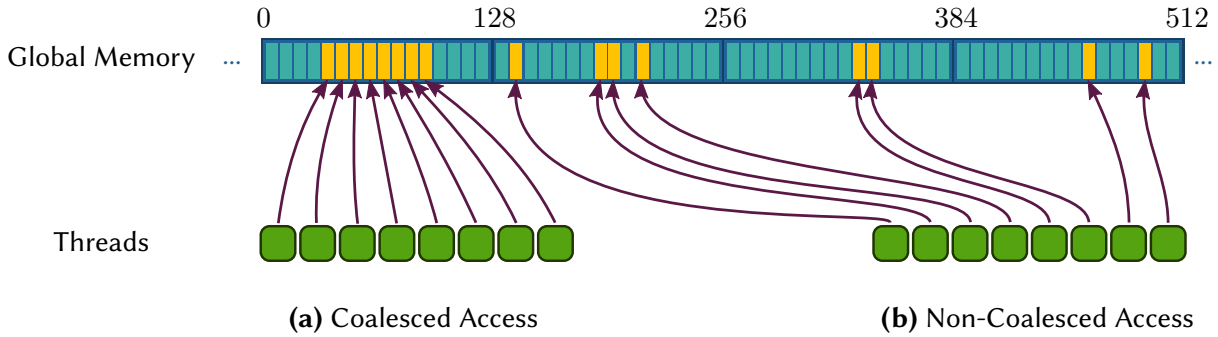


Figure 3.7. (a) Threads within a warp can access data coalesced in memory, allowing them to retrieve all necessary data within a single 128-byte transaction. (b) When data are scattered in memory, threads must access it in a non-coalesced manner, leading to multiple memory transactions and degraded performance.

Through our experimentation, it has been demonstrated that maintaining the particles in a non-contiguous layout is globally less expensive than rearranging all arrays each time a new schedule is made available.

3.6 Methodology

3.6.1. Implementation. The prototype was implemented in *C++ 17* and *CUDA 11.7*. It was primarily designed and validated on *Linux* using an *Nvidia GTX 1080 Ti* GPU. Nevertheless, it is expected to run correctly on other platforms that support *Nvidia* GPUs with compute capability

of 6.0 and higher, including *Windows*. As mentioned earlier, the neural network experimentation was conducted using the *PyTorch* framework, while the simulation inference was developed in *CUDA/C++* to ensure maximum adaptability. The program is reliant on numerous dependencies, such as:

glm/cci.20220420,
glfw/3.3.8,
glad/0.1.36,
imgui/cci.20220621+1.88.docking,
spdlog/1.10.0,
fmt/8.1.1,
gtest/cci.20210126,
eigen/3.4.0,
nlohmann_json/3.11.2,
openvdb/9.0.1,
boost/1.79.0,
onetbb/2020.3,
cxxopts/3.0.0, and
partio/1.14.6.

These are mainly managed by the *Conan* package manager. Alongside these libraries, we have integrated certain helper functions from the *Nvidia SDK Samples* into our codebase. For the singular value decomposition (SVD), we utilize the GPU implementation created by Kui Wu [GWW+18] of the algorithm outlined in [MST+11]. To ensure portability across different platforms, we employ *CMake 3.18* as our build system. Our solver, comprised of over 35 classes, consists of roughly 10 000 lines of code, excluding third-party libraries.

3.6.2. Prototype Features. The *OpenVDB* library plays a central role in our application, with colliders and particle sources relying exclusively on VDB signed distance fields. These VDB files can be animated, regardless of whether they are used as colliders or emission sources. One ad-

vantage of this representation is the ability to resample the sparse grids according to the domain resolution, which will be further discussed in subsequent sections.

Particle Emission. The generation of particles is solely enabled via the usage of *OpenVDB* files, which provides the benefit of accommodating any shape as opposed to analytical signed distance functions. This format is widely adopted within the visual effects industry, and thus, numerous models are freely accessible online and can be directly utilized in our solver. For specific requirements, *Houdini Apprentice*, a free software, can facilitate the conversion of any mesh into a VDB file.

Following the guidelines presented by Stomakhin et al. [SSC+13] in their research on snow, we have adopted the strategy of emitting on average eight particles per Eulerian grid cell, regardless of the material being simulated. To achieve this, we utilize the capabilities of the *OpenVDB* library, which allows us to resample the source VDB to half the voxel size of the Eulerian grid $\Delta x/2$ before performing stratified sampling on it. Our approach involves creating a single particle instance within each voxel based on the condition that $\phi(\mathbf{x} + \epsilon) < 0$. Here, $\mathbf{x}$ is the central position of the VDB voxel, $\epsilon = (\xi_1 \Delta x, \xi_2 \Delta x, \xi_3 \Delta x)$, and $\xi_i \in (-\frac{1}{4}, \frac{1}{4})$ is a uniformly distributed pseudo-random number. By introducing jittering with ξ_i, we can break up any unnatural alignment of particles while ensuring adequate material coverage and reduced overlap thanks to the stratified sampling method (Figure 3.8). Note that similar and potentially better results could be obtained using blue-noise-based sampling. Furthermore, this even distribution in space positively impacts the performance of the atomic calls in the ParticleToGrid( ) kernel.

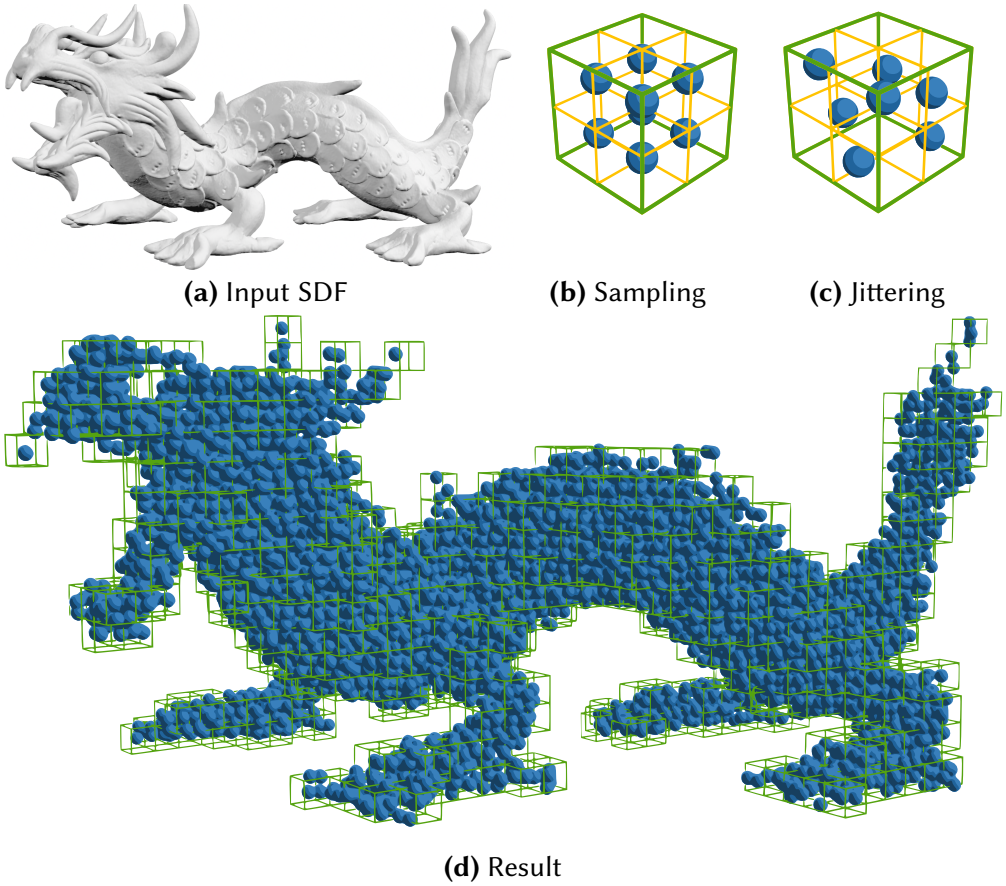


Figure 3.8. Sourcing particles using an SDF representation of the Stanford dragon from *OpenVDB*'s download page. (a) The sourcing mechanism requires a VDB SDF as input. (b) The SDF is resampled to half (colored yellow) the Eulerian grid voxel size (colored green), then a particle is created at the center of each voxel in the narrow band. (c) Created points are then jittered in space to a maximum distance of a quarter of the Eulerian grid voxel size. Then, the points that land within a positive region of the SDF are deleted. (d) The resulting point cloud from this process overlayed with the wireframe of the Eulerian grid voxels containing the particles.

In our simulation, every particle originating from the same source shares a unique material, establishing its physical properties. Additionally, an initial velocity that applies uniformly to all particles upon their emission can be prescribed per source.

Our system provides continuous emission and deletion of particles. In the continuous emission case, the device arrays are only reallocated when the current particle count exceeds the previous device allocation. The data in the old array is then copied (device-to-device) to the new extended array, followed by a copy (host-to-device) of the newly generated source particles at the

end of the new array. If particles need to be deleted, all arrays containing them are sorted based on their $killed$ attribute before updating the particle count to only consider the particles still alive. Since the sort-by-key operation cannot be done in place, deleting also triggers a reallocation. This waste of performance is mitigated by only pruning the killed particles when a certain proportion of dead particles is reached. As mentioned in Section 3.5, these reallocations must be done sequentially to prevent memory depletion on the device. To further reduce the impact of these copies, we only perform them on whole frames. While particles can be killed on substeps, the solver simply ignores them before they are eventually cleaned up. Changes in particle count must be communicated to other system components, such as the *Scheduler* and the *Ground-Truth Generator*, which maintain data structures that must stay synchronized with the particle system.

Exporting particles to disk is possible using the *PartIO* library, developed by *Walt Disney Animation Studios* [Wal]. The library allows efficient storage of particle attributes in the *BGEO* file format, which can be read and processed in *Houdini*. This functionality facilitates the visualization, modification, and rendering of simulation data.

Collider Objects. While sampling the SDF of a collider, we do not conduct tri-linear interpolation but only round down the sample position within the voxel space of the collider. Although this approach results in a loss of precision, we overcome this issue by resampling the collider to half of the Eulerian grid voxel size $\Delta x/2$, similar to the procedure employed for particle emission. This step increases the device memory usage for each collider, but it reduces the computational cost of sampling at each timestep. All collider queries are executed on the device utilizing the *NanoVDB* library [Mus21].

In order to maximize the speed and accuracy of our solver, we restrict animation to rigid transformations applied to static VDBs. This approach offers superior performance, requiring resampling the VDB file only once and then uploading it to the device for use. Allowing dynamic VDBs would entail this process at each frame, resulting in a substantial drop in performance. Additionally, interpolated transformations provide greater precision, as we can use spherical linear interpolation to interpolate the static colliders' orientation through quaternions. Since our solver employs explicit integration, we make many tiny substeps, possibly leading to hundreds

of collision queries between every rendered frame. Reliable interpolation is thus especially useful, as it can prevent unwanted artifacts that may be noticeable with direct SDF interpolation. Furthermore, interpolation of transformations makes the collider's velocity very precise, as it is calculated at each timestep based on the interpolated animation. Therefore, if there are enough substeps, a fast-spinning wheel can create a beautiful circular arc that carries with it the nearby particles through dynamic friction. This kind of phenomenon cannot be captured with an implicit solver taking large timesteps (Figure 3.9).

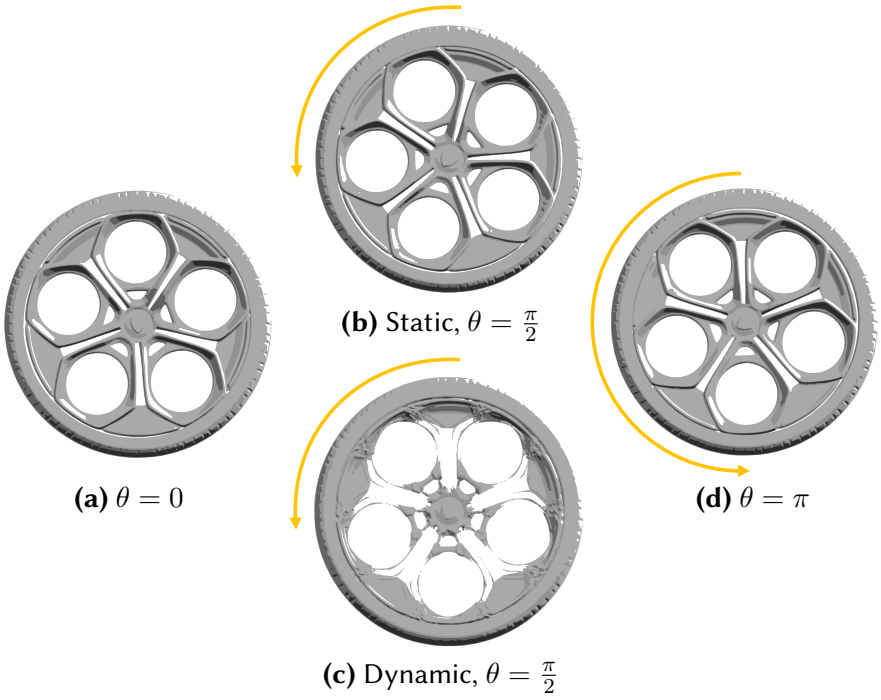


Figure 3.9. All SDFs are rendered as surfaces of level set $\phi = 0$ using a grey diffuse shading such that white areas represents positive signed distance values. Consider two SDFs with velocity fields that depict a rotating wheel: (a) with $\theta = 0$, and (d) with $\theta = \pi$. We compare two interpolation methods to generate an intermediate result. (b) Using static SDF with interpolated transformations, where the orientation is determined via spherical linear interpolation, provides the expected outcome. (c) Utilizing two different SDF files making it unavoidable to directly interpolate the dynamic SDF values, leads to suboptimal results.

