UNIVERSITY OF CALIFORNIA

Los Angeles

Scalable and Efficient Material Point Methods on Modern Computational Platforms

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Yuxing Qiu

2024

ABSTRACT OF THE DISSERTATION

Scalable and Efficient Material Point Methods on Modern Computational Platforms

by

Yuxing Qiu
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2024
Professor Demetri Terzopoulos, Chair

The challenge of efficiently and plausibly simulating deformable solids and fluids remains a significant challenge in the domains of Computer Graphics and Scientific Computing. This dissertation presents an in-depth exploration of physics-based simulation, with an emphasis on the Material Point Method (MPM) — a dominant technique in this arena. Our research aims to extend the capabilities of MPM, focusing on enhancing its performance, scalability, range of applications, and integration with emerging AI technologies. We first summarize our development of optimized MPM leveraging GPU architectures. This advancement accelerates scenarios involving hundreds of millions of particles in multi-GPU computational environments. Furthermore, the thesis introduces a device-agnostic and distributed MPM framework. This system is adept at dynamically allocating workloads across multiple computing ranks, thus enabling simulations at unprecedented particle-count scales. Additionally, the dissertation examines the application of physics-based simulation, specifically MPM, in real-time contexts. It also integrates simulation with generative AI tasks. This exploration includes developing unified frameworks for simulations, image rendering, and natural language processing, showcasing the versatile applicability of MPM in tackling contemporary computational challenges.

The dissertation of Yuxing Qiu is approved.

Joseph M. Teran

Glenn D. Reinman

Jingsheng Jason Cong

Chenfanfu Jiang

Demetri Terzopoulos, Committee Chair

University of California, Los Angeles

2024

To my parents, Feng, and friends

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

As I stand at this significant milestone of my academic journey, my heart brims with profound gratitude. Foremost, I extend my heartfelt thanks to my advisors, Professor Chenfanfu Jiang and Professor Demetri Terzopoulos. Professor Jiang, whose brilliance and diligence are unparalleled, has been a beacon of inspiration. His guidance and insights have not only shaped my academic endeavors, but also kindled in me a zest for life and learning. Professor Terzopoulos, in my moments of doubt and challenge, has been a pillar of support, offering guidance that has been instrumental in my Ph.D. journey.

My sincere appreciation also goes to the other members of my thesis committee, Professors Jingsheng Jason Cong, Glenn Reinman, and Joseph Teran. I am grateful for their insights on improving my work. Their patience and thoughtful discussions have also been a source of great support for me.

I am also very grateful to Professor Song-Chun Zhu, my initial advisor, whose wisdom and guidance in the realms of AI and cognition profoundly influenced my early academic path. Alongside him, Professor Ying Nian Wu, the co-advisor of the lab in which I previously worked, deserves my utmost thanks for his kind-hearted and wise support.

My time at Adobe Research and LightSpeed Studios was enriched by the incredible minds I was privileged to work with. I extend my deepest thanks to Dr. Jui-Hsien Wang, Dr. Jérémie Dumas, Dr. Timothy Langlois, and Dr. Danny Kaufman for their invaluable mentorship. Also, a special note of gratitude to my mentor, Dr. Kui Wu, for his patient and selfless guidance and support during my last internship, and to my colleagues Dr. Haozhe Su, Siyu Zhang, Tao Huang, Jerry Hsu, Zhen Chen, and Tongtong Wang, for their camaraderie and support.

I would also like to express my thanks to all the incredible researchers with whom I collaborated at the Oak Ridge National Laboratory. My gratitude goes to Dr. Stuart Slattery, whose insights and expertise in both research and engineering were crucial in overcoming research challenges. Dr. Samuel Temple Reeve's patient assistance and support, especially with the Cabana integration and benchmarking on the Summit supercomputer,

have been invaluable. My thanks also extend to Dr. Kwitae Chong, Austin Isner, Dr. Duan Z. Zhang, and Dr. David Hyde for their collaborative spirit.

I am grateful to all my collaborators at UCLA and UPenn. I would like first to thank Dr. Yixin Zhu who offered much assistance and guidance in my research and daily life during my early years at UCLA. I am grateful to have collaborated with Dr. Minchen Li, whose brilliance and work ethic have greatly inspired me, and with Dr. Xinlei Wang, whose exceptional skills impressed me greatly. The companionship of Dr. Yu Fang, Ziyin Qu, Xuan Li, Yunuo Chen, Yue Li, and Dr. Joshuah Wolper during those long nights working towards the SIGGRAPH 2020 submission deadline and beyond has been unforgettable. To my friends and collaborators at UCLA, including Zeshun Zong, Tianyi Xie, Yadi Cao, Chang Yu, and Ying Jiang, thank you all for your warm friendship and support. To Dr. Chi Zhang, Dr. Baoxiong Jia, Dr. Yixin Chen, Dr. Qing Li, Dr. Hangxin Liu, Dr. Xu Xie, Zeyu Zhang, Ziyuan Jiao, Shuwen Qiu, Dr. Lifeng Fan, Dr. Ruiqi Gao, Dr. Yuanlu Xu, Dr. Siyuan Qi, Pan Lu, Muzhi Han, Dr. Zhenliang Zhang, Dr. Siyuan Huang, and Ziheng Zhou, thank you for being my great labmates and friends. All of you have taught me so much throughout our time together with all your numerous shining points, and I sincerely hope that our paths continue to cross in the future.

A heartfelt thanks to Professor Paul Eggert, Professor Raghu Meka, Ning Wang, Jason Cheng, and all the TAs and LAs with whom I've worked at UCLA. The teaching experiences have been immensely enriching, teaching me about collaboration and communication. To the students enrolled in those quarters, your enthusiasm and passion have been a constant reminder of my own academic aspirations. My gratitude also extends to Joseph Brown, Helen Tran, and Glenda Jones for their assistance with administrative matters.

To Shuwen Qiu, my first female friend at UCLA, whose empathy and encouragement have been a source of strength, and to my friends Dr. Yiling Chen, Ziqi Zhu, Yifan Gao, Dr. Chuchu Wu, Pingying Chen, Yuxi Ma, Xinwen Guo, Yi Zhou, Dr. Yiming He, Yuxin Chen, Xinyu Yang, and Siyu Ma, thank you for the laughter and support through the challenging times, especially during the pandemic.

And to my family, whose unwavering support and understanding have been my foundation: My parents, who have always strived to understand and support my choices, have been a constant source of love and encouragement. My partner, Dr. Feng Gao, providing his steadfast belief, companionship, and encouragement, has been my rock through all the toughest times.

Reflecting upon this adventure, I am reminded of the countless individuals who have touched my life. My deepest gratitude to everyone who has been a part of my Ph.D. journey — your combined impact has been immeasurable. Thank you for being a part of my story.

VITA

2020–2023	Teaching Assistant, Department of Computer Science, UCLA.
2023	Research and Development Intern, Tencent America, LightSpeed Studios
2022	Visiting Scholar, Oak Ridge National Lab
2021	Research Intern, Adobe Research
2018–2020	Graduate Student Researcher, Department of Compute Science, UCLA.
2015–2018	M.S. in Computer Science, Beihang University (BUAA).
2011–2015	B.E. in Computer Science, Beihang University (BUAA).

PUBLICATIONS

X. Xie, H. Liu, Z. Zhang, **Y. Qiu**, F. Gao, S. Qi, Y. Zhu, S.-C. Zhu. VRGym: A Virtual Testbed for Physical and Interactive AI. *Proceedings of the ACM Turing celebration conference-China*, 2019.

X. Wang*, **Y. Qiu*** (*equal contribution), S. Slattery, Y. Fang, M. Li, S. Zhu, Y. Zhu, M. Tang, D. Manocha, C. Jiang. A Massively Parallel and Scalable Multi-GPU Material Point Method. *ACM Transactions on Graphics (SIGGRAPH 2020 Proceedings)*, 39(4), pp.30-1, 2020.

S. Slattery, S.T. Reeve, C. Junghans, D. Lebrun-GrandiÃ©, R. Bird, G. Chen, S. Fogerty, **Y. Qiu**, S. Schulz, A. Scheinberg, A. Isner. Cabana: A Performance Portable Library for Particle-Based Simulations. *Journal of Open Source Software*, 7(72), p.4115, 2022.

Y. Qiu, F. Gao, M. Li, G. Thattai, Y. Yang, C. Jiang. TPA-Gen: A Multi-Modal Data Generative Method for Text and Physics-Based Animation. *arXiv preprint arXiv:2211.13887*, 2022.

Y. Qiu, S. Reeve, M. Li, Y. Yang, S. Slattery, C. Jiang. A Sparse Distributed Gigantic Resolution Material Point Method. *ACM Transactions on Graphics (SIGGRAPH 2023 Proceedings)*, 42(2), pp.1-21, 2023.

T. Xie*, Z. Zong*, **Y. Qiu***, X. Li* (*equal contribution), Y. Feng, Y. Yang, C. Jiang. PhysGaussian: Physics-Integrated 3D Gaussians for Generative Dynamics. *arXiv preprint arXiv:2311.12198 (Under Review)*, 2023.

CHAPTER 1

Introduction

As humans, we inhabit a world subject to underlying principles and laws that influence our behaviors and interactions. Relentless research into understanding the mechanics and rules of motion has empowered the application of physical laws and theories in transformative ways. These advancements have had profound impact on various sectors, from manufacturing to entertainment. In recent decades, the advent and ubiquity of computer technologies has dramatically reshaped everyday life and yielded indispensable tools for a wide range of applications. Concurrently, researchers and engineers have harnessed and expanded computational capabilities to address practical challenges rooted in physical laws. Amidst these technological advancements, physics-based simulation stands as a pivotal discipline. Its importance is increasingly acknowledged as pivotal in the fields of computer graphics and scientific computing.

To bring the natural phenomena of deformable solids and fluids into both virtual and real-world applications, researchers have delved into Lagrangian and Eulerian computational methods. Lagrangian Finite Element Method (FEM) is a cornerstone for the simulation of solid-like behaviors, while the Eulerian approaches are suitable for fluid-like behaviors. The ideas behind both methods are combined in hybrid simulations, offering enhanced numerical stability in accurately predicting the behavior of materials with intermediate characteristics — those that are deformable like solids, but undergo complex changes akin to fluids. The employment of Cartesian grids in these hybrid methods naturally handles self-collisions and achieves unified computations across different materials. Meanwhile, Lagrangian particles effectively track mass, momentum, and deformation, adeptly managing significant deformations and topological variations.

The inception of Particle-In-Cell (PIC) and Fluid Implicit Particle (FLIP) techniques in fluid simulation has paved the way for the introduction in the last decade of the Material Point Method (MPM), a hallmark in physics-based simulation and animation. The MPM has demonstrated exceptional flexibility and versatility in a wide array of continuous material simulations within a unified framework, including snow (Stomakhin et al., 2013; Gaume et al., 2018), sand (Klár et al., 2016; Daviet and Bertails-Descoubes, 2016; Gao et al., 2018b; Zhao et al., 2019), cloth (Jiang et al., 2017; Fei et al., 2018; Guo et al., 2018; Montazeri et al., 2019), hair (Jiang et al., 2017; Fei et al., 2018; Guo et al., 2018), and non-Newtonian fluids and foam (Yue et al., 2015; Ram et al., 2015; Yue et al., 2018; Nagasawa et al., 2019).