ORIENTED DOMAIN. Although our current implementation employs a dense MPM background grid, non-axis-aligned domains are supported, which can more tightly enclose the simulated region. Internally, the simulation occurs in an axis-aligned domain, meaning that external forces

such as gravity must be oriented using the inverse transformation of the domain. Likewise, translated domains are brought back to the origin through the same mechanism, promoting enhanced numerical precision, even if the domain is far from the origin in the scene. The particle positions and the domain boundaries are only transformed to world space for real-time display and export.

The domain can be equipped with open boundaries that effectively remove particles as they transit beyond them. This feature is particularly useful for fast-moving particles that would otherwise disturb or obstruct the rest of the simulation.

Multi-Material. To test the versatility of our approach, we have implemented three different material types that can interact with each other: $liquid$, $chunky$, and $solid$. $Chunky$ material exhibits plastic deformation similar to snow and soil, while the material referred to as $solid$ is more commonly known as hyperelastic in the literature. Each material type can be customized to exhibit a broad range of behaviors by adjusting its physical parameters, which include density (ρ), Young's modulus (E), Poisson's ratio (ν), and hardening (κ, only for chunky materials). By allowing these diverse materials to interact, we can produce results rich in complexity and variations to model a wide range of phenomena commonly encountered in production.

Scene Description. Our solver processes scene files with the ".ice" extension. These scene descriptions are written in human-readable ASCII format and can be fully customized to represent a wide range of phenomena, as illustrated in Chapter 4. The scene description contains various settings and links to files, such as particle sources, colliders, domain, solver, camera, and caching. However, geometric data is excluded to keep the file concise and easy to interpret. VDB colliders and sources, along with their respective animation files, are referred to using absolute links to external files.

3.6.3. Visualization and Interactivity. This section describes the scene creation process and the *Houdini* bridge that allows users to quickly configure and launch scenes from an artist-friendly environment. Additionally, we discuss the real-time visualization of the simulation achieved using a simple Phong shading model and fake sphere instancing.

Scene Creation. A scene description can be written manually, as we did during the development phase of the solver. The only requirements are that the file complies with the JSON syntax and includes all the attributes our application expects.

In order to streamline scene creation, a set of *Houdini Digital Assets* (HDA) has been developed (Figure 3.10). This toolkit can be used in the free version of *Houdini Apprentice* and features five nodes: *Ice VDB Cache*, *Ice Collider Config*, *Ice Source Config*, *Ice Domain*, and *Ice Solver*. The *Ice VDB Cache* node allows the user to save a static VDB file to disk and pack additional data expected by nodes downstream. The *Ice Collider Config* and *Ice Source Config* nodes are meant to be connected directly after the *Ice VDB Cache* node and provide the necessary properties to configure the incoming packed VDB as a collider or source, respectively. The *Ice Domain* node defines the simulation domain's position, dimensions, resolution, and open/close boundary configuration. The *Ice Solver* node consumes the input streams from all previous nodes with its three inputs expecting the *Ice Source Config*, *Ice Collider Config*, and *Ice Domain* as first, second, and third input respectively. Colliders and sources can be animated between the *Ice Source/Collider Config* and *Ice Solver* nodes. The source can also be provided with a velocity point attribute `v` to define its initial velocity. The *Ice Solver* interface provides many parameters to configure the solver, the simulation duration, and the caching options. A "Build" button located at the top of the interface processes all inputs across the defined frame range and writes the scene and animation files to disk.

In addition to the "Build" button, the interface also includes "Launch" and "Kill" buttons. These buttons respectively launch our solver with the generated scene as a subprocess or terminate a previously launched subprocess still running. The ability to quickly configure and launch scenes from within a powerful application such as *Houdini* significantly expedites the process of scene creation compared to manual text editing. Furthermore, our solver runs almost exclusively on the GPU, allowing a user to inspect and manipulate the data being written to disk in *Houdini* using the CPU while the solver still operates in the background.

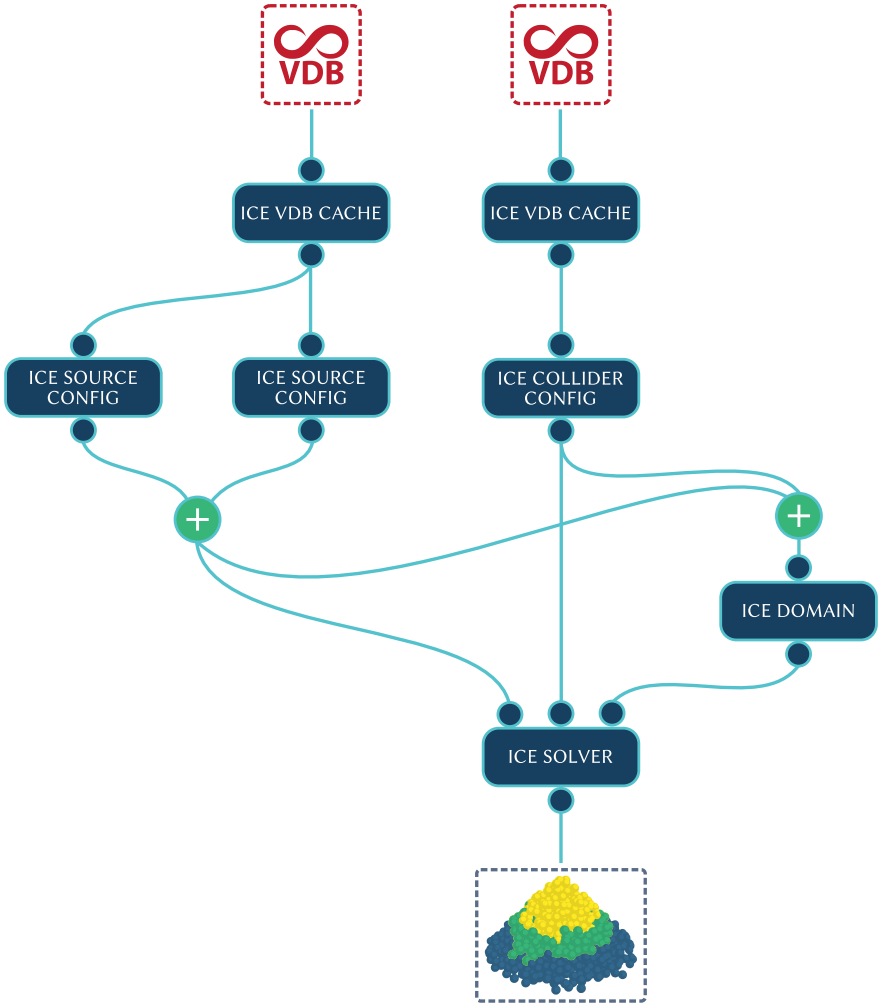


Figure 3.10. An example of a scene graph, as seen in *Houdini*, showcasing the five available nodes in the toolkit that are utilized to generate novel scenes. VDB files are expected to serve as input, while the simulation data is outputted via the *Ice Solver* node, considering that caching has been enabled.

Rendering. Real-time rendering of the simulation is achieved through the utilization of *OpenGL 3.3*, and a simple Phong shading model [Pho75], which is applied to both triangle meshes and particles. We leverage *CUDA*'s interoperability with Graphic API to accelerate the particle visualization process. A memory buffer for particle visualization is initially allocated via *OpenGL* and subsequently manipulated with *CUDA* to update particle positions and other relevant attributes for display.

To provide a more accurate visualization of the shape defined by the particles in our simu-

lation, we adopt the strategy of rendering them as shaded spheres. This approach allows us to better anticipate the final appearance of the simulation before it is converted to an SDF and then meshed in a post-processing stage. Given the large number of particles that need to be rendered, copying unique geometry on individual particles would be impractical; even instancing geometry could be too resource-intensive for real-time rendering, depending on the complexity and scale of the scene. To overcome these issues, we create the illusion of sphere instancing by computing sphere normals on sprites cut out using a disk stencil (Figure 3.11).

We set the disk's radius r to half the sprite width and discard any fragment outside according to $x^2 + y^2 > r^2$. In order to shade the disk as a sphere, we calculate normals of the hemisphere that would be projected onto the x, y disk using the sphere equation and solving for z such that

$$r^2 = x^2 + y^2 + z^2 \implies z = \sqrt{r^2 - x^2 - y^2}. \tag{3.10}$$

Subsequently, we can obtain the normal of the sphere in camera space using

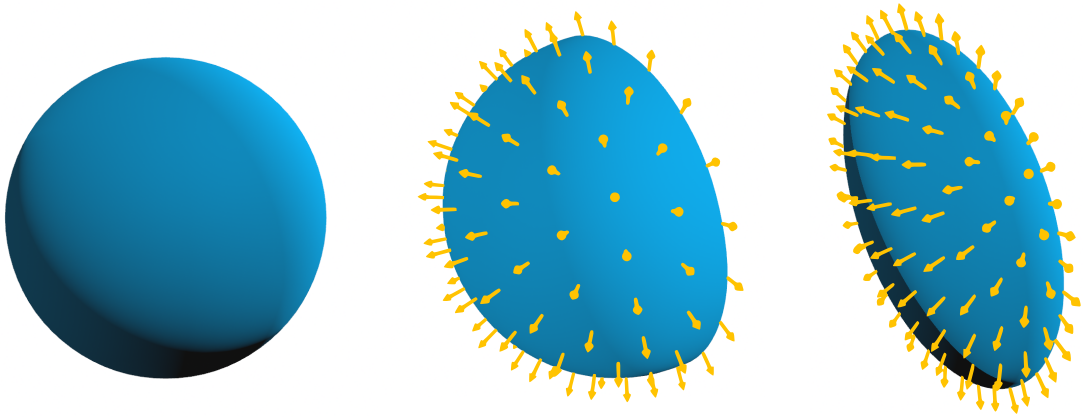
$$\mathbf{n} = \frac{1}{\sqrt{x^2 + y^2 + z^2}}(x, y, z). \tag{3.11}$$

However, for shading to be carried out correctly, we need to bring the vector into world space by multiplying it with the transpose of the view matrix $\mathbf{V}$. Notably, $\mathbf{V}$ is the inverse of the camera transformation matrix $\mathbf{M}$ in the graphics pipeline. Therefore, the world normal can be expressed as

$$\mathbf{n}_{world} = \mathbf{n}_{view} \times (\mathbf{M}^{-1})^{\top} = \mathbf{n}_{view} \times \mathbf{V}^{\top}. \tag{3.12}$$

The present approach to particle rendering introduces a negligible overhead on the rendering time and supports as many particles as the GPU can handle rendered as pixels. It provides an efficient preview of the simulation without the expenses of geometry instancing, which may become unbearably slow when the number of particles in the scene reaches tens of millions. To gain some perspective, our visualization system supports more particles rendered as spheres in real time than *Houdini*. However, this technique is still flawed when it comes to sphere intersec-

tion handling. When a sphere moves in front of another, it instantly jumps from behind due to the uniform depth assigned to each underlying disk. Nonetheless, this visual artifact is almost imperceptible in practical scenarios that involve just a moderate number of particles since those are usually very small in screen space.



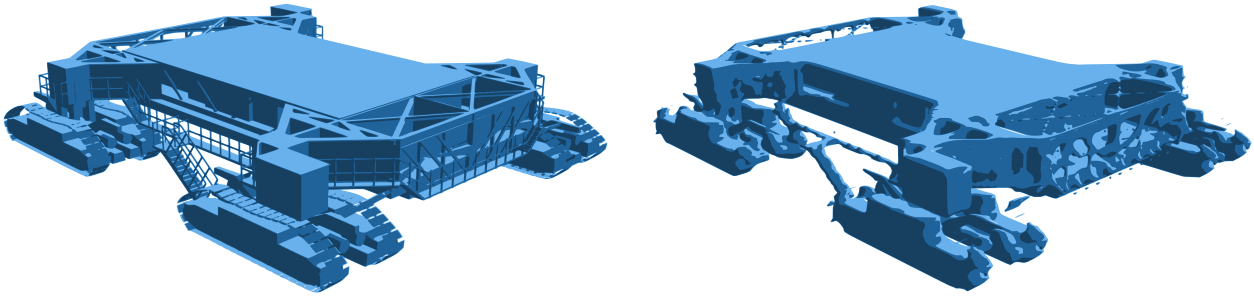
(a) Disk shaded as sphere **(b)** Hemisphere debug view **(c)** Disk debug view

Figure 3.11. (a) Particles are rendered using a diffuse shading as front-facing disks with sphere normals using sprites. Typically, the shading obtained in (a) is achieved with a hemisphere, as shown from the perspective view in (b). We calculate the sphere normals on the disk as seen from the same angle as (b). This debug view (b, c) enables us to distinguish between both geometries and expose the trick. However, in practice, the sprite always faces the camera, and both the hemisphere and the disk will appear indistinguishable from each other at render time (a).

The particles are assigned colors either randomly based on their material index or by utilizing their activation attribute γ. The colors green, purple, and red, respectively, represent the states of active, passive, and boundary particles.

In the current implementation, colliders' SDF are meshed and rendered as triangles after they are resampled to a voxel size of $\Delta x/2$. This approach takes advantage of the efficient hardware rasterizer of the GPU and is more straightforward than implementing our own SDF ray marcher. In contrast with using a precomputed triangle mesh of fixed resolution, the resampled meshed colliders provide the user with an indicator of the level of detail to expect from the simulation (Figure 3.12). For high-resolution simulations, the colliders will be resampled very finely up to its original resolution, whereas, for low-resolution simulations, the resulting colliders will be

smooth, with high-frequency details being mostly eliminated. This visualization is yet another feature that can increase productivity and accelerate troubleshooting. As the VDB remains static, the meshing process only occurs once during scene initialization and does not slow down the solver between timesteps.



(a) High resolution at voxel size = 0.1 **(b)** Low resolution at voxel size = 2.0

Figure 3.12. A rendering of the meshed crawler model available on *OpenVDB*'s download page. The SDF is visualized in two configurations: (a) The SDF is meshed at its native voxel size of 0.1. (b) The SDF is resampled to a voxel size of 2.0 before being meshed. This visualization enables a qualitative appraisal of the loss in detail due to the SDF's downsampling.

USER INTERFACE. Upon launching the scene, the solver displays a GUI that permits the user to modify many settings related to how the simulation is performed. The GUI provides an ideal platform for debugging and experimenting with the available options without affecting the original scene description saved on disk. In addition, the user can navigate through the 3D scene while the simulation is in progress or paused and even step frame by frame to explore specific areas of interest in greater detail.

Due to limited memory resources on the GPU, we provide the user with the option to define the simulation's resolution in terms of megavoxels rather than the conventional voxel size, which can be challenging to adjust due to its cubic relationship with memory requirements. Expressing the resolution in terms of megavoxels offers linear memory scaling, making it easier to manage, particularly when approaching the limits of the GPU's capacity.

The GUI also comes equipped with profiling information to monitor execution times of both the GPU and CPU. Additionally, the distribution of the tiles' activation (i.e., active, passive, and boundary) is displayed as percentages, which can serve as a valuable metric for adjusting the

signal multiplier α. For instance, if the particles appear static and half of the tiles are still active, it may be worthwhile to reduce α to more aggressively halt the simulation and boost performance.

3.6.4. System Optimization. This section briefly outlines noteworthy software engineering optimizations included in our prototype.

Memory Management. To reduce the number of memory transactions made during each kernel call, the data is arranged in memory using a structure of arrays (SoA) format. When a warp accesses data from an array, it will often retrieve the necessary data for all 32 threads if the memory is accessed contiguously. Unfortunately, this does not occur when arrays of structures (AoS) are used. The SoA format enables data to be reallocated progressively in a memory-efficient manner to reduce overhead. This can be particularly beneficial for operations that cannot be performed in place.

As previously mentioned, sequential processing is necessary to prevent memory exhaustion when reallocating multiple arrays. The same principle applies when sorting arrays. A first array is used as keys to generate a permutation array, which is subsequently utilized to rearrange each of the remaining arrays one by one. As a result, memory is gradually released as it is no longer required.

To optimize memory usage, the physical characteristics of simulated materials are stored as separate objects in memory and are only referenced to by particles. This is more efficient than assigning a unique material description to each particle, which would consume a significant amount of memory on the device.

Network Inference. Although a single forward pass with our CNN is relatively inexpensive, the accumulated cost of evaluating the network at each timestep can quickly become significant. To address this issue, we propose decoupling the network evaluation from the timestep frequency. This enables the user to evaluate the solver much more frequently than the neural network, thus amortizing the cost of network evaluation across multiple timesteps. A good convention is to evaluate the network at each rendered frame, which ensures that all substeps in-between stick to the same prediction. In practice, this approach works very well since the prediction is made at the

tile level, and the timesteps in explicit MPM are small, which minimizes late activation artifacts. This amortized cost enhances the competitiveness of our method compared to traditional MPM with virtually no loss of accuracy.

As discussed earlier, increasing the depth of the neural network and using small convolution kernels provide the model with an appropriate receptive field to capture the dynamics of the simulation while maintaining a low computational cost.

Maintaining a minimal number of input channels is also an important factor to consider. First, aggregating data into a tensor that can be fed to the network is a computationally expensive task that scales linearly with the amount of data to be scattered. In addition, the number of input channels usually impacts the shape of the network's hidden layers. The subsequent layers must have sufficient capacity to process the incoming data and create their latent representation. Consequently, the number of parameters to be trained will increase with the number of input channels, leading to longer training times and, more importantly, slower inference times.

Export to Disk. Apart from sourcing the particles, each step of the solver is performed on the device, meaning that for scenes where sources are only emitting once, transfers between the host and the device should never become a bottleneck. One exception to this is when we are caching particle data to disk. This dramatically impacts the efficiency of our solver as it needs to first transfer the data to the host memory and then wait for it to be done writing the data to disk before moving on to the next frame. A slowdown of $17\times$ was observed as a worst case in some of the scenes we tested. To address this issue, we utilize a pool of threads to asynchronously write the data to disk, thus reducing latency. For each frame, a separate thread is launched with its own copy of the data to be freed when writing is completed. To prevent the host memory from depletion if the simulation is much faster than our ability to write the data, we set a maximum number of threads to be spawned simultaneously. Through profiling, we found that a pool of four threads yields optimal results on our system and is guaranteed to never run out of host memory. This multithreaded caching optimization performs in the worst case at around $4.5\times$ the usual speed of the solver without caching.

Chapter 4

Results

Our methodology was tested through six distinct demo scenes. These scenes were intentionally designed to emulate real-world production scenarios encountered in visual effects. This approach was adopted to eliminate any bias that may arise from tailoring the scenes to suit our method's strengths, thereby ensuring our approach's genuine applicability. However, note that this also entails that certain scenes may not benefit from our method, thereby representing the worst-case scenarios that our solver may encounter.

Each of the example scenes features consistent visual support. The rendered images were simulated using a timestep of Δt and a voxel size of Δx, while the corresponding plots were generated from simulations using a timestep of $2\Delta t$ and a voxel size of $2\Delta x$. This adjustment was made due to computational resource limitations, as nonadaptive simulations were needed as references.

Among the six scenes, three of them include an additional layer of secondary particles in their top (largest) image. These particles are simulated using third-party software and serve to convey scale within the scene. It is important to note that this secondary simulation layer, emitted from the cached MPM particles, solely provides high-frequency details and does not influence the primary simulation. In contrast, the thumbnail images sampled through animation only feature results from our solver.

4.1 Ship Breach

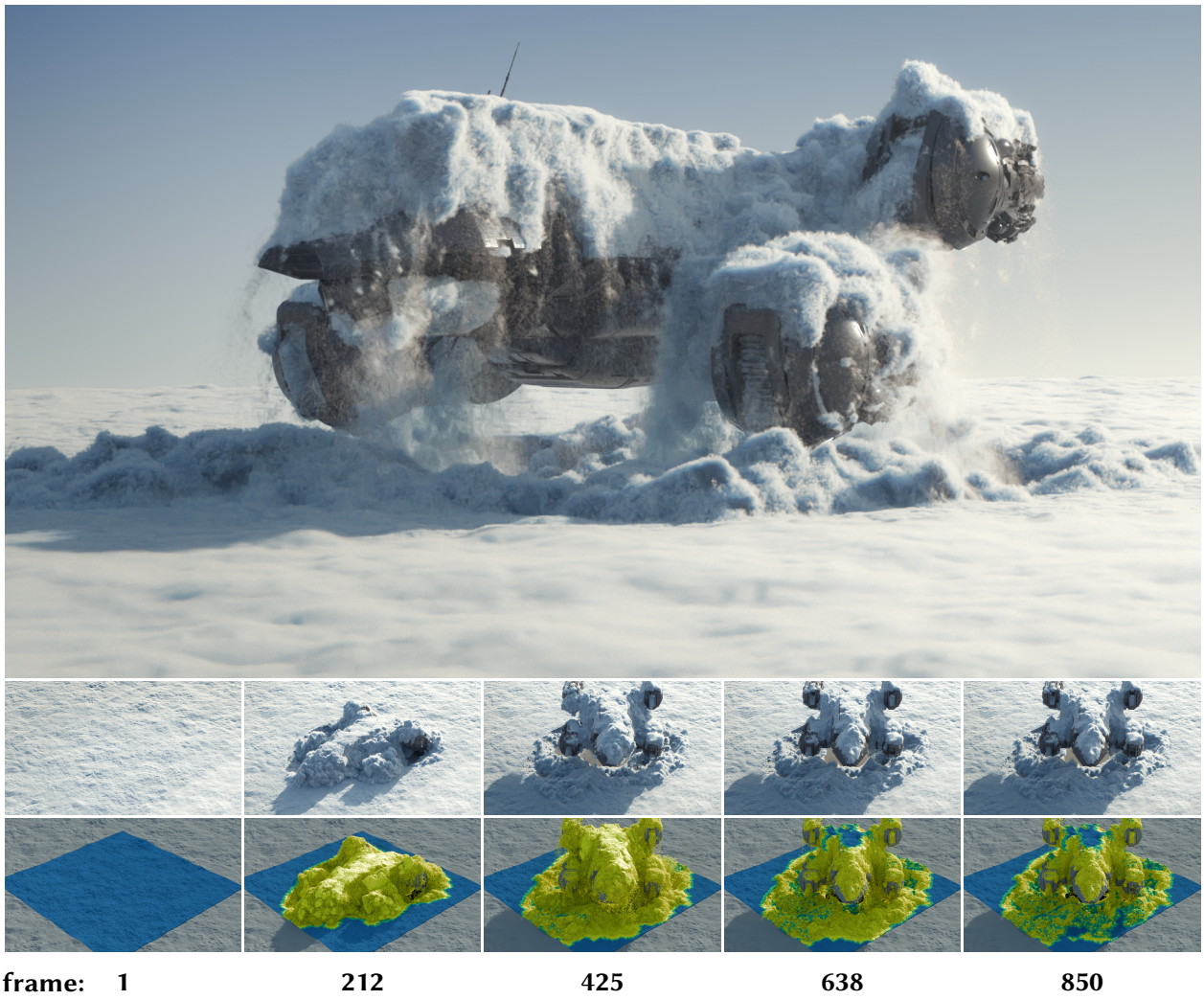


Figure 4.1. A large animated ship breaching through a thick layer of snow while rotating about the y-axis. The animation ends abruptly, triggering the collapse of large chunks of snow stacked on top of the ship. The image on top contains a secondary simulation layer of ballistic particles generated in third-party software to convey scale. The second and third rows show the primary MPM simulation only. The third row highlights the activation of the material throughout the animation, where yellow is active, green is boundary, and blue is passive.

In this simulation, a ship [For15] emerges from a thick layer of snow (Figure 4.1). Initially passive, the snow is gradually activated as the ship lifts off, exploiting sticky static friction to maintain

clumps of snow attached to its sides and underhanging.

As the ship's animation ends abruptly, a sequence of events in which the snow detaches from it is triggered. The previously passive snow on the ground is activated as falling fragments of snow collide. As the simulation progresses, the activated snow gradually returns to a passive state, with small snow clumps continually eroding from snow accumulation on top of the ship (Figure 4.2). Secondary ballistic particles are emitted from the primary MPM simulation to convey scale. The probability of secondary particle emission increases with the stretching $\frac{d}{dt}\det(\mathbf{F}_p)$, speed $||\mathbf{v}||$, and proximity to the surface defined by the material particles $\phi(\mathbf{x})$.