The primary challenge in MPM applications lies in the extensive computational requirements. First, the particle degree of freedom is generally high for most of the applications due to the following facts: (i) The material particles carry a plethora of attributes crucial for motion and deformation. (ii) In practice, achieving physically accurate simulations often necessitates using hundreds of thousands to millions of particles. This large particle count is also vital for capturing detailed material information for specific scientific uses. Second, as a hybrid approach, the MPM involves frequent attribute transfers between particles and grids in each computational step, numerous times within each video frame. The necessity for C1 continuity in Eulerian interpolating functions for achieving stability means using at least second-order spline kernels for interpolations (*e.g.*, quadratic or cubic B-splines), requiring each particle to interact with multiple neighboring grid cells. Furthermore, in 3D, each grid cell typically contains multiple particles, intensifying the computational load. During particle-to-grid data transfers, atomic additions, essential for simultaneous data writing by multiple particles, lead to serialization bottlenecks in the parallelization of MPM pipelines. These concurrent particle data writings create bottlenecks in MPM parallelizations. Moreover, the grid-to-particle transfer process, despite not requiring atomic additions, faces performance limitations due to the random access patterns in the particle and grid data.

To enhance simulation performance and scalability, and to broaden the applicability in both virtual and real-world scenarios of physics-based simulations, particularly the MPM, this dissertation embarks on a series of innovative research investigations from diverse perspectives.

1.1 Thesis Contributions and Overview

Massively Parallel and Scalable Multi-GPU Material Point Method.

The MPM, already implemented on contemporary architectures such as GPUs, has seen enhancements in performance and scalability. The evolution of GPU capabilities, particularly the advancements in scalable atomic operations, enables the execution of numerous MPM steps in a condensed timeframe on a single GPU. Despite this, as previously discussed, the MPM inherently demands intensive computational effort, especially in particle computations and data transfers. To further refine the MPM's efficiency on single GPUs, Chapter 3 introduces a reformulated MPM pipeline and a specialized Array-of-Structure-of-Array (AoSoA) particle data structure, achieving speed improvements of 2× to 3× over conventional MPM methods. Moreover, Chapter 3 also discusses the expansion of the algorithm to multi-GPU settings using static partitioning and computation-communication masking strategies. Utilizing multiple GPUs overcomes the single GPU's limitations in memory and computational power, enabling simulations at a scale of 134 million particles within a minute on an 8-GPU workstation.

A Sparsely Distributed Gigantic Resolution Material Point Method.

Contrary to state-of-the-art solvers that focus on exploiting single machines for MPM simulation scale and efficiency, Chapter 4 further aims to overcome the limitations of single machines in terms of memory and computational power. In Chapter 4, we introduce a device-portable distributed simulation system that can be applied to Lagrangian, Eulerian, as well as hybrid simulation methods. The design of this system is multifaceted. The first objective is to allow for the effortless interchange of computing devices without altering any code. Secondly, the system is intended to expand upon existing libraries and incorporate a distributive sparse grid data structure to conserve device memory usage. Thirdly, it aims to facilitate dynamic workload distributions across various machines, thereby improving simulation efficiency and robustness. Finally, the system is intuitively designed, ensuring its user-friendliness for engineers, scientists, and artists alike. Chapter 4 showcases the system's capacity to customize large-scale simulations, maximizing device resource utilization while embodying the aforementioned features.

Real-time Simulation Applications.

The aforementioned explorations, particularly the development of distributed systems, have effectively shattered the scale barrier for the MPM. With sufficient computing resources, these systems can be linked to facilitate extensive simulations. This breakthrough led us to revisit the potential of simulation methods, especially the MPM algorithm, in real-time applications with limited computing resources. Chapter 5 delves into various applications of physics-based simulations: (i) a physically interactive environment for robot training, and (ii) MPM simulations within real-time interactive gaming environments.

The Material Point Method for AI Generation Tasks.

Having achieved high-performance simulations for real-time applications and detailed large-scale simulations, we next turn our attention to the future possibilities of simulation technology. Simulation, at its core, is a generative process that starts from an initial state and evolves according to physical laws. This perspective leads us to consider the evolution of 2D generation in the computer vision (CV) field as a potential roadmap for the future of 3D generation. A notable trend in image generation is the empowerment of general users, who lack formal training in art, to become creators through the use of AI models and natural language. In Chapter 6, we explore the potential of bridging simulations, images, and natural languages. Our initial trial in §6.1 involves the development of an automatic sampler that generates data points encompassing 3D simulations, rendered videos, and language descriptions. This effort aims to integrate various aspects of human knowledge into a unified representation, paving the way for future large-scale model training and testing. Additionally, we introduce a universal representation method that is applicable to both simulation and rendering in §6.2. This innovation enables the capture and simulation of real-world data, including the deformation of specific objects imaged in videos.

CHAPTER 2

Background of The Material Point Method

MPM is a hybrid simulation method that uses particle and grid representations to discretize the simulation domain. In this chapter, we briefly summarize the deformation theories that MPM build upon and the basic algorithm for the MPM. We refer to (Jiang et al., 2016) for a more detailed introduction to deformation theory and discretization algorithms.

2.1 Deformation Theory

The motion of the material-of-interest is conceptualized as a transformation from material point position X to their deformed configuration x. This process allows for the definition of the deformation gradient as

$$\mathbf{F}(\mathbf{X}, t) = \frac{\partial \phi}{\partial \mathbf{X}}(\mathbf{X}, t) = \frac{\partial \mathbf{x}}{\partial \mathbf{X}}(\mathbf{X}, t). \tag{2.1}$$

Different types of constitutive models, which often use the derivatives of the deformation gradient $mathbfF$ and its determinant, are used to explain how different materials behave. Using $J(\mathbf{F})$ to represent the determinant of $\mathbf{F}$, we can compute the differentials of the determinant as $\delta J = J\mathbf{F}^{-T}\delta\mathbf{F}$. Accordingly, we can write the derivative as

$$\frac{\partial J}{\partial \mathbf{F}} = J\mathbf{F}^{-T}. \tag{2.2}$$

The simulation cases happen mostly in 2D or 3D, in which cases $\mathbf{F}$ is a 2D or 3D matrix, and $J\mathbf{F}^{-T}$ can be written as another matrix represented by the polynomial of $\mathbf{F}$'s entries.

According to that, we can compute the other differentials according to the concrete requirements of the constitutive models.

With SVD computations being generally involved in constitutive computations, another commonly used differential is the differential of $\mathbf{F}$'s SVD results. The SVD of $\mathbf{F}$ is represented as

$$\mathbf{F} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T. \tag{2.3}$$

Here, $\mathbf{U}$ and $\mathbf{V}$ are rotation matrix, which means $\mathbf{U}^T\mathbf{U} = \mathbf{I}$ and $\mathbf{V}^T\mathbf{V} = \mathbf{I}$. Taking differentials on both sides of Equation 2.3, we get

$$\delta\mathbf{F} = \delta\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T + \mathbf{U}\delta\boldsymbol{\Sigma}\mathbf{V}^T + \mathbf{U}\boldsymbol{\Sigma}\delta\mathbf{V}^T. \tag{2.4}$$

According to $\mathbf{U}^T\mathbf{U} = \mathbf{I}$ and $\mathbf{V}^T\mathbf{V} = \mathbf{I}$, we have

$$0 = \delta\mathbf{U}^T\mathbf{U} + \mathbf{U}^T\delta\mathbf{U}, \tag{2.5}$$

$$0 = \delta\mathbf{V}^T\mathbf{V} + \mathbf{V}^T\delta\mathbf{V}. \tag{2.6}$$

According to Equation 2.3, we can compute the $\delta\Sigma$ as $\delta\boldsymbol{\Sigma} = \mathbf{U}^T\delta\mathbf{F}\mathbf{V} - \mathbf{U}^T\delta\mathbf{U}\Sigma\mathbf{V}^T\mathbf{V} - \mathbf{U}^T\mathbf{U}\boldsymbol{\Sigma}\delta\mathbf{V}^T\mathbf{V}$. After simplification, we have

$$\delta\Sigma = \mathbf{U}^T\delta\mathbf{F}\mathbf{V} - \mathbf{U}^T\delta\mathbf{U}\Sigma - \Sigma\delta\mathbf{V}^T\mathbf{V}. \tag{2.7}$$

Then, by substituting Equation 2.5 and Equation 2.6 we know that the diagonal entries of $\mathbf{U}^T\delta\mathbf{U}$ and $\mathbf{V}^T\delta\mathbf{V}$ are zeros. Thus, we can rewrite the $\delta\boldsymbol{\Sigma}$ as:

$$\begin{aligned} \delta\Sigma &= \operatorname{diag}(\mathbf{U}^T\delta\mathbf{F}\mathbf{V} - \mathbf{U}^T\delta\mathbf{U}\Sigma - \Sigma\delta\mathbf{V}^T\mathbf{V}) \\ &= \operatorname{diag}(\mathbf{U}^T\delta\mathbf{F}\mathbf{V}). \end{aligned} \tag{2.8}$$

Notation	Meaning	Relation to E, ν
E	Young's modulus	
ν	Poisson's ratio	
μ	Shear modulus	$\mu = \frac{E}{2(1+\nu)}$
λ	Lamé modulus	$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}$
κ	Bulk modulus	$\kappa = \frac{E}{3(1-2\nu)}$

Table 2.1: Material Parameters.

2.2 Constitutive Models

In this section, we introduce several commonly used constitutive models in this dissertation. The commonly used parameters in discussing the constitutive models are summarized in Table 2.1.

To simulate the deformation behaviors of different materials, we use different constitutive models. In detail, a given constitutive model describes how the internal forces are generated from material deformations, or, in other words, how the material stresses are computed from the material strain. With these internal forces computed, plus the influences of external forces (such as winds and gravity), we are able to compute the acceleration of discretized material parts. Then, according to the governing motion equations, we can further compute the velocities and advect material parts to different positions. By recording the evolution process of material point positions, we achieve the simulation of the objects with corresponding materials.

Generally, continuum mechanics uses energy density functions according to material deformation gradients ($\Psi(\mathbf{F})$) to present constitutive models that depict elastic deformations. Accordingly, we can compute the first Piola-Kirchoff stress, $\mathbf{P}$, according to $\mathbf{P} = \frac{\partial \Psi}{\partial \mathbf{F}}$. For any general isotropic elastic materials, we can further compute the Cauchy stress through

$$\begin{aligned} \sigma &= \frac{1}{J}\mathbf{P}\mathbf{F}^T \\ &= \frac{1}{\det(\mathbf{F})}\frac{\partial \Psi}{\partial \mathbf{F}}\mathbf{F}^T. \end{aligned} \tag{2.9}$$

The plasticity is generally modeled as a constraint on the deformation gradient. By decomposing $\mathbf{F}$ into an elastic and a plastic part (*i.e.* $\mathbf{F} = \mathbf{F}^E\mathbf{F}^P$), we can have the material parts "remembering" the elastic deformations $\mathbf{F}^E$ and consider $\mathbf{F}^P$ as the new local rest state of the material. Changing the rest configuration setups, we actually enforce permanent deformations on the material.

Fixed Corotated Elasticity. The First Piola-Kirchoff stress, $\boldsymbol{P}$, is defined as

$$\boldsymbol{P} = 2\mu(\boldsymbol{F}^E - \boldsymbol{R}) + \lambda(J-1)J\boldsymbol{F}^{E^{-T}}, \tag{2.10}$$

where $\boldsymbol{R} = \boldsymbol{U}\boldsymbol{V}^T$ and $\boldsymbol{F}^E = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ is the singular value decomposition of elastic deformation gradient. J is the determinant of $\boldsymbol{F}^E$ (Jiang et al., 2015).

StVK Elasticity. The irst Piola-Kirchoff stress, $\boldsymbol{P}$, is defined as

$$\boldsymbol{P} = \boldsymbol{U}\left(2\mu\boldsymbol{\Sigma}^{-\mathbf{1}}\boldsymbol{\epsilon} + \lambda\,\mathrm{sum}(\boldsymbol{\epsilon})\boldsymbol{\Sigma}^{-\mathbf{1}}\right)\boldsymbol{V}^T, \tag{2.11}$$

where $\boldsymbol{\epsilon} = \log(\boldsymbol{\Sigma})$ and $\boldsymbol{F}^E = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ (Klár et al., 2016).

Neo-Hookean Elasticity. The First Piola-Kirchoff stress, $\boldsymbol{P}$, is defined as

$$\boldsymbol{P} = \mu(\boldsymbol{F}^E - \boldsymbol{F}^{E^{-T}}) + \lambda\log(J)\boldsymbol{F}^{E^{-T}}, \tag{2.12}$$

where J is the determinant of $\boldsymbol{F}^E$ (Jiang et al., 2015).

Drucker-Prager Plasticity. The return mapping of Drucker-Prager plasticity for sand (Klár et al., 2016) is, given $\boldsymbol{F} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ and $\boldsymbol{\epsilon} = \log(\boldsymbol{\Sigma})$,

$$\boldsymbol{F}^E = \boldsymbol{U}\mathcal{Z}(\boldsymbol{\Sigma})\boldsymbol{V}^T, \tag{2.13}$$

$$\mathcal{Z}(\boldsymbol{\Sigma}) = \begin{cases} \mathbf{1}, & \operatorname{sum}(\boldsymbol{\epsilon}) > 0, \\ \boldsymbol{\Sigma}, & \delta\gamma \leqslant 0, \text{ and } \operatorname{sum}(\boldsymbol{\epsilon}) \leqslant 0, \\ \exp\left(\boldsymbol{\epsilon} - \delta\gamma \frac{\hat{\epsilon}}{\|\hat{\epsilon}\|}\right), & \text{otherwise}. \end{cases} \tag{2.14}$$

Here $\delta\gamma = \|\hat{\boldsymbol{\epsilon}}\| + \alpha\frac{(d\lambda+2\mu)\operatorname{sum}(\boldsymbol{\epsilon})}{2\mu}$, $\alpha = \sqrt{\frac{2}{3}}\frac{2\sin\phi_f}{3-\sin\phi_f}$, and ϕ_f is the friction angle. $\hat{\epsilon} = \operatorname{dev}(\epsilon)$.

Von-Mises Plasticity.

Similar to Drucker-Prager plasticity, given $\boldsymbol{F} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ and $\boldsymbol{\epsilon} = \log(\boldsymbol{\Sigma})$,

$$\boldsymbol{F}^E = \boldsymbol{U}\mathcal{Z}(\boldsymbol{\Sigma})\boldsymbol{V}^T,$$

where

$$\mathcal{Z}(\boldsymbol{\Sigma}) = \begin{cases} \boldsymbol{\Sigma}, & \delta\gamma \leqslant 0, \\ \exp\left(\boldsymbol{\epsilon} - \delta\gamma \frac{\hat{\epsilon}}{\|\boldsymbol{\epsilon}\|}\right), & \text{otherwise}, \end{cases} \tag{2.15}$$

and $\delta\gamma = \|\hat{\boldsymbol{\epsilon}}\|_F - \frac{\tau_Y}{2\mu}$. Here τ_Y is the yield stress.

Herschel-Bulkley Plasticity. We follow Yue et al. (2015) and take the simple case where $h = 1$. Denote $\boldsymbol{s}^{\text{trial}} = \operatorname{dev}(\boldsymbol{\tau}^{\text{trial}})$, and $s^{\text{trial}} = ||\boldsymbol{s}^{\text{trial}}||$. The yield condition is $\Phi(s) = s - \sqrt{\frac{2}{3}}\sigma_Y \leqslant 0$. If it is violated, we modify s^{trial} by

$$s = s^{\text{trial}} - \left(s^{\text{trial}} - \sqrt{\frac{2}{3}}\sigma_Y\right) / \left(1 + \frac{\eta}{2\mu\Delta t}\right).$$

$\boldsymbol{s}$ can then be recovered as $\boldsymbol{s} = s \cdot \frac{\boldsymbol{s}^{\text{trial}}}{||\boldsymbol{s}^{\text{trial}}||}$. Define $\boldsymbol{b}^E = \boldsymbol{F}^E\boldsymbol{F}^{E^T}$. The Kirchhoff stress $\boldsymbol{\tau}$ is computed as

$$\boldsymbol{\tau} = \frac{\kappa}{2}\left(J^2 - 1\right)\boldsymbol{I} + \mu \operatorname{dev}\left[\det(\boldsymbol{b}^E)^{-\frac{1}{3}}\boldsymbol{b}^E\right].$$

2.3 The MPM Algorithm

In the MPM context, a continuum body is discretized into a set of Lagrangian particles (*i.e.* material points). The particles hold physical attributes to depict the motion and the deformation of the material. These attributes typically include mass (m_p), velocity ($\boldsymbol{v}_p$), deformation gradient ($\boldsymbol{F}_p$), and affine momentum ($\boldsymbol{C}_p$). The grid nodes, on the other hand, generally store mass (m_i) and momentum ($m_i\boldsymbol{v}_i$) that are transferred from particles. In other words, the Eulerian grids are treated as auxiliary scratchpad variables to perform spatial derivative computations and boundary condition enforcement.

With the basic first-order explicit MPM time integration scheme, we discretize the time into a sequence of time steps $t = 0, t^1, t^2, \ldots$. A fixed time-step size Δt is adopted according to the CFL condition.

Based on the notations mentioned before, let m_p, $\boldsymbol{x}_p^n$, $\boldsymbol{v}_p^n$, $\boldsymbol{F}_p^n$, $\boldsymbol{\tau}_p^n$, and $\boldsymbol{C}_p^n$ denote the mass, position, velocity, deformation gradient, Kirchhoff stress, and affine momentum on particle p at time t_n, while m_i^n, $\boldsymbol{x}_i^n$ and $\boldsymbol{v}_i^n$ refer to the mass, position, and velocity on grid node i at time t^n. In this dissertation, we assume the particle masses are invariant according to the mass conservation law. The explicit MPM algorithm can be summarized as below:

1. **Particles-to-Grid (P2G)** Compute grid mass and momentum from particles. In this dissertation, the Affine Particle-in-Cell (APIC) transferring scheme (Jiang et al., 2015) is applied by default. Also, the particle elastic stress force contributions are computed and transferred to the grid nodes (first term of Eq. (2.17)).

$$\begin{aligned} m_i^n &= \sum_p w_{ip}^n m_p, \quad m_i^n \boldsymbol{v}_i^n = \sum_p w_{ip}^n m_p \left(\boldsymbol{v}_p^n + \boldsymbol{C}_p^n \left(\boldsymbol{x}_i - \boldsymbol{x}_p^n\right)\right), \\ \boldsymbol{\tau}_p^n &= \boldsymbol{\tau}(\boldsymbol{F}_p^{E,n}). \end{aligned} \tag{2.16}$$

2. **Grid Update.** Update grid velocities based on external forces (second term of Eq. (2.17)). Also, we apply boundary conditions by restricting the momentum (or velocities) of specific grid nodes in this step according to specific application requirements.

$$\boldsymbol{v}_i^{n+1} = \boldsymbol{v}_i^n - \frac{\Delta t}{m_i} \sum_p \boldsymbol{\tau}_p^n \nabla w_{ip}^n V_p^0 + \Delta t \boldsymbol{g}. \tag{2.17}$$

3. **Transfer Grid to Particles.** Transfer velocities from grid nodes back to particles

and advect particle positions with their new velocities.

$$\boldsymbol{v}_p^{n+1} = \sum_i \boldsymbol{v}_i^{n+1} w_{ip}^n, \quad \boldsymbol{x}_p^{n+1} = \boldsymbol{x}_p^n + \Delta t \boldsymbol{v}_p^{n+1} \tag{2.18}$$

In addition, evolve particle strains (deformation gradient):

$$\nabla \boldsymbol{v}_p^{n+1} = \sum_i \boldsymbol{v}_i^{n+1} \nabla w_{ip}^{n\,T}, \quad \boldsymbol{F}_p^{\mathrm{E,\ tr}} = (\boldsymbol{I} + \nabla \boldsymbol{v}_p^{n+1}) \boldsymbol{F}^{E,n}. \tag{2.19}$$

According to the APIC (Jiang et al., 2015), we also need to update the affine momentum ($\boldsymbol{C}_p$). Here b is the B-spline degree, and Δx represents the Eulerian grid cell size.

$$\boldsymbol{C}_p^{n+1} = \frac{12}{\Delta x^2 (b+1)} \sum_i w_{ip}^n \boldsymbol{v}_i^{n+1} \left(\boldsymbol{x}_i^n - \boldsymbol{x}_p^n \right)^T, \tag{2.20}$$

CHAPTER 3

A Massively Parallel and Scalable Multi-GPU Material Point Method

Harnessing the power of modern multi-GPU architectures, we present a massively parallel simulation system based on the Material Point Method (MPM) for simulating the physical behaviors of materials undergoing complex topological changes, self-collision, and large deformations. Our system makes three critical contributions. First, we introduce a new particle data structure that promotes coalesced memory access patterns on the GPU and eliminates the need for complex atomic operations on the memory hierarchy when writing particle data to the grid. Second, we propose a kernel fusion approach using a new Grid-to-Particles-to-Grid (*G2P2G*) scheme, which efficiently reduces GPU kernel launches, improves latency, and significantly reduces the amount of global memory needed to store particle data. Finally, we introduce optimized algorithmic designs that allow for efficient sparse grids in a shared memory context, enabling us to best utilize modern multi-GPU computational platforms for hybrid Lagrangian-Eulerian computational patterns. We demonstrate the effectiveness of our method with extensive benchmarks, evaluations, and dynamic simulations with elastoplasticity, granular media, and fluid dynamics. In comparisons against an open-source and heavily optimized CPU-based MPM codebase (Fang et al., 2019) on an elastic sphere colliding scene with particle counts ranging from 5 to 40 million, our GPU MPM achieves over 100× per-time-step speedup on a workstation with an Intel 8086K CPU and a single Quadro P6000 GPU, exposing exciting possibilities for future MPM simulations in computer graphics and computational science. Moreover, compared to the state-of-the-art GPU MPM method (Hu et al., 2019a), we not only achieve 2× acceleration on a single GPU but our kernel fusion strategy and

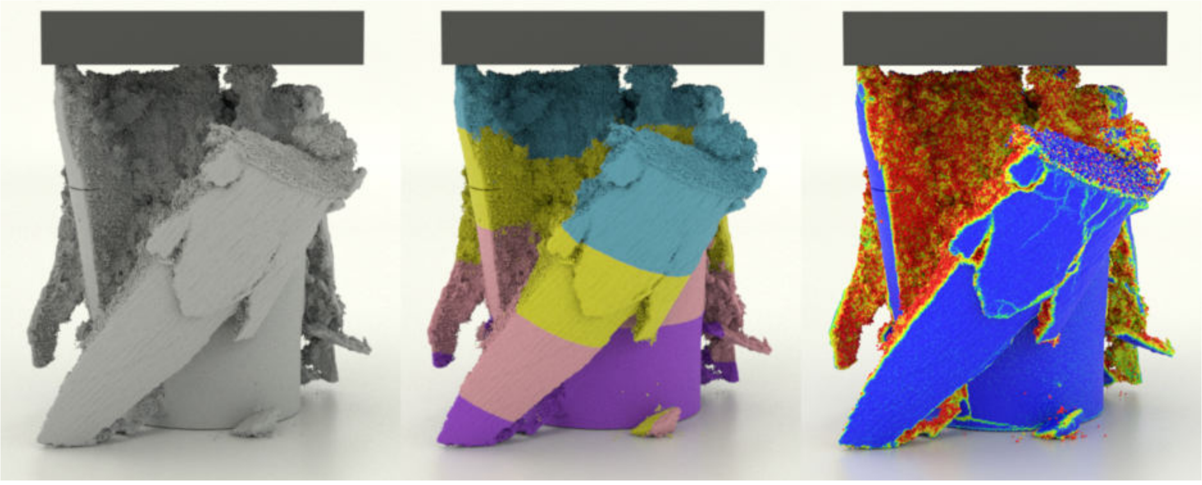


Figure 3.1: **Crushing concrete.** Our system enables this concrete crushing simulation (inspired by the hydraulic press) on a single workstation with 4 NVIDIA Quadro P6000 GPUs. This simulation contains 93.8 million particles on a 1024^3 grid, achieving a 3.9 min/frame performance. (Left) A concrete-style render. (Middle) Coloring by GPU. (Right) Coloring by the plastic volumetric strain for visualizing the damage propagation.