We see a sharp increase in passive and boundary particles around frame 400, which coincides with the abrupt end of the ship animation (Figure 4.2). Most of the particles accumulated on top of the ship are suddenly static and stable, which leads to their massive deactivation. It can be seen from the flattening of the orange curve on the right plot that this event translates into an immediate increase in performance.

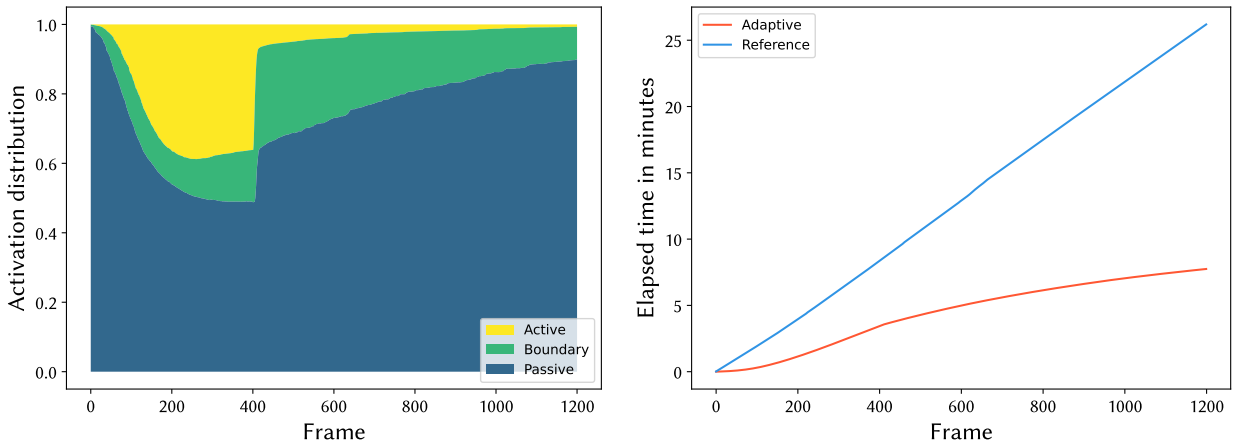


Figure 4.2. Ship breach activation and performance. (left) The activation distribution of material particles with respect to the rendered frame of the simulation. (right) The elapsed time of the adaptive and nonadaptive reference simulation with respect to the rendered frame of the simulation. Note that the plots were generated using a voxel size of $2\Delta x$ and timestep of $2\Delta t$ instead of the Δx and Δt used for the rendered images above because of computational resource limitations.

4.2 Bullet Time

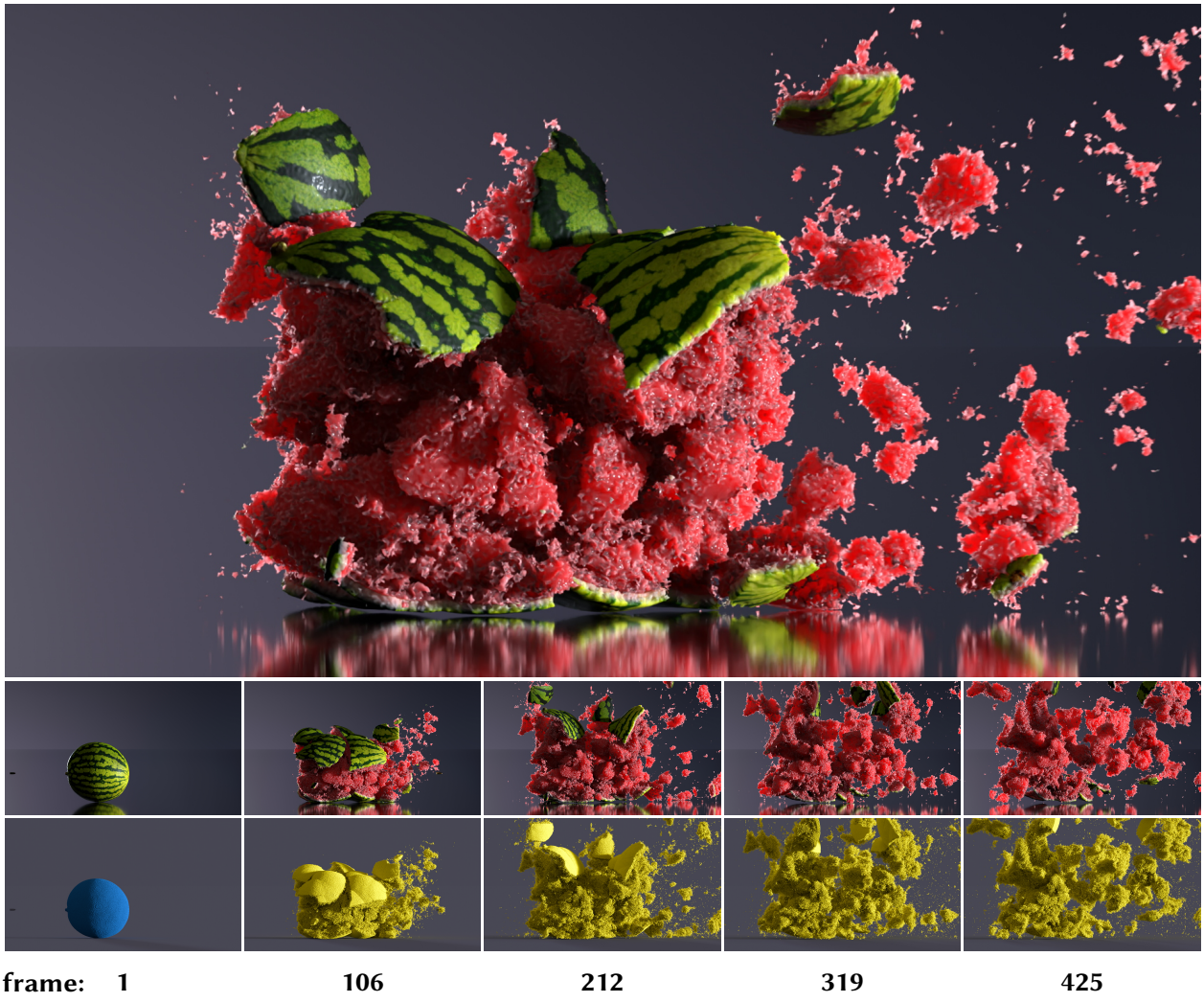


Figure 4.3. Slow-motion simulation at 24000 fps of a bullet going through a watermelon, making it explode. The bottom row highlights the activation of the material throughout the animation, where yellow is active, green is boundary, and blue is passive.

A 1000× slow-motion simulation of a bullet passing through a watermelon [Kau22] leading to its explosion (Figure 4.3). The explosion is entirely based on collision and friction; it does not rely on any custom velocity field. The watermelon comprises multiple materials, each with its own configurations, including the rind (outer layer), flesh, water overlapping the flesh, and seeds scattered inside. They all dynamically fracture as the bullet displaces them.

The material is initially passive and gradually activates as the bullet passes through it. Once the melon is fully activated, it will remain active throughout the shot as the pieces stay in motion (Figure 4.4). To achieve more realistic and dynamic effects, we duplicate the bullet ten times at the exact same location in space and allow each copy to vibrate independently near the original trajectory as the bullet traverse the watermelon.

Even though the watermelon never reaches a rest state, it is interesting to see that the head-start accumulated before impact not only amortizes the constant CNN evaluation cost but also allows the adaptive solver to slightly outperform its nonadaptive alternative (Figure 4.4). The violent impact of the bullet penetrating the watermelon propagates so quickly through the material that boundary tiles are barely visible on the left plot.

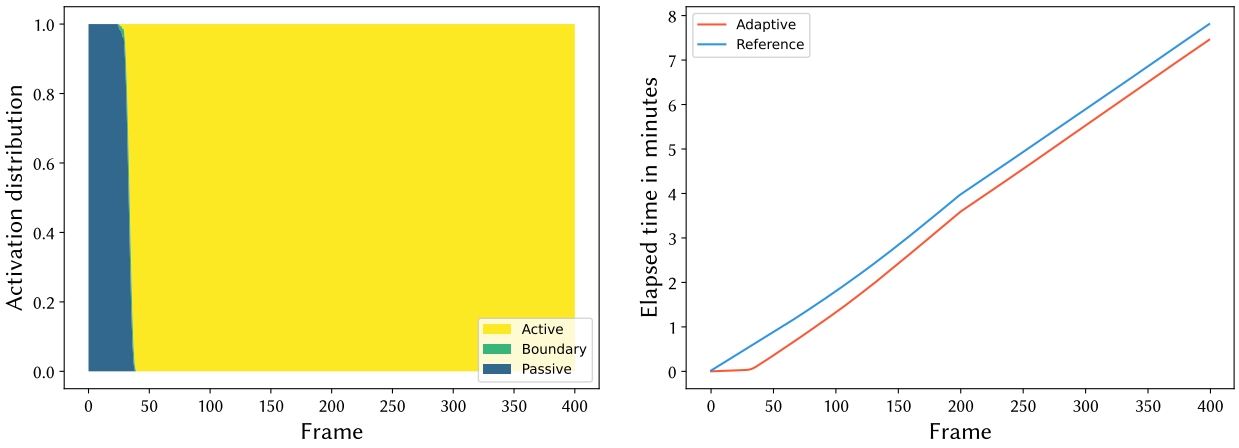


Figure 4.4. Bullet time activation and performance. (left) The activation distribution of material particles with respect to the rendered frame of the simulation. (right) The elapsed time of the adaptive and nonadaptive reference simulation with respect to the rendered frame of the simulation. Note that the plots were generated using a voxel size of $2\Delta x$ and timestep of $2\Delta t$ instead of the Δx and Δt used for the rendered images above because of computational resource limitations.

4.3 Train Push

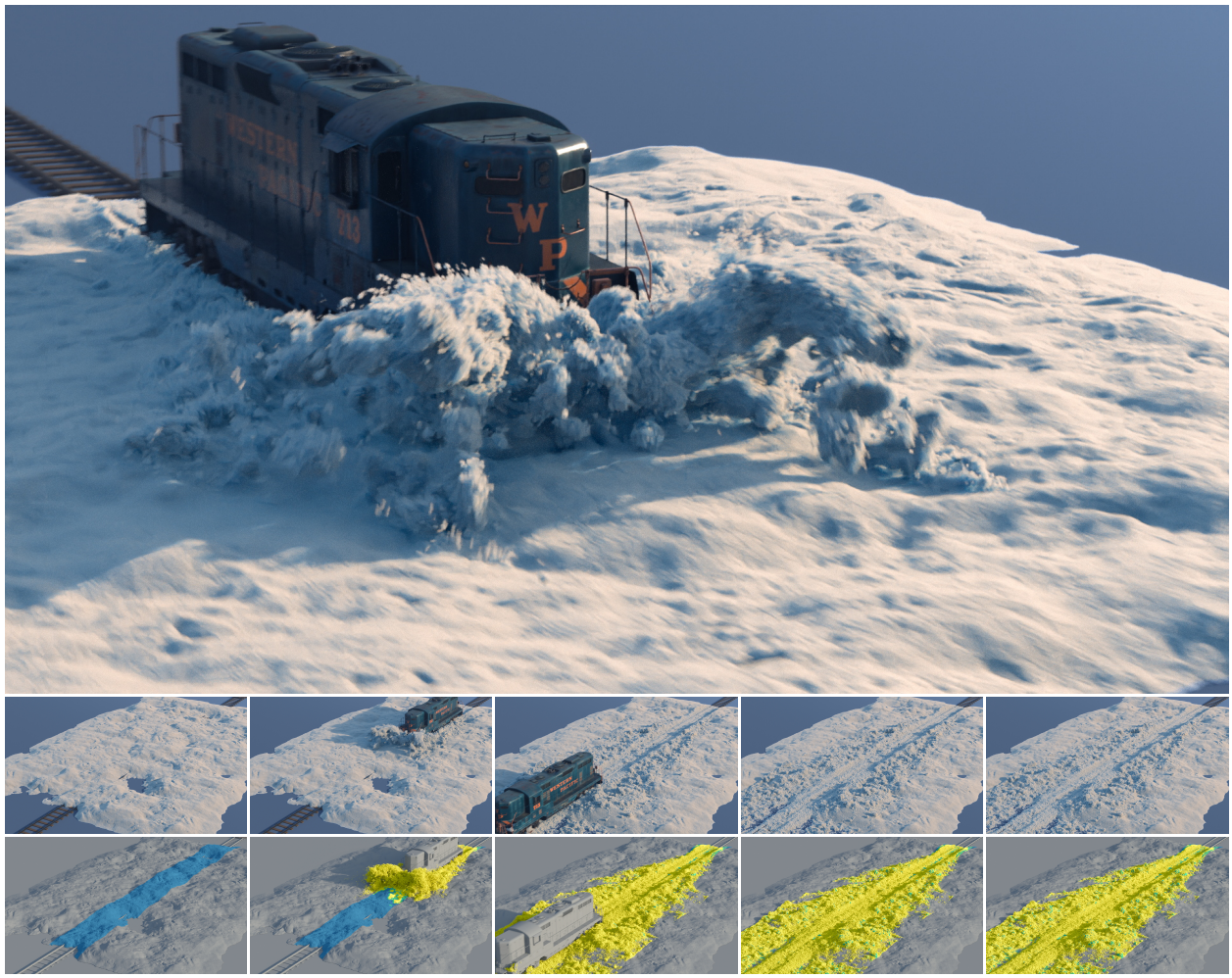


Figure 4.5. Train plowing through heavy snow over the tracks. The tumbling snow follows a cylindrical tube trajectory characteristic of snowplows. The bottom row highlights the activation of the material throughout the animation, where yellow is active, green is boundary, and blue is passive. Everything else in the scene is modeled as a static or animated SDF collider, including the snow extension on both sides of the tracks.