Array-of-Structs-of-Array (*AoSoA*) data structure design also generalizes to multi-GPU systems. Our multi-GPU MPM exhibits near-perfect weak and strong scaling with 4 GPUs, enabling performant and large-scale simulations on a 1024^3 grid with close to 100 million particles with less than 4 minutes per frame on a single 4-GPU workstation and 134 million particles with less than 1 minute per frame on an 8-GPU workstation.

3.1 Introduction

The Material Point Method (MPM) provides significant potential and opportunities to exploit parallelism on modern computing architectures. To date, most work on MPM performance has focused on how to thread the algorithm on conventional CPUs and, to a lesser extent, has attempted to exploit domain decomposition via the Message Passing Interface (MPI). This line of work includes threading particle and grid operations as well as handling the transfer of data between particles and grids, which often results in a bottleneck when parallelized. With the advent of modern accelerator architectures such as GPUs, enough memory and bandwidth are available on the accelerator to perform MPM simulations with the significant number of particles and grid cells needed for generating expansive and high-resolution visual scenes. These new accelerator performance capabilities, including advances in native support for scalable atomic operations on floating-point numbers, result in the ability to perform a relatively large number of computations in a relatively small amount of computing time on a single GPU.

However, the memory and computing power of a single GPU is not limitless. To the best of our knowledge, no prior work has attempted to develop a performant algorithm for MPM that utilizes multi-GPUs in a shared memory context. Given numerous multi-GPU platforms being deployed by vendors both in server and workstation configurations, algorithm development for multi-GPUs will enable us to perform even larger-scale simulations at a significantly reduced computing time on what could be considered commodity hardware. Re-designing MPM algorithms for multi-GPUs is non-trivial. First, as a hybrid simulation method, MPM involves complex operations on particles, grids, and the transfer of data between them. Compared to developing a scalable single GPU algorithm, algorithms utilizing multi-GPUs require inter-GPU communications to program the majority of these operations. Second, MPM simulations usually target scenarios with explosions, fractures, highly deformable solids, and fluids. For such highly dynamic problems, the particle population will fluctuate in time as a function of space and therefore incur load imbalance when multiple devices are used.

To further increase the computational power available to perform MPM simulations in both single- and multi-GPU execution contexts, we make three novel contributions. First, we reformulate the conventional GPU-based MPM pipeline with a fused *G2P2G* kernel function, which not only enables both single- and multi-GPU performance gains, but is also generalizable to prior MPM designs (Wolper et al., 2019; Fang et al., 2019). Secondly, we develop a specialized Array-of-Structs-of-Array (*AoSoA*) particle data structure tailored for our *G2P2G* kernel utilizing the delayed-ordering technique that maximizes bandwidth efficiency. Finally, we propose a domain-decomposition-invariant computation scheme tailored for multi-GPUs, which significantly reduces the additional memory overhead due to PCIe connections among GPUs. As a result, our method outperforms the heavily optimized state-of-the-art single-GPU MPM implementation (Hu et al., 2019a; Gao et al.,

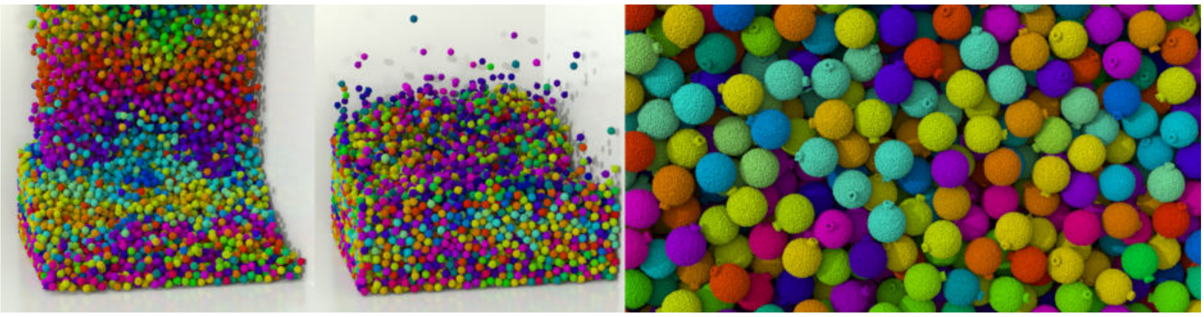


Figure 3.2: **Bomb falling.** We run the *bomb falling* test on 8 GPUs with 134M particles (6,688 bombs with nonlinear finite-strain hyperelastic constitutive models) and grid resolution $512 \times 2048 \times 512$. On average, each frame is finished within 1 minute, indicating the scalability of our proposed multi-GPU MPM pipeline.

2018b) with a 2× speedup and achieves almost linear scaling on multi-GPUs. Moreover, we accomplished large-scale MPM simulations with truly enormous particle and grid cell counts.

We organize this chapter as follows. We review related work in §3.2, serving as the basis for our comparisons to the state-of-the-art. In §3.3, we introduce our improved single-GPU algorithm and outline the kernel fusion procedure and data structure details. In §3.4, we present the new multi-GPU algorithm and include a discussion of memory management and communication, while details on the implementation of our code are provided in §3.5. In §3.6, we present results on an extensive selection of benchmarks using a variety of materials. We also include an analysis of both strong and weak scaling of our algorithm as a function of the number of GPUs, which shows significant performance improvements over the state-of-the-art in GPU implementations as well as significant performance gains when using the GPU algorithm relative to a highly optimized CPU implementation. Finally, in §3.7, we conclude the chapter with a discussion of the limitations of our new algorithm and the resulting avenues for future work.

3.2 Related Work

3.2.1 HPC-Based Simulations in Computer Graphics

Parallelized Solvers The rapid development of modern CPU and GPU architectures makes it possible to accelerate physics-based simulation by parallelizing existing algorithms using threads, domain decomposition, or some combination thereof. A basic approach to parallelism executes an algorithm using multiple threads on multiple CPU cores on a single node, supported by shared memory programming models such as Intel TBB (Willhalm and Popovici, 2008) and OpenMP (Dagum and Menon, 1998). Recent examples include that of Li et al. (2019b), which performs domain decomposition within an optimization time integrator for CPU-based parallel evaluation and factorization of subdomain Hessians.

To achieve even better performance, researchers have developed parallel simulation algorithms for the GPU, which enables more floating-point operations on a per-Watt and per-dollar basis when compared to traditional multi-core CPU architectures. In literature, parallelization of large-scale simulations in fluid dynamics, such as Eulerian fluids (Chentanez and Müller, 2011, 2013; Cohen et al., 2010; Pfaff et al., 2010), Lagrangian fluids (Goswami et al., 2010; Vantzos et al., 2018; Winchenbach et al., 2016; Amada et al., 2004; Macklin et al., 2014), and the hybrid Eulerian-Lagrangian solvers (Wu et al., 2018; Chentanez et al., 2015), have all been implemented on a single GPU. For GPU simulations of solid mechanics, Gao et al. (2018b) and Hu et al. (2019a) implement the high-performance Moving Least Squares MPM (Hu et al., 2018), and Bernstein et al. (2016) and Hu et al. (2019a) explore parallel Finite Element Method (FEM) techniques.

Due to the ever-increasing demand for computational resources and new hardware releases by vendors, developing multi-GPU solutions is an inevitable trend for physics-based simulations to utilize modern computing hardware effectively. Recent work, such as multi-GPU-based Smoothed Particle Hydrodynamics (SPH) (Verma et al., 2017; Rustico et al., 2012; Domínguez et al., 2013; Xiong et al., 2013), FEM (Li et al., 2020a), and parallelized Poisson equation solvers (Ament et al., 2010; Liu et al., 2016), has demonstrated the plausibility of physics-based simulation on multi-GPU platforms.

Another stream of high-performance physics-based simulation utilizes distributed platforms; *i.e.*, cloud-based simulation. Early work commonly makes use of MPI to assign computing tasks to distributed nodes automatically. To better adapt to large topological changes that can occur during a simulation, methods for fluid load balancing in cloud-based simulations are proposed (Mashayekhi et al., 2018; Shah et al., 2018a), showing significant potential to achieve high-performance distributed fluid animations.

Efficient Data Structures From the Eulerian viewpoint, the MPM simulation domain is represented by a discretized structured grid where the volumetric data involved is often spatially sparse in large-scale 3D simulations due to dynamic particle populations. This fact has inspired extensive studies on hierarchical and sparse data structures (Liu et al., 2018; Museth, 2013; Hoetzlein, 2016; Setaluri et al., 2014) to create efficient data access patterns that mitigate the effects of sparsity. For instance, OpenVDB (Museth, 2013), one of the most popular sparse storage schemes in computer graphics, dynamically arranges blocks of a grid in a hierarchical manner similar to B+ tree. Hoetzlein (2016) extends this idea further on GPU and proposes GVDB Voxels with an efficient memory pooling architecture to support dynamic topology changes. Alternatively, SPGrid (Setaluri et al., 2014; Gao et al., 2018b) has proven to be a promising data structure in both MPM (Hu et al., 2018; Aanjaneya et al., 2017) and other fluid simulations (Aanjaneya et al., 2017; Liu et al., 2016; Setaluri et al., 2014). Additionally, methods such as spatial-temporal coherent spatial hashing are also explored to take advantage of the spatial sparsity (Tang et al., 2016; Weller et al., 2017; Wang, 2018). Recently, Hu et al. (2019a) introduce the Taichi programming model, which exposes high-level interfaces for developing and processing spatially sparse multi-level data structures and benefits researchers by eliminating redundant work in data and performance management.

On the other hand, from the Lagrangian viewpoint, particle information is generally unstructured and stored in an Array-of-Structure (AoS) (Hu et al., 2019a) or Structure-of-Array (SoA) (Gao et al., 2018b) compact layout. *SoA* promotes coalesced memory accesses of particle data when sequential threads access sequential memory addresses.

However, the particles need to be re-sorted after each time step to maintain such an efficient data access pattern (Gao et al., 2018b). *SoA* is less efficient in gather/scatter operations such as serialization, where long strides in memory are needed to access all data for a single particle, resulting in the use of multiple memory pages. In contrast, *AoS* maps more readily to the concept of a particle and performs well in cases of un-coalesced memory access patterns due to the locality of the data for a single particle. However, such a memory layout prevents coalesced reads and writes of particle data, thereby significantly inhibiting both GPU and vectorized CPU performance when coalescing is possible. To exploit both the advantages mentioned above and mitigate the disadvantages, we propose an MPM-centric Array-of-Structs-of-Array (*AoSoA*) data structure for better performance, which possesses the qualities of both *SoA* and *AoS*. Inspired by the Hierarchical Particle Buckets introduced by Hu et al. (2019a) and Bailey et al. (2013), we store particles' data in a hierarchical manner with *AoSoA*. The particles are reorganized in low-level bins and high-level block-buckets to conserve the efficiency of both the memory access and the data transfer.

3.2.2 The Material Point Method in Computer Graphics

Introduced by Sulsky et al. (1994, 1995), the MPM is an extension of Hybrid-Fluid-Implicit-Particle (FLIP) (Brackbill and Ruppel, 1986; Zhu and Bridson, 2005) from fluid animation in hydrodynamics to general elastoviscoplastic materials simulation in solid mechanics. As one of the most promising discretization choices in physics-based simulation, MPM has been used for simulating numerous materials and diverse phenomena. Prior work includes snow (Stomakhin et al., 2013; Gaume et al., 2018), granular materials (Klár et al., 2016; Daviet and Bertails-Descoubes, 2016; Gao et al., 2018b; Zhao et al., 2019), viscoelastic solids (Fang et al., 2019), cloth (Jiang et al., 2017; Fei et al., 2018; Guo et al., 2018; Montazeri et al., 2019), hair (Jiang et al., 2017; Fei et al., 2018; Guo et al., 2018), and non-Newtonian fluids and foam (Yue et al., 2015; Ram et al., 2015; Yue et al., 2018; Nagasawa et al., 2019). Additionally, other complex phenomena have been simulated with MPM including melting (Stomakhin et al., 2014; Gao et al., 2018b), baking (Ding et al.,

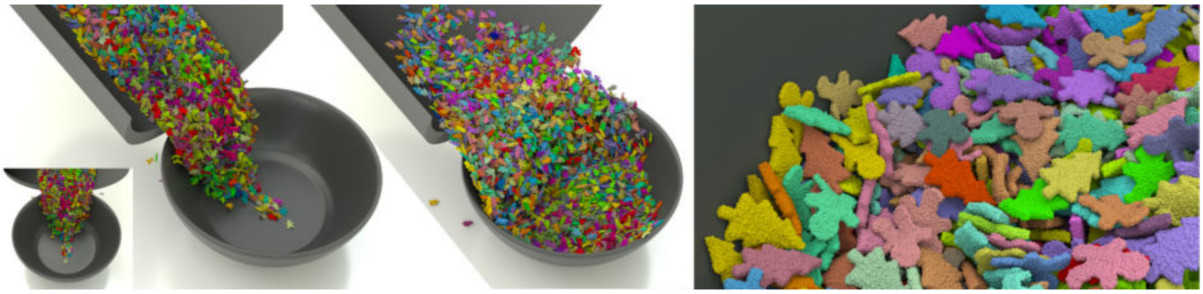


Figure 3.3: **Candy bowl**. We show how to gather a bowl of candies by pouring 6786 candies from a tube. This simulation runs on 4 GPUs with 23M particles and grid resolution 1024 × 1024 × 512. Each frame is simulated with approximately 4 seconds, demonstrating the acceleration achieved by our algorithm to simulate and generate large-scale scenes that cannot be efficiently processed using traditional MPM on smaller computing systems.

2019), topological changes and fracture (Wretborn et al., 2017; Wang et al., 2019; Wolper et al., 2019), multiple-material interaction (Tampubolon et al., 2017; Hu et al., 2018; Gao et al., 2018a; Yan et al., 2018; Han et al., 2019), frictional contact and collision (Ding and Craig, 2019), *etc.* Recently, GPU-based acceleration (Gao et al., 2018b; Hu et al., 2019a), as well as spatially (Gao et al., 2017; Yue et al., 2018) and temporally (Fang et al., 2018) adaptive methods have been proposed to improve the computational efficiency of MPM.

Prior work on GPU MPM has focused on the design of GPU-tailored data structures for both particles and grids, as well as the corresponding mathematical operations to achieve better performance; each sub-step is redesigned for GPU (largely using CUDA up to this point). For instance, both Gao et al. (2018b) and Hu et al. (2019a) reduce write conflicts during the *Particles-to-Grid (P2G)* transfer, either by CUDA warp-level reductions (Gao et al., 2018b) or the random-shuffling of particles inside each block (Hu et al., 2019a). As reported in these papers, using GPUs can considerably improve performance compared to traditional CPU-based MPM.

3.2.3 Data Structures and Simulations in HPC

AoSoA Particle data structures are largely responsible for CPU and GPU performance as they dictate memory access patterns when parallelizing codes via threading or vectorization.

The most commonly adopted memory layouts in HPC are *SoA* and *AoS*. Specifically, in terms of particle data layouts, *SoA* stores all particle data components (*e.g.*, mass, each velocity direction, *etc.*) in separate arrays, ensuring coalesced memory access when reading/writing the same component of adjacent particles. However, when performing non-coalesced operations like particle-grid transfers, additional sorting methods are required to maintain particle order to guarantee that consecutive thread indices access consecutive particle indices (Gao et al., 2018b). In contrast, *AoS* reduces the need for sorting in non-coalesced operations, since its improved memory locality has better performance when randomly accessed. However, the same data components of adjacent particles are no longer adjacent in memory (Hu et al., 2019a), resulting in a non-coalesced data access pattern even when coalescing would otherwise be possible. To take advantage of both the *AoS* and *SoA* layouts, researchers have proposed *AoSoA* to achieve both coalescing/vectorizing data access patterns whenever possible and to improve performance via memory locality when it is not (Wald, 2010; Weber and Goesele, 2014). §3.3.2 discusses the implementation of *AoSoA* in greater detail.

HPC Simulation Frameworks For scientific simulations in HPC, accelerators are already being adopted broadly with a number of the current top supercomputers leveraging GPU hardware to achieve the majority of their performance (TOP500.org, 2019). In these types of supercomputing configurations, thousands of accelerators are combined with a high-speed interconnect with the goal of reaching exascale-class levels of floating-point operations in the next few years. To achieve portability across the variety of accelerator architectures in use in modern supercomputers, several programming models, libraries, and frameworks have been developed to allow for the manipulation of data structures (*e.g.*, *AoS* vs. SoA) and parallel loop patterns based on the underlying hardware. Examples of performance portability programming models include Kokkos (Edwards et al., 2014) and its derivative libraries Cabana (Slattery et al., 2021), a portable library for writing multi-GPU particle simulations via the *AoSoA* data structure as well as multi-GPU grid-based simulations which can be used to implement hybrid particle-in-cell algorithms such as MPM. Other examples include SMILEI (Derouillat et al., 2018), an open-source multi-purpose Particle-In-Cell (PIC) implementation that has been applied to a wide range of physics studies, from astrophysical plasma to relativistic laser-plasma interaction.

An analysis of the accelerated machines on the TOP500 list, as mentioned above, and a review of the computational patterns in libraries (such as Kokkos and Cabana) reveal that multi-GPU programming on such machines is relegated mainly to a single GPU per MPI rank. Such a programming model allows for a more straightforward description of parallelism and more accessible programming. However, in the case of many simulation algorithms such as MPM, it forces the application more quickly into the strong scaling limit by further subdividing the problem into smaller pieces. By developing a multi-GPU shared memory programming model in this work, we aim to gain additional performance on modern supercomputers by reducing the number of subdomains needed for parallelization, thus increasing the number of GPUs per MPI rank and reducing the dependence on the performance of the network, including its bandwidth and latency. The multi-GPU advancements in this work are particularly important for machines such as Summit (Facility, 2018) as a subset of the GPUs on each compute node has a significantly faster local interconnect than the PCI connection and therefore would strongly benefit from the MPI-free algorithm presented here.

3.3 Improved Single-GPU Algorithm

Before introducing our algorithmic improvements, we first summarize the essential steps of a conventional first-order MPM time integration scheme for incremental dynamics from t^n to t^{n+1} ($\Delta t = t^{n+1} - t^n$).

1. **Particles-to-Grid (P2G)**.

   Transfer mass and momentum from particles to grid nodes: $\{m_p, m_p\boldsymbol{v}_p^n\} \rightarrow \{m_i, m_i\boldsymbol{v}_i^n\}$;

2. **Grid Update**.

   Update grid velocities with either explicit or implicit time integration: $\boldsymbol{v}_i^n \rightarrow \boldsymbol{v}_i^{n+1}$;

3. **Grid-to-Particles (G2P) and Particle Advection**.

   Transfer velocities from grid nodes to particles, evolve particle strains, and project particle deformation gradients for plasticity (if any). Update the particle positions with their new velocities: $\{\boldsymbol{v}_i^{n+1}\} \rightarrow \{\boldsymbol{v}_p^{n+1}, \boldsymbol{F}_p^{n+1}\}$, $\{\boldsymbol{p}_p^n, \boldsymbol{v}_p^{n+1}\} \rightarrow \{\boldsymbol{p}_p^{n+1}\}$;

4. **Partition Update**.

   Maintain the sparse data structure topology by updating the active-block array and the mapping from block coordinates to array indices.

Typically, each particle has several attributes including mass m_p, position $\mathbf{x}_p$, velocity $\boldsymbol{v}_p$, deformation gradient $\boldsymbol{F}_p$, initial volume V_p^0, and the affine matrix $\mathbf{C}_p$, which is the same as the velocity derivative matrix in MLS-MPM (Hu et al., 2018). On the grid, each node generally stores the grid mass m_i and the momentum $m_i\boldsymbol{v}_i$, from which the nodal velocity v_i can be calculated.

For the grid data structure, we use the GPU-SPGrid (Gao et al., 2018b), a variant of the CPU-based SPGrid (Setaluri et al., 2014). Although both GPU- and CPU-based SPGrid use *SoA* layout for blocks, their underlying arrangements of blocks are fundamentally different. CPU-based SPGrid (Setaluri et al., 2014) leverages the extensive hardware acceleration mechanisms inherent in the virtual memory system for performant sequential and stencil operations on grid data. The GPU-based SPGrid (Gao et al., 2018b), on the contrary, explicitly manages grid blocks with spatial hashing, which maps spatial block coordinates to block indices in an array. Both structures can maintain the sparsity of the grid and minimize the memory footprint. In this work, we use the quadratic B-spline weighting kernel for both mass and velocity transfers between particles and grids, and therefore each particle is associated with $3 \times 3 \times 3$ grid nodes in 3D (3×3 in 2D). However, our algorithm works for all typical interpolating kernels that use compact stencils.

When parallelizing MPM algorithms, the general concern about the performance is the transfer operations between particles and grids, *i.e.*, *P2G* and *G2P*. These sub-steps become even more crucial to the performance of implicit schemes where significantly more transfer operations are required. Below, we present two techniques to accelerate the transfer operations: 1) *Grid-to-Particles-to-Grid (*G2P2G*)*, an innovative and fused algorithmic kernel, and 2) Array-of-Structs-of-Array (*AoSoA*), a new application of a particle data structure with an associated parallel loop strategy.

3.3.1 G2P2G

Similar to many PIC/FLIP-based solvers, the MPM method uses particles to represent discrete Lagrangian elements of the simulated continuum material and employs the Eulerian background grid as the auxiliary scratchpad to compute spatial derivatives and apply boundary conditions. Within a conventional MPM formulation, the particle states are the primarily evolved quantities. When parallelizing the MPM algorithm, the computations in all the sub-steps (*i.e.*, *P2G*, *grid update*, *G2P*, *particle advection*, and *partition update*) are implemented in separate GPU kernels. Prior methods adopt GPU-tailored data structures for particles and grids and reduce write-conflicts during *P2G*, either through CUDA warp-level reductions (Gao et al., 2018b) or by randomly shuffling particles inside each block (Hu et al., 2019a). Although each kernel is highly optimized, the synchronization of the grid state required by the *grid update* incurs the separation of kernels, hindering the GPU MPM performance. This limit calls for additional treatments.