This simulation presents a train [Lan22] plowing through a dense and heavy layer of snow over the tracks (Figure 4.5). The particles are initially inactive and gradually activate as the train moves forward, disrupting the material's equilibrium. We employ two different stiffness con-

figurations to generate more fracture variations in the snow. After the train has passed, snow particles gradually settle and return to a passive state (Figure 4.6).

It is worth noting that this kind of phenomenon is typically accompanied by secondary simulations emitted from the primary particles, including a ballistic particle layer to model fine debris and a fluid simulation to model dust-like snow particles interacting with the surrounding air. However, the present result focuses solely on the main MPM simulation and does not include these additional layers.

The many active debris scattered across passive tiles generate many boundary tiles that prevent our system from becoming more efficient (Figure 4.6). Setting the signal multiplier α (Section 3.4) to a more aggressive value could help.

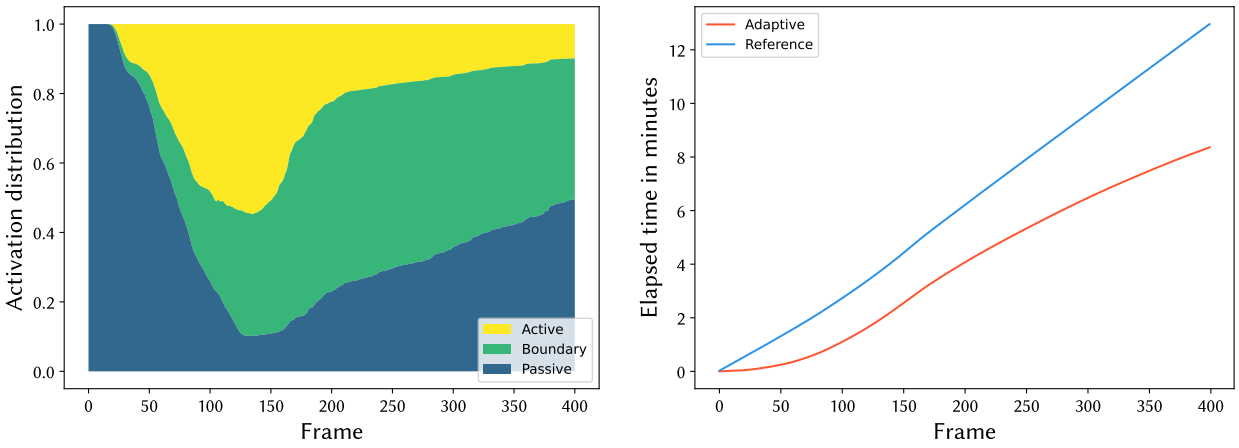


Figure 4.6. Train push activation and performance. (left) The activation distribution of material particles with respect to the rendered frame of the simulation. (right) The elapsed time of the adaptive and nonadaptive reference simulation with respect to the rendered frame of the simulation. Note that the plots were generated using a voxel size of $2\Delta x$ and timestep of $2\Delta t$ instead of the Δx and Δt used for the rendered images above because of computational resource limitations.

4.4 Avalanche

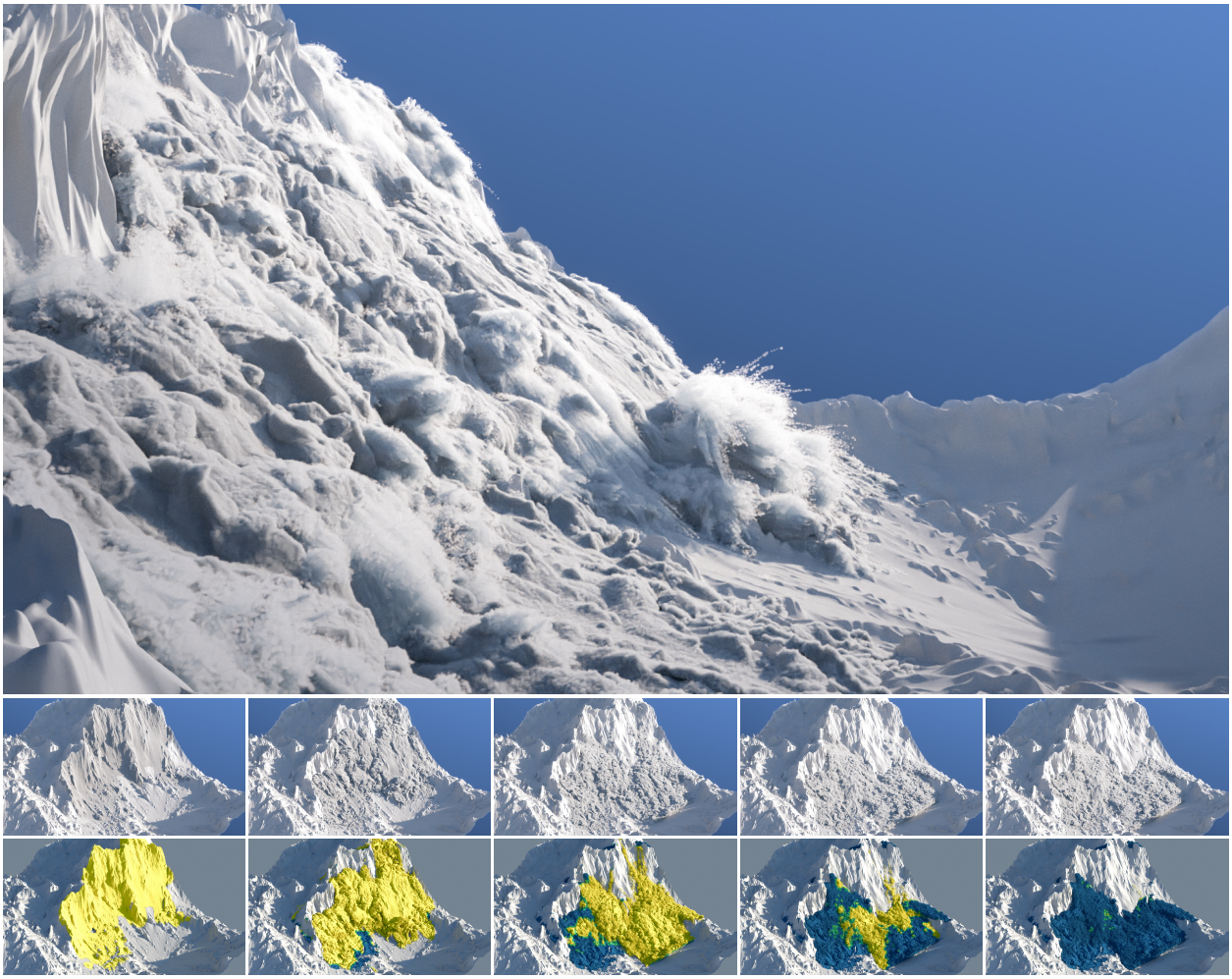


Figure 4.7. Simulation of a small-scale avalanche moving down a mountain slope. Interesting fractures occur when material collides with the uneven terrain of the slope. The top row contains a secondary simulation layer of ballistic particles generated in third-party software to convey scale. The second and third rows show the primary MPM simulation only. The third row highlights the activation of the material throughout the animation, where yellow is active, green is boundary, and blue is passive.

This next example follows the simulation of a relatively small-scale avalanche (Figure 4.7). While large-scale avalanches are typically modeled as a combination of solids and fluids, smaller-scale instances are mainly characterized by fracturing snow sliding down the slope. In such cases,

the snow does not usually travel fast enough to generate the massive cloud observed in large-scale avalanches due to air resistance. To simulate the avalanche, a chunk of the mountain slope [Bap18] is extracted and simulated as detaching snow, with varying stiffness used to encourage interesting fractures and behaviors.

The rest of the mountain is used as a static collider where the slope friction is also made irregular by distributing small sticky patches uniformly across its surface. This adds more breakup to the snow's sliding pattern and triggers folding and rolling events. The simulated particles are initially sourced with a small downward velocity, which makes them active at the beginning of the simulation. The particles gradually become passive as they collect at the bottom of the slope (Figure 4.8). Before frame 700, most of the particles are active, which makes the nonadaptive solution slightly more interesting. This drastically changes from this moment forward as particles start becoming passive.

An oriented domain that tightly fits the simulation is utilized to address the inefficiency caused by the diagonal structure of the slope. This approach enables the simulation of more particles within the same memory footprint. Furthermore, secondary ballistic particles are emitted from the primary MPM simulation to enhance the realism of the simulation. The probability of secondary emission follows the heuristic described in Section 4.1.

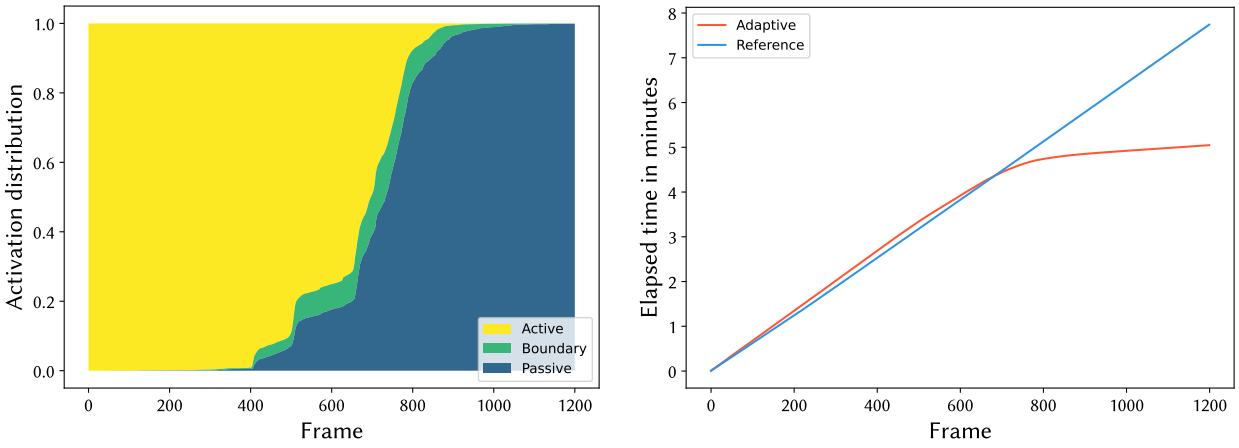


Figure 4.8. Avalanche activation and performance. (left) The activation distribution of material particles with respect to the rendered frame of the simulation. (right) The elapsed time of the adaptive and nonadaptive reference simulation with respect to the rendered frame of the simulation. Note that the plots were generated using a voxel size of $2\Delta x$ and timestep of $2\Delta t$ instead of the Δx and Δt used for the rendered images above because of computational resource limitations.

4.5 Rally Drift

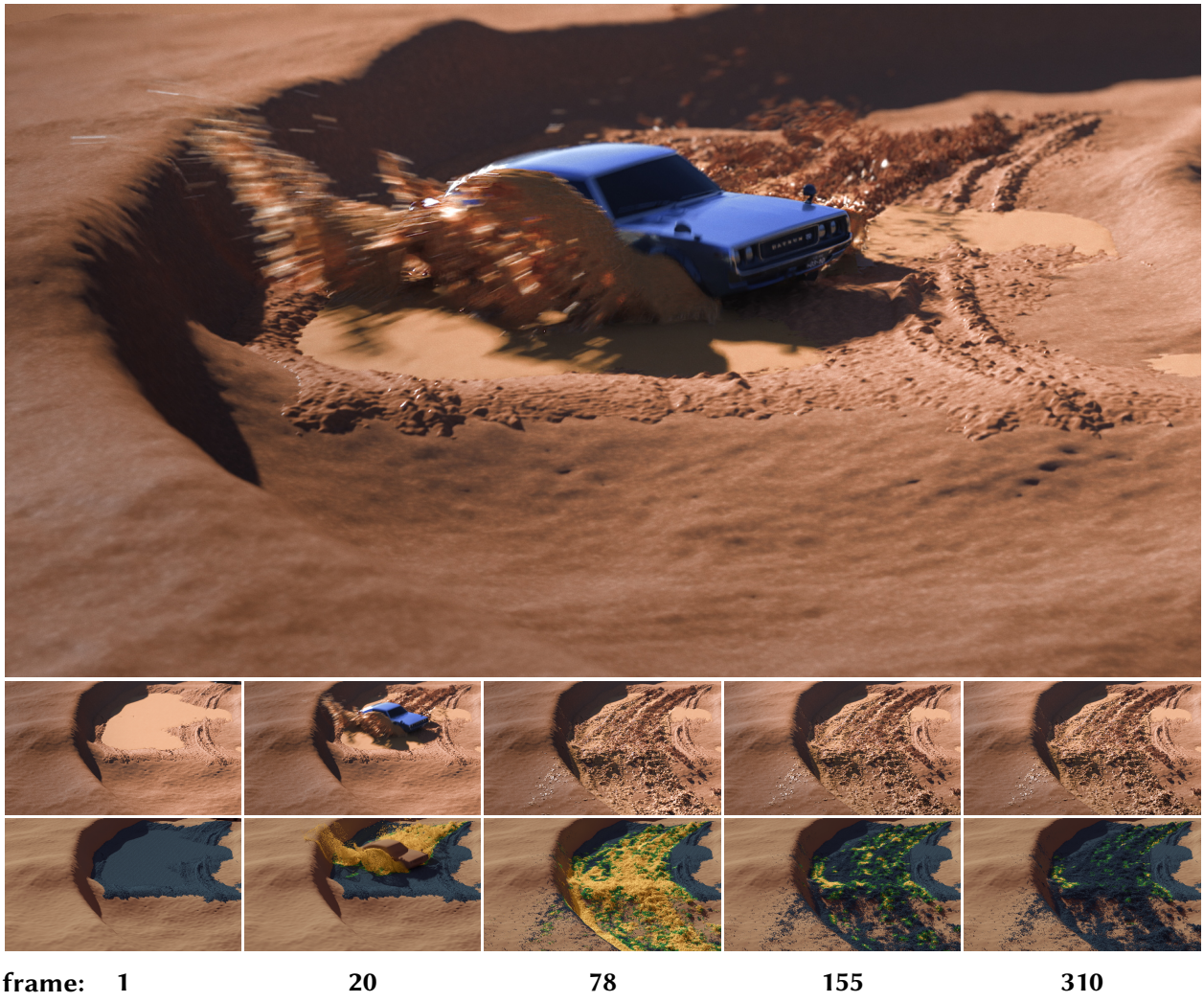


Figure 4.9. A fast-moving car drifting through a mud puddle. The animation of the material is solely based on collision and friction with the moving collider. The spinning tires are throwing mud and soil against the outer slope. The plasticity of the mud and soil allows the tires' tracks to hold throughout the shot. The bottom row highlights the activation of the material throughout the animation, where yellow is active, green is boundary, and blue is passive.

This simulation of a car [Mik17] drifting through a mud puddle is modeled using a combination of three different materials: a first layer of stiff chunky material representing compact soil pieces, a second softer layer of chunky material representing the solid that can deform more easily with

some level of plasticity, and finally, a third layer of muddy water (Figure 4.9). The particles are initially inactive and are progressively activated as the car moves forward. Once the vehicle exits the domain, the material settles and becomes passive again (Figure 4.10).

Splashes characteristic of spinning tires exclusively result from dynamic friction and collisions, thanks to the large number of substeps and the colliders' temporal interpolation. Traditionally, in the context of implicit integration with larger timesteps, this kind of effect would require custom velocity fields around the tires.

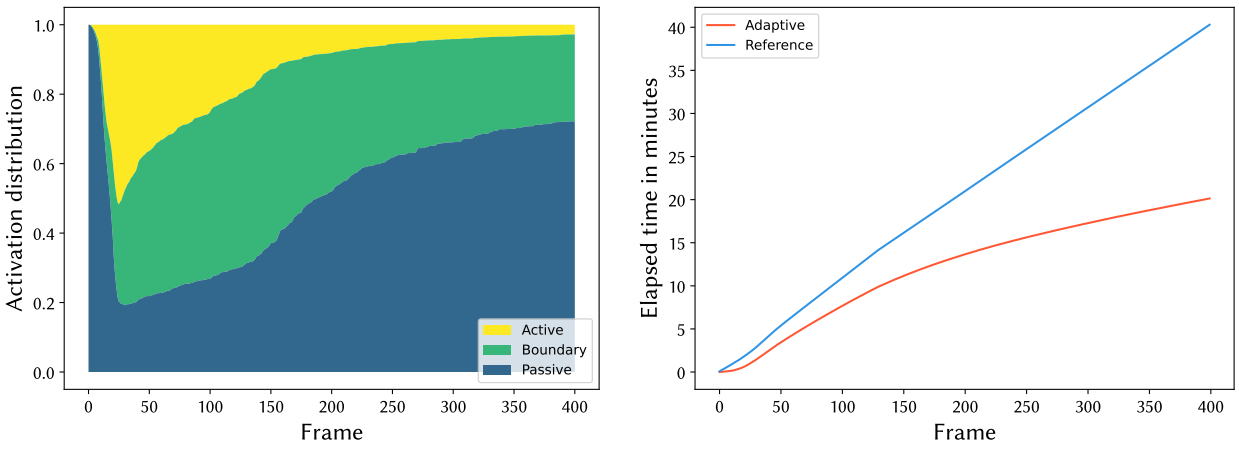


Figure 4.10. Rally drift activation and performance. (left) The activation distribution of material particles with respect to the rendered frame of the simulation. (right) The elapsed time of the adaptive and nonadaptive reference simulation with respect to the rendered frame of the simulation. Note that the plots were generated using a voxel size of $2\Delta x$ and timestep of $2\Delta t$ instead of the Δx and Δt used for the rendered images above because of computational resource limitations.

4.6 Glacier Collapse

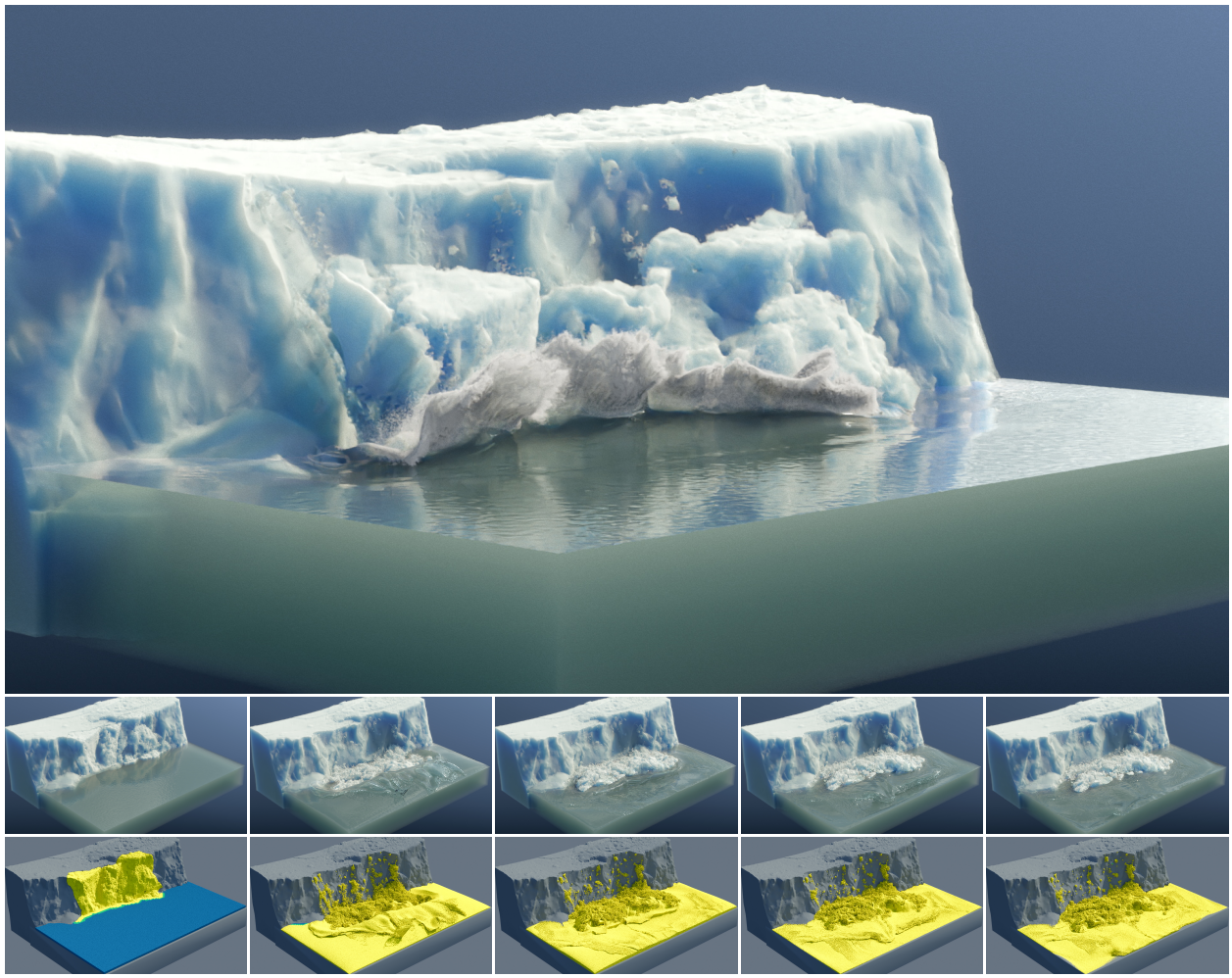


Figure 4.11. A glacier fragment detaches and collapses into the water. The image at the top contains a secondary simulation layer of whitewater particles generated in third-party software to convey scale. The second and third rows show the primary MPM simulation only. The bottom row highlights the activation of the material throughout the animation, where yellow is active, green is boundary, and blue is passive.

A glacier's section [Mat21] detaches and collapses into the ocean. The ice material is modeled with varying stiffness levels to produce interesting fractures upon impact with the water (Figure 4.11). Density is modeled accurately, causing the ice material to float on top of the water due

to buoyancy forces.

In order to enhance the realism of the scene, a secondary foam layer is generated in third-party software, which conveys scale and adds a more convincing touch to the water. To optimize memory usage, a collider partially fills the bottom of the water volume, allowing water particles to be densely packed where the glacier's fragment interacts with the water.

The water starts inactive but, once activated, it will remain active throughout the simulation to preserve its subtle movements (Figure 4.12). This scene represents the worst-case scenario for our adaptive solver as all particles quickly become active and are never made passive again. It is therefore reassuring to see that its performance is comparable to the nonadaptive alternative.

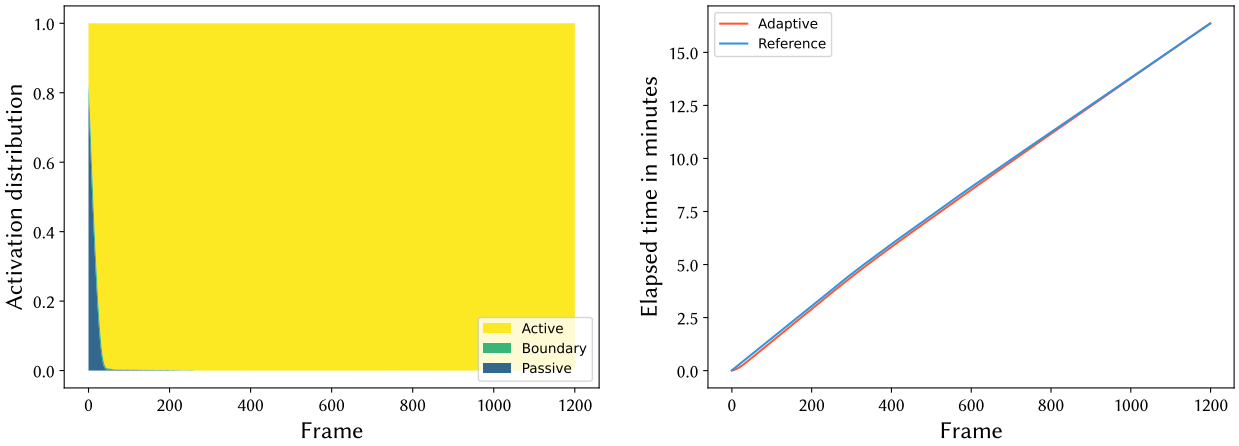


Figure 4.12. Glacier collapse activation and performance. (left) The activation distribution of material particles with respect to the rendered frame of the simulation. (right) The elapsed time of the adaptive and nonadaptive reference simulation with respect to the rendered frame of the simulation. Note that the plots were generated using a voxel size of $2\Delta x$ and timestep of $2\Delta t$ instead of the Δx and Δt used for the rendered images above because of computational resource limitations.