To further reduce the latency on modern GPU architectures, reordering the traditional time-stepping strategy and combining several kernels are necessary. In each traditional MPM time step, particle quantities have to be streamed in and out of the GPU global memory for multiple times, *i.e.*, in *P2G* and *G2P*. Unlike the GPU MPM kernels implemented by Gao et al. (2018b) where $\boldsymbol{F}_p$ is updated at the end of the *G2P* kernel, Hu et al. (2019a) reorder the pipeline by moving the update of $\boldsymbol{F}_p$ to the beginning of the *P2G* kernel before the *P2G* transfer to reduce the redundant particle data accesses. With this modification, the evolved $\boldsymbol{F}_p$ can be reused immediately inside the *P2G* kernel, thus removing the operations to write and reload the updated $\boldsymbol{F}_p$ to and from the GPU global memory in both the current *G2P* kernel and the next *P2G* kernel. Using a similar strategy, we could further reorder the traditional MPM time step and reformulate a new

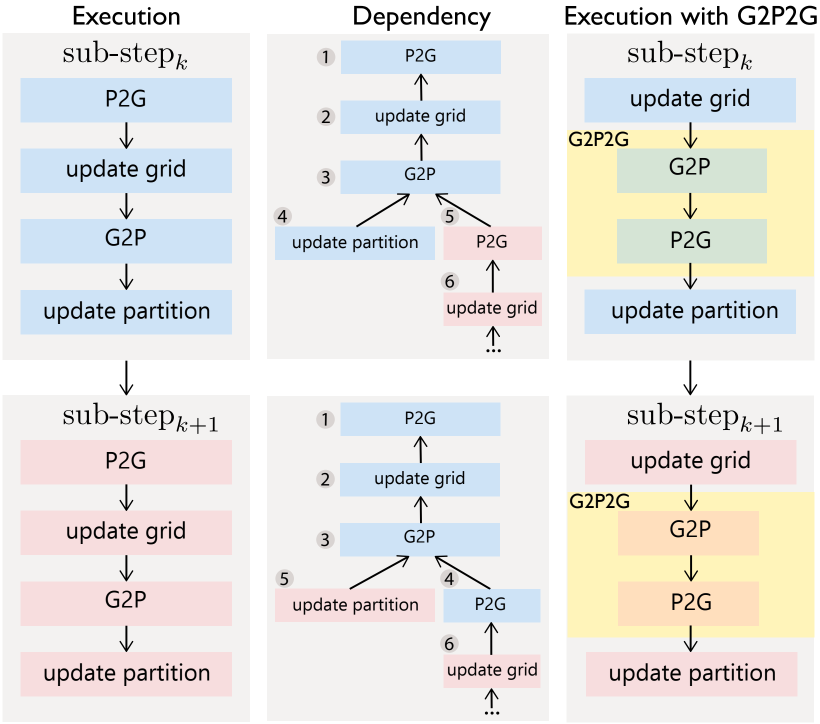


Figure 3.4: **MPM pipeline reformulation.** The left column shows the traditional MPM pipeline, and the middle one illustrates an analysis of data dependencies among the sub-steps. The exchange of the *update partition* at the current time step k and the *P2G* at the next time step $k+1$ would not break the data dependency or the execution correctness, making it possible to reorder and assemble the *G2P* and the *P2G* to form a more efficient *G2P2G* kernel, as shown on the right.

kernel for better efficiency.

We start by analyzing the data dependencies among adjacent MPM sub-steps. As shown in the left column of Fig. 3.4, we observe some order constraints on data dependencies and execution orders of the sub-steps: 1) The *P2G* must be finished before the *grid update*, and the *G2P* is performed after all the grid states been evolved. 2) The *partition update*, wherein the particle-grid mapping and the sparse grid data structure are maintained, depends only on the results of *G2P*, *i.e.*, the advected particle positions. 3) The *P2G* transfer relies on the particle-grid mapping, *i.e.*, particles need to know to which grid nodes they should rasterize to, which leads to the dependency between the *partition update* in the current time step k and the *P2G* in the next time step $k+1$. The first two observations exhibit strict data dependencies, which are unchangeable to ensure correct computations. The third one, however, is a weak dependency, since the particle-grid mapping can be staggered differently. Therefore, we can reformulate the execution order of the sub-steps for better performance should the strict data dependencies were preserved.

Following the above analysis, we devise a novel *G2P2G* kernel by grouping the *G2P* in time step k and the *P2G* in time step $k + 1$ together; see Fig. 3.4 for a graphical illustration. Specifically, during the *G2P*, transferring the velocity $\boldsymbol{v}_p$ and any other higher-order velocity modes of the particles can be interpolated from grids to update particle positions and deformation gradients. When grouping the *G2P* and the *P2G* together, these interpolated attributes can be referenced immediately for both the *particle updates* and the next momentum transfer from particles to grids, converting these quantities to temporary variables within the kernel instead of arrays allocated in GPU global memory; the only particle attributes that need to be preserved are the mass, positions, and deformation gradients. With such a *G2P2G* reformulation, the new MPM pipeline inverts the traditional MPM time step by regarding the grid states as the primarily evolved quantities in each time step, with particles treated as intermediate integration points instead. At a high level, this *G2P2G* reformulation not only eliminates twice of transfer kernel launches and twice of particle data accesses for each time step, which significantly improves the performance but also reduces the particle storage. Note that, in addition to refactoring an explicit time step as presented in this work, the *G2P2G* approach could also be applied to implicit MPM schemes where the transfer process can take up to 90% of the wall time of a given simulation.

As for the particle-grid mapping strategy, the traditional GPU MPM solvers (*e.g.*, Gao et al. (2018b) and Hu et al. (2019a)) employ an off-by-one particle-grid mapping, wherein each particle block only touches $2 \times 2 \times 2$ grid blocks in both the *P2G* and the *G2P* transfer kernels. After the *particle advection*, the particles may move out of their original particle blocks, and the next *P2G* could then write to a different set of $2 \times 2 \times 2$ grid blocks. Although the *partition update* kernel may remap the particles to grids to ensure the *P2G* still loads only $2 \times 2 \times 2$ grid blocks in the next time step, the *partition update* and the *P2G* only possess a weak dependency; *i.e.*, the correctness of the calculation would still be guaranteed if the next *P2G* is executed immediately after the

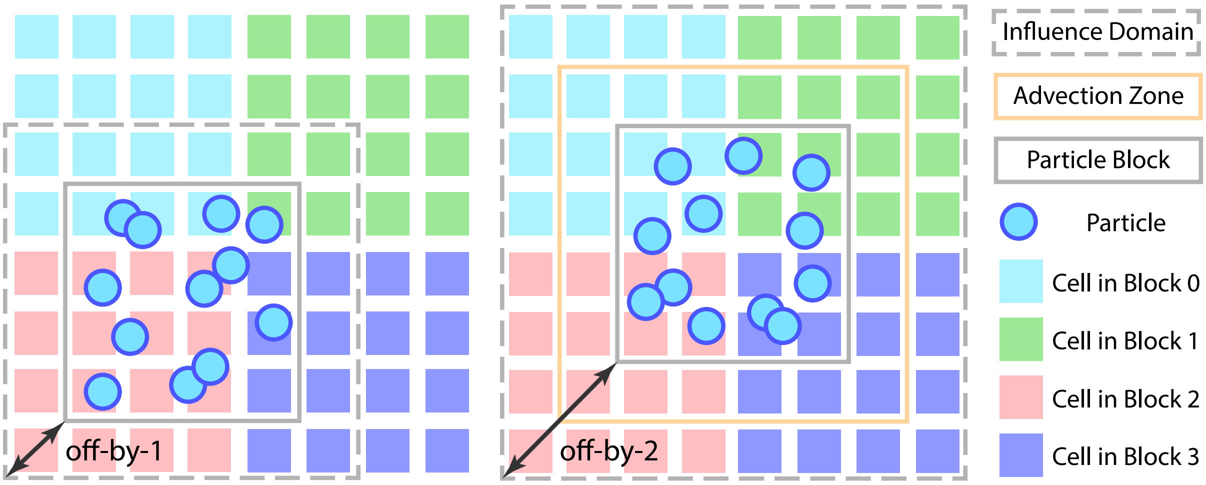


Figure 3.5: **Different staggered mappings.** Each square represents a cell in space, marked with a color that indicates the block it is located in. During the transfer, particles represented by circles contribute properties to the background grid. (Left) The conventional GPU MPM pipeline adopts an off-by-one staggered mapping between blocks and particles for more efficient use of the shared memory. (Right) The *G2P2G* pipeline adopts an off-by-two strategy: particles from a block should be located at least two-cell distance from the border of the arena. Such a design ensures the particles to stay in the same blocks after CFL-bounded advection in the *G2P2G*.

G2P without updating the partition. What does change is that the data accessed in the *P2G* kernel may need to involve more grid blocks. To eliminate the influence of *particle advection* on the grid blocks accessed by the *G2P* and the following *P2G* kernel, we design an off-by-two mapping strategy, making it possible to reorganize the time step without sacrificing the performance during the *P2G* transfer. Below, we present the technical details needed to adopt this new *G2P2G* pipeline.

Particle-Grid Offset In general, the "scratchpad" pattern is critical to the performance of transfer operations; it refers to a software-managed local data buffer stored in shared memory in the context of GPU computing. For the *P2G* kernel, this buffer stores the grid attributes, *i.e.*, mass, and momentum, to which particles will rasterize. For the *G2P* kernel, on the other hand, it stores the attributes of grid nodes from which the particle states would be interpolated. Instead of using a direct mapping between particles and blocks, traditional GPU MPMs use an off-by-one staggering strategy (Hu et al., 2019a; Gao et al., 2018b). In detail, a staggered mapping between particles and grid blocks with

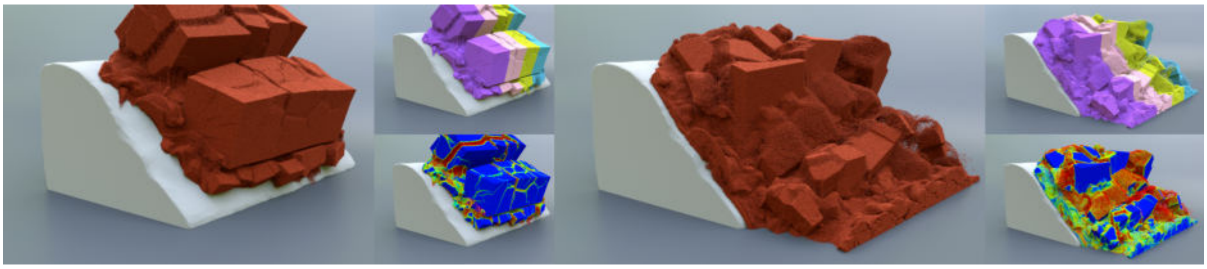


Figure 3.6: **Soil falling.** We show a *soil falling* test with Non-Associated Cam-Clay (NACC) running on 4 GPUs with 53M particles and grid resolution 512 × 512 × 512. We illustrate the Multi-GPU Static Partitioning by Particles (*MGSP*) on the right-top corner of each subfigure. On the right-bottom corner, the NACC-α (the plastic volumetric strain hardening variable) is also visualized to indicate the fracture pattern, where red indicates significant material fractures.

a one-cell-distance is applied to the *P2G* and the *P2G* kernel. In this way, each transfer kernel requires a small shared memory buffer with only 2 × 2 × 2 grid blocks loaded, as shown in the left-side of Fig. 3.5. Without such a staggering, 3 × 3 × 3 grid blocks (3 × 3 in 2D) will be needed, increasing the cost of both memory storage and the data accessing.