A summary of the scenes' configuration and performance can be found in Table 4.1. *Ship Breach* exhibited the most significant speedup, achieving an acceleration of $3.4\times$. This performance can be attributed to the predominance of static snow within the ship's vicinity. It is worth highlighting that this particular scene reached the highest particle count, which serves as a testament to the scalability of our method. To accommodate the rapid movement of colliders in *Train Push* and *Rally Drift*, the evaluation frequency of the CNN was augmented to $5\times$ per rendered

frame. Using an oriented domain in the *Avalanche* played a pivotal role in mitigating the wasteful memory allocation for empty voxels which led to an remarkable 7.5 seconds per frame on average. In the cases of *Bullet Time* and *Glacier Collapse*, a relatively high signal multiplier α was used to ensure the preservation of subtle motion. Nevertheless, compared to the nonadaptive alternative, no discernible performance degradation was observed in either of these scenes.

Example (Figure)	**Particles** $\times 10^6$	**Grid**	$\boldsymbol{\Delta t}$ $\times 10^{-4}$	$\boldsymbol{\alpha}$	**CNN eval/frame**	**sec/frame**	**Adaptive speedup**
Ship Breach (4.1)	54.0	$492 \times 212 \times 492$	6.9	0.5	1	16.6	$3.38\times$
Bullet Time (4.2)	24.1	$688 \times 364 \times 492$	4.2	0.99	1	17.9	$1.05\times$
Train Push (4.3)	24.9	$520 \times 144 \times 1748$	2.6	0.5	5	28.8	$1.55\times$
Avalanche (4.4)	35.5	$736 \times 148 \times 760$	21.0	0.5	1	7.5	$1.53\times$
Rally Drift (4.5)	14.9	$1400 \times 196 \times 536$	1.1	0.5	5	40.7	$2.00\times$
Glacier Collapse (4.6)	39.5	$320 \times 184 \times 520$	6.9	0.999	1	16.4	$1.00\times$

Table 4.1. Summary of the configuration and performance of each example scene.

Chapter 5

Discussion

In this chapter, we provide an analysis of our approach based on neural temporal adaptivity, building upon the results showcased in Chapter 4. Our method exhibits attractive performance gains when significant portions of the domain are static. It generalizes well to varying scene scales, translating into predictable generalizations without requiring user interventions.

Despite these advantages, certain inaccuracies may surface depending on scene materials and configurations. However, such inaccuracies can be effectively mitigated by tuning specific user-defined parameters. Furthermore, we have found that our method can accelerate numerous simulations while concealing sagging effects by overlapping them with actual displacements required in the animation.

5.1 Method Analysis

Our approach delivers a noteworthy enhancement of computational efficiency in situations where significant portions of the domain remain static. Chaotic scenes, such as *Bullet Time* (Section 4.2) and *Glacier Collapse* (Section 4.6), are not ideal case studies for evaluating performance improvements of our method. In such scenarios, all objects remain active throughout the animation after initial activation. Moreover, the CNN must still be queried, leading to additional computational overhead and potential subsequent deterioration of the overall solver performance. Nonetheless, it is worth mentioning that such occurrences represent a minority in real-world applications.

In the worst-case scenario, the CNN is continually evaluated without ever deactivating a single tile passed an early point in the animation. This is encountered in *Glacier Collapse* (Section 4.6). In this case, this led to no performance degradation (Figure 4.12) thanks to the computational efficiency of our CNN compared to the rest of the solver.

Scenes that resemble *Ship Breach* (Section 4.1), *Train Push* (Section 4.3), *Avalanche* (Section 4.4), and *Rally Drift* (Section 4.5) are significantly more prevalent in practice; a substantial volume of material remains stationary, a collider activates it, and it subsequently returns to its resting state. Our method exhibits performance gains in these scenarios, as it expands computational resources exclusively on simulating the event of interest and not on maintaining the material at equilibrium. In most cases, the cost incurred by continuously evaluating the CNN is trivial compared to the cost of assessing the solver on static tiles (Figure 4.12).

5.1.1. Generalization. The presented method generalizes well to domains of varying dimensions, thanks to the scene scale Ω multiplying the velocity consumed by the CNN. This enables scenes such as *Bullet Time* (Section 4.2) and *Avalanche* (Section 4.4) to utilize the same temporal adaptivity configuration. That being said, the propagation of small ripples can pose a challenge, as these oscillations are prone to be made passive by the CNN. Nevertheless, as explained in Section 5.2, some user-defined parameters can facilitate the accurate simulation of such phenomena.

Our system has been trained to predict notable motion by leveraging local speed distribution in 3D space. This implies that the user is relieved from specifying a speed threshold, search distance, or distance-based attenuation function. All of these parameters, crucial to traditional methods, are accounted for by the learning algorithm, which captures the appropriate behavior based on our generated data.

Furthermore, the embedded CNN could potentially be fine-tuned on particular edge-case scenes to achieve the optimal balance between performance and accuracy without user-driven trial and error. In the visual effects industry, executing numerous iterations of the same simulation is standard practice. Therefore, it could be logical to retrain the model to improve its reliability for subsequent iterations. Nevertheless, our current generalized training methodology yields satisfactory performance on average, making per-scene retraining unnecessary in most

cases.

5.1.2. Physical Accuracy. Depending on the scene components and configuration, this temporal adaptivity scheme can increase or decrease simulation accuracy. A mainly static scene with slow-moving colliders will benefit from passive tiles by preventing the material from compressing on itself when the timestep used is slightly too large. However, for simulations with fast-moving objects, inactive tiles may not activate quickly enough, resulting in momentum loss and unwanted fractures of stiff structures due to partial activation.

In certain scenes featuring slow erosion, such as *Ship Breach* (Section 4.1), staircase artifacts can be encountered at the edges of erosion regions. These artifacts may arise when small debris detaches and falls at an insufficient speed to activate the adjacent tiles that should also erode. Although such inaccuracies are infrequent and barely noticeable in practice, they can be eliminated by adjusting the signal multiplier α or the CNN evaluation frequency, as discussed in Section 5.2.

5.1.3. Practical Improvements. By only simulating the particles in motion, we can obtain high-resolution simulation previews much more quickly while incurring only a fraction of the cost. This acceleration of the simulation process makes the user experience more interactive, leading to superior results by enabling rapid iterations to achieve the desired outcome.

Our simulations are always sag-free, implying that the material remains perfectly still until it needs to move. Nevertheless, this approach may be deemed inaccurate as the initial state of the material may not represent its actual equilibrium. In visual effects, practitioners tend to use a technique called "preroll" to allow the material to sag and reach its actual rest state before the shot begins. Unfortunately, this technique can become prohibitively expensive for high-resolution simulations, and therefore, it remains an active area of research, as referenced by Hsu et al. [HTYW22].

Our proposed method can conceal sagging by overlapping it with the actual displacement required in the shot. This is seamless in cases where the material has been modeled with the appropriate stiffness. However, if the material's stiffness or particle packing is too low, visually disturbing sinking effects may occur on activation.

Depending on the material and timestep utilized, it is not uncommon to witness material

particles slowly drifting to rest after experiencing a significant perturbation. To tackle this, our adaptive method deactivates the particles once they become nearly static. This is particularly useful for materials exhibiting plasticity, but it should be avoided for vibrating elastic solids or small liquid ripples.

5.2 Limitations and Future Work

As previously stated, our proposed method is not flawless, and user-defined parameters significantly influence the reliability of the output. As is the case with all freezing temporal adaptivity schemes, the reactivation mechanism is the most critical aspect of the system. While our approach can result in impressive performance improvements, visual artifacts may appear in the simulation if the adaptive parameters are not appropriately selected for a given scene.

5.2.1. Scattered Active Tiles. Following a substantial perturbation, small material fragments can fly away from an impact point and land with high velocity on passive material. These pieces of debris will usually stay active for a long time, slowly drifting to their rest configuration. This has the unfortunate consequence of turning all the neighboring tiles into boundary tiles (26 in the worst case). Although not as computationally expensive as active tiles, boundary tiles still need to be scattered to the grid along with active particles. Scenes such as *Ship Breach* (Figure 4.2), *Train Push* (Figure 4.6), and *Rally Drift* (Figure 4.10) are all remarkable instances of this issue which significantly impacts our system's ability to reduce computation. Some future work might involve detecting the finer pieces of debris so there are treated differently.

5.2.2. Amortized Evaluation Artifacts. As mentioned earlier, querying the CNN in lockstep with the solver can be inefficient in terms of performance. Hence, a general rule of thumb is to set the evaluation frequency to once per rendered frame, which is typically a suitable default. However, this optimization, combined with fast-moving colliders, may cause a delay in tile activation and prevent them from reacting to certain forces, resulting in kinetic energy loss and even instability. Consequently, users must carefully consider the type of motion involved in each scene and adjust the CNN evaluation frequency accordingly.

While not currently implemented, it is possible to derive a lower bound of the CNN evaluation frequency from a CFL condition, which would automate this process for the most part. However, in some cases, this approach may result in excessively conservative bounds, where the simulation would not visually benefit from the increased computational cost.

5.2.3. Limited Receptive Field. Delayed tile activation can result in significant deviations from the ground truth, especially when simulating large, stiff objects. If only partially activated, material may break apart in response to a collider. This problem can arise due to the limited receptive field of the CNN, which may prevent the solver from properly activating a material fragment as a whole in a single iteration. The ripple activation following a substantial perturbation may also cause delayed reactions in the material, leading to inaccurate results.

Moreover, the spatial dimension of the CNN's receptive field is inversely proportional to the simulation resolution. As a result, late activation artifacts are more likely to occur in high-resolution domains, exacerbating the issue. Combined with amortized evaluation, this problem can result in highly inaccurate simulations.

5.2.4. Critical Subtle Motion. In the context of liquid simulation, low-amplitude effects such as subtle ripple motion can have a visually significant impact on the resulting simulation. To accurately capture these effects, it is often necessary to increase the signal multiplier α to a value in the neighborhood of 0.98. However, this approach may prevent other materials from staying or becoming passive. One possible solution to improve the robustness of our method is to provide the CNN with material-specific information, which could make the default of $\alpha = 0.5$ more reliable across the space of all materials.

5.2.5. Manual Adjustments. The signal multiplier α is a parameter that warrants careful consideration and can be adjusted on a per-scene basis. Generally, a default value of 0.5 suffices for most scenarios, but it becomes imperative to increase it when low-amplitude motion must be preserved. Conversely, a reduced α can expedite the simulation and promote rapid approximative outcomes. To mitigate visual artifacts arising from excessive tile freezing, one could augment this parameter before a final high-resolution simulation. Despite offering much-needed flexibility,

this constitutes yet another parameter that the user must judiciously calibrate.

As previously stated, the CNN evaluation frequency also constitutes an adjustable parameter to alter the default behavior of the temporal adaptivity mechanism. However, the latter should only be modified in the presence of fast-moving colliders and does not affect the system's sensibility.

5.2.6. Regional Time Stepping. During the initial phases of this project, we implemented the temporally adaptive method proposed by Fang et al. [FHHJ18]. Similar to what we did in this work, we implemented their method on the GPU and incorporated a CNN into the solver to predict per-tile timesteps instead of utilizing the analytical lower bound on Δt presented by the authors. While our preliminary results were encouraging, we abandoned this approach after realizing that even a minor error in the CNN predictions could trigger instability in specific tiles, leading to a catastrophic failure of the entire simulation domain.

To mitigate this issue, we had to introduce very conservative padding to the predicted Δt, effectively nullifying any gains derived from the adaptivity. In most instances, this counterproductive measure led to performance degradation compared to the nonadaptive approach. Our early experimentation using neural networks as part of our simulation workflow taught us that it is ill-advised to employ such techniques in contexts unable to recover from infrequent errors. Future studies could attempt to identify such failure cases, rewind the simulation, locally increase Δt, and then attempt to proceed again. However, such a strategy would complicate the algorithm without guaranteeing improved performance.