However, in the *G2P2G*, the off-by-one staggered mapping between particles and grid nodes cannot be used as it is impossible to keep the assumption that particles would only touch 2 × 2 × 2 grid blocks during transfers, since we now advect the particles during the *G2P2G* kernel execution. We solve this problem with an off-by-two staggered mapping, tailored for our *G2P2G* pipeline. Overall, the local buffer size remains the same as in prior off-by-one staggered mapping (Hu et al., 2019a; Gao et al., 2018b), *i.e.*, 2 × 2 × 2 grid blocks, with each block containing 4 × 4 × 4 grid nodes. In detail, bounded by a Courant-Friedrichs-Lewy (CFL) condition, particles would never move more than one-cell distance during the *particle advection*. Therefore, the grid cells that particles may write to during the *P2G* transfer would not extend by more than one cell in each 3D direction. When enforcing the particle-grid mapping with the off-by-two strategy, the touched grid blocks would not change for the *P2G* after the previous *G2P* and the *particle advection*. Therefore, the *G2P2G* pipeline reformulation does not increase the shared memory usage or the data-accessing cost. Note that if CFL condition is violated, the premise for the *G2P2G* pipeline reformulation will no longer be valid. Thus, the execution of the *G2P2G* kernel could fail due to the out-of-bound shared memory access. If such a situation happens, one needs to re-run the solver with a shorter stepping time until the CFL condition is satisfied.

Compute Dt The time step size dt for MPM evolution should be carefully chosen under the restriction of CFL condition to preserve the numerical stability while at the same time as large as possible to accelerate the simulation process. In order to satisfy both requirements, the maximum velocity of particles is typically used to compute dt. However, since the state of other particles cannot be inferred during the execution of a single *G2P2G* kernel thread, retrieving such a global quantity inside the *G2P2G* kernel is impossible. As a substitute, we use the maximum velocity of the grid nodes, which can be computed before entering the *G2P2G* kernel. Since the particle velocities are interpolated from the surrounding grid nodes, the maximum velocity of particles will not be larger than the maximum grid velocity, and therefore the CFL restriction will be conserved. Moreover, this method is more computationally efficient in dt estimation since the number of grid nodes is much less than the number of material particles. Although this approach estimates a more conservative dt, experimental results show little difference in the computed dt (less than 1%) between the computation performed with the maximum velocities of grid nodes and particles.

3.3.2 AoSoA

Particle data layouts and the corresponding memory access patterns also significantly influence performance, since the particle attributes constitute the majority of the simulation data. In general, for a gather-style transfer, the particle memory throughput is at least one order of magnitude larger than the throughput of the grid data, making it impossible to cache all the particle data in the limited GPU shared memory. However, it is feasible to cache the grid attributes in the corresponding *G2P* kernel. For a scatter-style transfer, on the other hand, each particle is commonly assigned to one specific thread, making particles invisible to each other. It is, therefore, more meaningful to cache the grid data instead of the particle attributes to the shared memory. In both cases, inside the *G2P* or the *P2G* kernel, there is at least a one-time reading from or writing to the GPU global memory to access the particle data, which cannot be cached for better performance. Therefore, optimizing the efficiency of particle data accesses from GPU global memory becomes one of the most significant factors when maximizing performance.

Although both state-of-the-art approaches (Hu et al., 2019a; Gao et al., 2018b) use the GPU-tailored SPGrid variant for grid storage, they adopt fundamentally different particle data structures and algorithmic strategies. Gao et al. (2018b) store particle attributes in an *SoA* layout and devises a *delayed-reordering* technique to maintain the particle order; without reordering, the change of the spatial distribution of particles may lead to an insufficient GPU cache line utilization and cause performance degradation. To get rid of the cost of the particle reordering, Hu et al. (2019a) use an *AoS* layout, making the performance less sensitive to the particle order. Nevertheless, the performance is still limited by the non-coalesced read/write of particle attributes from/to the GPU global memory.

To exploit the advantages of both *SoA* and *AoS* layouts without compromising performance, we devise an *AoSoA* data structure to store particle attributes. The particles are grouped according to their positions, such that particles mapping to the same block can be gathered together in the memory. We adopt an *SoA* structure to store the particle attributes inside each group, while the particle groups are organized using an *AoS* structure. With such a design, the proposed *AoSoA* particle data structure has the following advantages:

- As long as the *SoA* group size is a multiple of the CUDA warp size, each warp of threads can access (read and write) particle data in a coalesced manner to ensure bandwidth efficiency.

- The particles are grouped according to their positions, and the particle groups are organized in an *AoS* layout. Therefore, each block (a $4 \times 4 \times 4$ cell size in our pipeline) of particles resides in contiguous memory, easier for faster migration among multi-GPUs.

Note that the *SoA* layout does not possess such property as particle attributes are stridden across the GPU memory. Such a design suits better for the proposed *G2P2G* pipeline, wherein each CUDA block handles only one particle block.

- By organizing particles inside each particle block with a finer granularity, we can reduce memory usage by making each particle block to occupy a minimal amount of memory to accommodate the particles inside; see details in the *binning strategy* paragraph.

Particle Bins To devise an appropriate particle data structure that possesses these properties, we introduce the concept of particle bins, inspired by the designs of SPGrid (Setaluri et al., 2014) and *Hierarchical Particle Buckets* (Hu et al., 2019a; Bailey et al., 2013).

One intuitive idea is to group particle data in particle blocks such that particles that belong to the same grid block are gathered together. In a single particle block, the particle then becomes the basic unit, with the particle attributes corresponding to the grid channels in the conventional SPGrid. However, compared to the grid block, the particle block would suffer from the large granularity and the uncertainty of the in-use number of particles. In particular, the number of particles residing in a single block is generally orders of magnitude larger than the number of grid nodes, and each particle usually contains more attributes than a grid node. Thus, the actual size of a particle block could be much larger than a grid block. Additionally, the number of particles inside a particle block changes dynamically throughout the simulation, causing memory waste and additional bookkeeping operations.

To remedy these problems, we further group the particles inside a single block into particle bins; the size of a particle bin can be customized as needed. For performance considerations, we recommend setting the bin size to be a multiple of the thread group size on a given GPU architecture. For example, one can set the bin size as 32, which is the size of a CUDA warp on an NVIDIA GPU.

As illustrated in Fig. 3.7, particle data is organized in an *SoA* layout within each

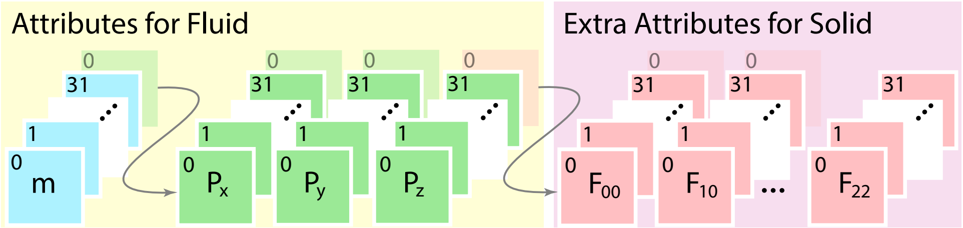


Figure 3.7: **Binning.** The internal layout of each particle bin is *SoA*. In this example, the array length of the particle bin equals to the thread group size on the NVIDIA GPU architecture (*i.e.*, 32 for an NVIDIA GPU warp) for coalesced memory access within a bin. The number of properties is flexible according to the material; we used 4 for fluids and 13 for solids.

particle bin. In this way, coalesced global memory accesses are ensured with the CUDA 32-, 64-, or 128-byte transactions that are aligned to these sizes. Another advantage of using particle bins instead of a monolithic *SoA* particle block is related to the page management in the virtual memory system. For example, a particle bin containing 64 particles, with each particle owning 16 float-type attributes, consumes a 4KB memory. In contrast, the particle block with the same setting would consume a space much larger than the 4KB configuration. Although the actual page size in CUDA might differ from the CPU page setting in practice, the particle binning strategy still provides the potential to better utilize the automatic CUDA unified virtual memory management.

The mapping from a block to its particles is implemented through the *Hierarchical Particle Bucket* design. Specifically, particle attributes and particle indices are stored separately in particle blocks and particle buckets, both in a 4 × 4 × 4 block granularity. Each particle is reached hierarchically through the block index and the local index inside the block. In practice, an upper bound of the particle bucket size is predetermined statically at compile-time, the maximum number of bins inside a block is predetermined by the bucket size when compiling, and the number of bins that each block contains can also be decided at run-time before execution. However, as illustrated in Fig. 3.8, such a uniformly allocated particle-block memory may cause a significant memory waste. To further reduce memory usage, we count the number of bins in-use and establish a mapping from the block ID to the bin ID through a lightweight exclusive scan.

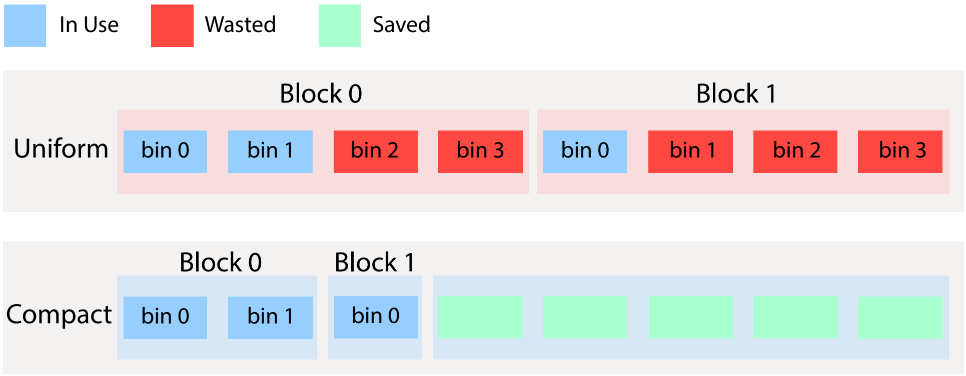


Figure 3.8: **Compact storage.** Particles are often unequally distributed in space. Hence, uniformly allocating fixed memory space for each particle block would result in a significant amount of unused memory. Instead, a more efficient strategy is to allocate just enough particle bins for each block according to the current distribution of particles.

Binning Strategy There are typically two strategies to reduce the write conflicts in the *P2G* kernel:

- Group particles by cells, reduce at warp-level, perform a single shared memory atomic increment per warp, and perform a single global memory atomic increment per block (Gao et al., 2018b).
- Leave particles unsorted to reduce the chances of atomic-write conflicts and avoid the warp-level reduction (Hu et al., 2019a).

The first method imposes restrictions on the order of the particles, which cannot be satisfied in the context of the *G2P2G* pipeline; particles may advect to the neighbor cells after the *G2P* transfer in the *G2P2G* kernel, breaking the cell-based sorted order.

Adopting ideas from the second strategy, we use a pseudo-coloring procedure and collect particles from different cells within a particle block to build the particle bins. The algorithm is outlined in Algorithm 1, which stops when there are not enough particles left to form a bin. With this strategy, particles inside a single bin are forced to write to different nodes unless the bin is formed after satisfying the stopping condition; *i.e.*, there exist at least two particles from the same cell inside this bin. As a result, the chance of write conflicts occurring within a warp is significantly reduced. Note that the warp aggregated atomic increment is required to ensure the correctness of the Algorithm 1 (refer to Adinets (2014) for more implementation details).

Update Particles Inside the *G2P2G* kernel, the maintenance of the particle structure must be performed after the *particle advection*. To ensure the execution correctness of the proposed *G2P2G* pipeline, we adopt a double buffer strategy for both particles and grids; *i.e.*, the *G2P2G* kernel reads from and writes to different particle/grid buffers. To maintain the particle structure, a naive scheme can be adopted to update the particle attributes in place while the particle orders are rearranged in an extra kernel incurring additional overhead. Following the *delayed-ordering* (Gao et al., 2018b), we postpone the particle reordering in the *G2P2G* kernel to the next time step.