Furthermore, the regional time stepping method we explored required doubling the memory footprint of our simulation due to buffer blocks needed to provide accurate boundary conditions to blocks running at finer timesteps. Besides the added computational overhead in managing these duplicates, this earlier approach severely constrained our solver by halving its maximum resolution when adaptivity is enabled. In the GPU context, where memory is a scarce resource, this presents a significant limitation to the practicality of this technique for real-world applications.

5.2.7. Scalability and Distributed Computing. Although our method was initially designed to operate on a single GPU, there could be benefits to enabling the application to run on distributed systems. Nevertheless, we must consider the intricacies of Multi-GPU computation and distributed systems. Communication between nodes can often become a bottleneck in these systems, particularly for physics simulations where a significant amount of data must be continually exchanged between nodes to maintain particle state integrity. Load balancing could also become an issue in certain scenes, where a significant portion of the particles are located in one partition while other nodes mostly wait for synchronization, resulting in decreased efficiency.

Note that our method's performance would not increase linearly with the addition of compute nodes. Improvements in load balancing could be achieved with dynamic partitioning, but this would result in additional inter-node communication costs, hence the sub-linear performance growth. However, distributed computation is well-suited to programming models like *CUDA*, and only minor modifications to the codebase should be required.

It may be advantageous to port the application to a hardware-agnostic programming language such as OpenCL, enabling it to also run on CPUs and potentially benefit distributed scenarios. However, this would likely result in a performance drop due to the algorithm being designed for massively parallel architectures. In addition, the lower memory bandwidth of CPUs could exacerbate bottlenecks caused by inter-node communication. Furthermore, optimizations utilizing the hardware-specific API, such as shared memory, texture memory, and constant memory, would need to be replaced by less efficient alternatives. Many data structures would most likely also require an overhaul. Depending on the needs and available hardware, the benefits of this port may be worth considering.

Chapter 6

Conclusion

Throughout this thesis, we have explored a machine learning-based approach to achieve temporal adaptivity within the context of the material point method (MPM). Our novel procedure to construct the training set by evaluating the ground-truth future state of the material appears to be effective. By leveraging tile partitioning, accurate boundary conditions can be maintained in the presence of passive material.

Our convolutional neural network (CNN) is able to fit the training data, indicating sufficient capacity to approximate the desired function while very high accuracy is obtained on the validation set by the end of training. By formulating the problem as an hybrid method that combines the strength of an established physics framework with the reasoning and generalization capabilities of a neural network, we were able to outperform our nonadaptive approach across a wide variety of test scenes.

The customized integration of a CNN within the solver presents an advantage from a performance and dependency management standpoint. The asynchronous evaluation of the CNN and physics solver results in a noteworthy performance boost. The addition of the signal multiplier parameter α gives the user enhanced control and flexibility in fine-tuning the neural network's behavior post-training. Moreover, the implementation on the GPU expedites execution while retaining a manageable codebase complexity. By integrating industry-standard libraries into our prototype, we were able to construct intricate demo scenes akin to those utilized in practical settings rather than simplified toy examples typically used in academic studies.

Our methodology has been validated through several examples where we have observed significant performance improvements in partially static scenes. Furthermore, our method has demonstrated remarkable robustness due to the efficient implementation of the CNN, ensuring that even worst-case scenarios do not lead to performance degradation.

However, the technique has some limitations, such as the need for manual adjustments and potential artifacts arising from static tiles. These limitations present opportunities for further refinement of the method in future work.

Combining machine learning with traditional computational physics holds exciting promise for computer graphics and physically-based animation. Hybrid approaches, such as this one, can provide greater flexibility and control compared to end-to-end training. Moreover, they do not require an absurd amount of computing power to converge even in complex scenarios featuring considerable degrees of freedom, such as physics simulations.

Bibliography

[Bap18] Bapichu. Mountain peak. Sketchfab [3D model], 2018.

[BET+10] Jan Bender, Kenny Erleben, Matthias Teschner, Markus Ihmsen, Nadir Akinci, and Marc Gissler. Boundary handling and adaptive time-stepping for PCISPH. In *Workshop on virtual reality interaction and physical simulation VRIPHYS*, volume 6, 2010.

[BET14] Jan Bender, Kenny Erleben, and Jeff Trinkle. Interactive simulation of rigid body dynamics in computer graphics. In *Computer Graphics Forum*, volume 33, pages 246–270. Wiley Online Library, 2014.

[BKR88] Jeremiah U Brackbill, Douglas B Kothe, and Hans M Ruppel. FLIP: a low-dissipation, particle-in-cell method for fluid flow. *Computer Physics Communications*, 48(1):25–38, 1988.

[BMF03] Robert Bridson, Sebastian Marino, and Ronald Fedkiw. Simulation of clothing with folds and wrinkles. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, page 28–36, Goslar, DEU, 2003. Eurographics Association.

[Bri15] Robert Bridson. *Fluid simulation for computer graphics*. CRC Press, 2015.

[CBK20] Eulalie Coevoet, Otman Benchekroun, and Paul G Kry. Adaptive merging for rigid body simulation. *ACM Transactions on Graphics (TOG)*, 39(4):35–1, 2020.

[CFL67] Richard Courant, Kurt Friedrichs, and Hans Lewy. On the partial difference equations of mathematical physics. *IBM Journal of Research and Development*, 11(2):215–234, 1967.

[Cou23] Erwin Coumans. Bullet physics library, 2003–2023. Version 3.x.x.

[DG96] Mathieu Desbrun and Marie-Paule Gascuel. Smoothed particles: A new paradigm for animating highly deformable bodies. In *Computer Animation and Simulation'96:*

Proceedings of the Eurographics Workshop in Poitiers, France, August 31–September 1, 1996, pages 61–76. Springer, 1996.

[FHHJ18] Yu Fang, Yuanming Hu, Shi-Min Hu, and Chenfanfu Jiang. A temporally adaptive material point method with regional time stepping. In *Computer Graphics Forum*, volume 37, pages 195–204. Wiley Online Library, 2018.

[For15] Simon Formanski. Prometheus. Sketchfab [3D model], 2015.

[FSH11] Basil Fierz, Jonas Spillmann, and Matthias Harders. Element-wise mixed implicit-explicit integration for stable dynamic simulation of deformable objects. In *Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 257–266, 2011.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[GPB+11] Prashant Goswami, Renato Pajarola, Jan Bender, Kenny Erleben, and Eric Galin. Time adaptive approximate SPH. Eurographics, 2011.

[GWW+18] Ming Gao, Xinlei Wang, Kui Wu, Andre Pradhana, Eftychios Sifakis, Cem Yuksel, and Chenfanfu Jiang. GPU optimization of material point methods. *ACM Transactions on Graphics (TOG)*, 37(6):1–12, 2018.

[HFG+18] Yuanming Hu, Yu Fang, Ziheng Ge, Ziyin Qu, Yixin Zhu, Andre Pradhana, and Chenfanfu Jiang. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Transactions on Graphics (TOG)*, 37(4):1–14, 2018.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[HTYW22] Jerry Hsu, Nghia Truong, Cem Yuksel, and Kui Wu. A general two-stage initialization for sag-free deformable simulations. *ACM Transactions on Graphics (TOG)*, 41(4):1–13, 2022.

[JBS06] Mark W Jones, J Andreas Baerentzen, and Milos Sramek. 3D distance fields: A survey of techniques and applications. *IEEE Transactions on visualization and Computer Graphics*, 12(4):581–599, 2006.

[JSS+15] Chenfanfu Jiang, Craig Schroeder, Andrew Selle, Joseph Teran, and Alexey Stomakhin. The affine particle-in-cell method. *ACM Transactions on Graphics (TOG)*, 34(4):1–10, 2015.

[JST+16] Chenfanfu Jiang, Craig Schroeder, Joseph Teran, Alexey Stomakhin, and Andrew Selle. The material point method for simulating continuum materials. In *ACM SIGGRAPH 2016 Courses*, pages 1–52. 2016.

[Kau22] Lassi Kaukonen. Watermelon. Sketchfab [3D model], 2022.

[KSH17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[Lan22] Jesper Landin. Emd gp7 western pacific 713. Sketchfab [3D model], 2022.

[Lar14] Hugo Larochelle. Neural networks; training neural networks - back-propagation algorithm. Université de Sherbrooke ift725 Lecture Slides Back-propagation, 2014. https://info.usherbrooke.ca/hlarochelle/ift725/2_07_backpropagation.pdf.

[LC87] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3D surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21(4):163–169, 1987.

[LCJB+89] Yann Le Cun, Lawrence D Jackel, Brian Boser, John S Denker, Hans Peter Graf, Isabelle Guyon, Don Henderson, Richard E Howard, and William Hubbard. Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11):41–46, 1989.

[LMOW04] Adrian Lew, Jerrold E. Marsden, Michael Ortiz, and Matthew West. Variational time integrators. *International Journal for Numerical Methods in Engineering*, 60(1):153–212, 2004.

[Mat21] Matousekfoto. White cliffs of dover. Sketchfab [3D model], 2021.

[MHHR07] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position based dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118, 2007.

[Mik17] Karol Miklas. 1972 datsun 240k gt. Sketchfab [3D model], 2017.

[MLJ+13] Ken Museth, Jeff Lait, John Johanson, Jeff Budsberg, Ron Henderson, Mihai Alden, Peter Cucka, David Hill, and Andrew Pearce. OpenVDB: An open-source data structure and toolkit for high-resolution volumes. In *ACM SIGGRAPH 2013 Courses*. 2013.

[MMCK14] Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Unified particle physics for real-time applications. *ACM Transactions on Graphics (TOG)*, 33(4):1–12, 2014.

[Mon92] Joe J Monaghan. Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30(1):543–574, 1992.

[MS01] Victor Milenkovic and Harald Schmidl. Optimization-based animation. *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2001*, pages 37–46, 06 2001.

[MST+11] Aleka McAdams, Andrew Selle, Rasmus Tamstorf, Joseph Teran, and Eftychios Sifakis. Computing the singular value decomposition of 3x3 matrices with minimal branching and elementary floating point operations. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2011.

[MSW+09] Aleka McAdams, Andrew Selle, Kelly Ward, Eftychios Sifakis, and Joseph Teran. Detail preserving continuum simulation of straight hair. *ACM Transactions on Graphics (TOG)*, 28(3):1–6, 2009.

[Mus21] Ken Museth. NanoVDB: A gpu-friendly and portable VDB data structure for real-time rendering and simulation. In *ACM SIGGRAPH 2021 Talks*, pages 1–2. 2021.

[MWN+17] Pierre-Luc Manteaux, Christopher Wojtan, Rahul Narain, Stéphane Redon, François Faure, and Marie-Paule Cani. Adaptive physically based models in computer graphics. In *Computer Graphics Forum*, volume 36, pages 312–337. Wiley Online Library, 2017.

[NH10] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala.

Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32, 2019.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[SABS14] Rajsekhar Setaluri, Mridul Aanjaneya, Sean Bauer, and Eftychios Sifakis. SPGrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Transactions on Graphics (TOG)*, 33(6):1–12, 2014.

[Set96] James A Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

[SGGP+20] Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning*, pages 8459–8468. PMLR, 2020.

[SK10] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[SKB08] Michael Steffen, Robert M Kirby, and Martin Berzins. Analysis and reduction of quadrature errors in the material point method (MPM). *International Journal for Numerical Methods in Engineering*, 76(6):922–948, 2008.

[SLF08] Andrew Selle, Michael Lentine, and Ronald Fedkiw. A mass spring model for hair simulation. *ACM Transactions on Graphics (TOG)*, 27(3):1–11, aug 2008.

[SSC+13] Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. A material point method for snow simulation. *ACM Transactions on Graphics (TOG)*, 32(4):1–10, 2013.

[SZS95] Deborah Sulsky, Shi-Jian Zhou, and Howard L Schreyer. Application of a particle-in-cell method to solid mechanics. *Computer Physics Communications*, 87(1-2):236–252, 1995.

[TGK+17] Andre Pradhana Tampubolon, Theodore Gast, Gergely Klár, Chuyuan Fu, Joseph Teran, Chenfanfu Jiang, and Ken Museth. Multi-species simulation of porous sand and water mixtures. *ACM Transactions on Graphics (TOG)*, 36(4):1–11, 2017.

[Tsi95] John N Tsitsiklis. Efficient algorithms for globally optimal trajectories. *IEEE Transactions on Automatic Control*, 40(9):1528–1538, 1995.

[Wal] Walt Disney Animation Studios. Partio: A library for particle io and manipulation. https://github.com/wdas/partio. Accessed: March 14, 2023.

[ZB05] Yongning Zhu and Robert Bridson. Animating sand as a fluid. *ACM Transactions on Graphics (TOG)*, 24(3):965–972, 2005.

[Zha05] Hongkai Zhao. A fast sweeping method for eikonal equations. *Mathematics of Computation*, 74(250):603–627, 2005.