Specifically, the updated particle attributes are written back to the particle blocks in the coalesced manner, and the particle indices are inserted into the particle buckets according to their updated positions. In the following time step, we determine the particle attributes in the previous particle block buffer from the indices saved in the current particle bucket. Theoretically, the particle block ID and its local index inside the block would change after the *particle advection*. However, we do not update the hierarchical particle indices immediately after updating particle positions. Instead, we compute the new indices from the advection vector and their original location; as indicated by the CFL bound, the particles will move at most a one-cell-distance in each time step. In practice, we use -1, 0, or $+1$ to indicate the particle's movement in x, y, or z direction to form the 3D advection vector (*i.e.*, one specific vector from a set of 27 possibilities). Given the previous block ID and the advection information, the new particle indices are then uniquely determined by a spatial hash with a 32-bit integer.

3.4 Multi-GPU Pipeline

Using multi-GPUs for MPM simulations affords significantly larger simulations and shortens the overall simulation time. To extend from using a single GPU to running

Algorithm 1 Distribute particles from cell buckets to block bucket

```
Input: cell_buckets, ppcs, maxppc, blockid, cellid          ▻ ppc: particles per cell
Output: block_bucket, ppbs                                  ▻ ppb: particles per block
  function DistributeParticlesc2b(test)
      laneid ← cellid  mod  warpsize
      ppc ← ppcs[cellid]
      pidic ← 0                                  ▻ pidic: local particle index in cell
      while pidic < maxppc do
          if pidic < ppc then
              pidib ←AGGREGATE_ATOMICADD(laneid, ppbs[blockid])
                                                ▻ pidib: local particle index in block
              block_bucket[pidib] ← cell_bucket[pidic]
          end if
          SYNCTHREADS_IN_BLOCK( )
          pidic ← pidic + 1
      end while
  end function
```

on multi-GPUs, we divide the whole simulation domain into partitions according to the device number and assign one partition to one GPU device. Load balancing is one of the essential considerations when distributing partitions for multi-GPU applications. Depending on the dynamics of the simulation, the same partitioning scheme could result in drastically different performances on various problems. Ultimately, the parallel efficiency of multi-GPUs is primarily determined by 1) how large the halo region is compared to the whole partition, and 2) how equally the partitions are distributed on all devices. Here, we focus on arranging the computations once the partitioning strategy is confirmed.

Additionally, we maintain sparse spatial information according to particle positions at each time step. The partition on each device is maintained through a list of activated blocks that cover all particles. Since the particles may rasterize to grid blocks, which can be halo blocks and shared by multi-GPUs, the attributes on gird blocks must be synchronized after the *P2G* transfer. Therefore, in addition to partitioning strategies, efficient utilization of multi-GPUs for MPM also needs to consider:

- *Halo Block Tagging*:

  tag the blocks that overlap partitions on other devices (*i.e.*, the halo blocks).

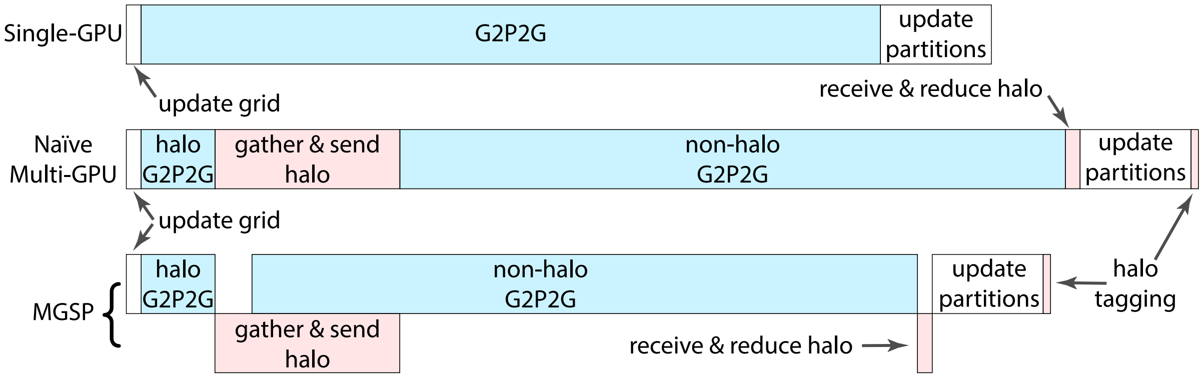


Figure 3.9: **Instruction pipelines.** The additional operations in the multi-GPU MPM compared to the single-GPU MPM are displayed in red. By masking these halo-region-related data transfers with the execution of the *G2P2G* kernel, one can achieve more optimized scaling results with multi-GPUs.

- *Halo Block Merging*:

  share block data in the halo region with other devices after executing the *G2P2G* kernel, for grid reduction and/or particle migration depending on partitioning strategies.

In the following subsections, we introduce detailed designs of two MPM-tailored variations of the most widely adopted partitioning methods, *i.e.*, the static geometric partitioning methods.

3.4.1 Multi-GPU Static Partitioning by Particles (MGSP)

MGSP is an ideal option for solid simulations, including elastic jellos, sand, and other granular materials, due to the stable halo distribution of solids. Since the overall shape of solid models remains intact even under large deformations, the halo regions typically reside on the model surfaces. Even when significant fractures happen (see examples in Figs. 3.1 and 3.6), the halo regions still only occupy a small portion of the whole partition.

Carrying out both *halo block tagging* and *halo block merging* relies heavily on multi-GPU communication. The latency of the related operations relies highly on the underneath hardware setup. In most consumer-level machines, multi-GPU devices are connected via the slow PCI-Express x16 Gen 3, which may lead to high communication latency. Fortunately, nearly all CUDA devices with compute capability of 1.1 or higher can

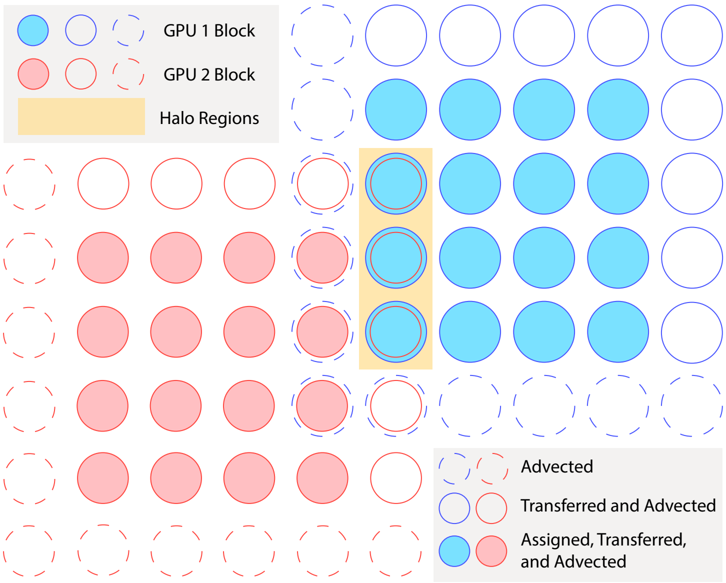


Figure 3.10: **Halo block tagging.** We tag grid blocks with three labels: assigned, transferred, and advected. Assigned grid blocks are the blocks where particles are residing in. Transferred grid blocks contain the grid-nodes that particles may write to; *e.g.*, with quadratic B-spline kernel, each particle may write to neighbor nodes within a three-cell distance. Advected grid blocks represent the blocks where particles may advect to after the *G2P* and the *particle advection*. The transferred and the advected grid blocks may contain no particles in the current time step. In the context of multi-GPU MPM, the partition from one device can overlap partitions from other devices. These overlapping regions (*i.e.*, halo regions) may include all three types of grid blocks. The grid data in the halo regions should be shared and synchronized by the corresponding GPUs to ensure execution correctness.

concurrently perform the memory copies and computing kernels. Therefore, it is possible to hide the latency by overlapping data transfers with computations (*i.e.*, *G2P2G*) through CUDA streams, as shown in Fig. 3.9.

Halo Block Tagging For each device, to acquire its intersections with other devices, the coordinates of the active blocks from all the other devices are gathered and then checked in a local hash table. We perform the *halo block tagging* as an additional step of the MPM algorithm; see Fig. 3.10. Due to the data dependencies, this step should not be overlapped with other computations. The data size of the active block coordinates increases with the growth of the simulation scale when more blocks are involved. However, in general, such data size is still small, making this additional overhead introduced by multi-GPU extensions insignificant.

Halo Grid Reduction The heavy workload of the *G2P2G* kernel provides the potential of overlapping the memory copies with the computations; see an illustration in Fig. 3.9. Based on the *halo block tagging* results, we split the particle blocks on each device into two groups. One group produces data for halo grid blocks during the *G2P2G* execution, whereas the other only works with the interior grid blocks. The *G2P2G* kernel is first launched for grids and particles inside the halo regions. After that, the following two operations are performed simultaneously with different CUDA streams, *i.e.*, 1) the halo grid attributes on each partition are gathered and sent to other partitions, and 2) the *G2P2G* kernel is evaluated on the particles and grids outside the halo regions on each device. In this way, the overhead of the memory copies among GPUs is masked with the *G2P2G* execution for interior particles and grids.

3.4.2 Multi-GPU Static Partitioning by Space (MGSS)

In an MPM simulation, the size of halo regions among multiple partitions may grow beyond a threshold, such that the latency of the non-halo *G2P2G* kernel is not high enough to mask the device-to-device memory copies. This situation is especially common for fluid simulations where fluids can theatrically mix (see Fig. 3.12 as an example), making halo sizes increase dramatically as time goes by. In such cases, re-partitioning particles is necessary for load balancing, and statically partitioning by space is a simple yet efficient strategy.

Halo Block Tagging Unlike in *MGSP*, the blocks in the halo region in *MGSS* can be tagged without the knowledge of any other partition. While updating the partition, blocks located in the spatially predefined halo region are directly tagged as halo blocks, and halo regions can be shared by two or more devices depending on the splitting scheme. The handling of the tagged halo grid blocks in *MGSS* is the same as in *MGSP*, but the particles moving to partitions on other devices are also migrated in addition to the grid data.

Halo Particle Migration Although the overhead of halo tagging in *MGSS* is avoided, and *halo grid reduction* in *MGSS* is similar to the one in *MGSP*, there is an additional task in *MGSS*; namely, particles moving out of the current domain must be migrated to the corresponding device. This operation is easily supported by our *AoSoA* particle data structure since particles are already grouped by blocks, and it is efficient to retrieve these particles before streaming. Furthermore, gathering particles in halo regions in bulk and streaming to other devices are always better than sending the same amount of data in pieces at a time, *e.g.*, particle by particle. Therefore, the same *AoSoA* particle data structure also specifies the particle buffer array for sending halo data to and receiving data from other devices.

The migration of halo particles in *MGSS* is inherently more memory-intensive than sharing halo grid blocks in *MGSP*. In general, particles have more quantities compared to grid nodes, and the number of particles inside each particle block is an order of magnitude larger than the number of grid nodes inside each grid block. Consequently, within the same halo region, particle blocks use significantly more memory than grid blocks. Moreover, the number of particle bins at each location near the boundary of a domain is only known after *G2P2G* kernels in all neighboring partitions are done, which breaks the premise of "compact storage" (§3.3.2). Fortunately, the maximum number of such halo blocks is bounded and known at compile time and is small compared to the whole domain. A simple workaround regarding the number of particle bins is to preserve a space conservatively that is fit for the maximum number of particles specified in the "Hierarchical Particle Bucket."

3.5 Implementation

In this section, we provide essential implementation details. Please refer to Appendix A for more compile-time settings and coding examples.

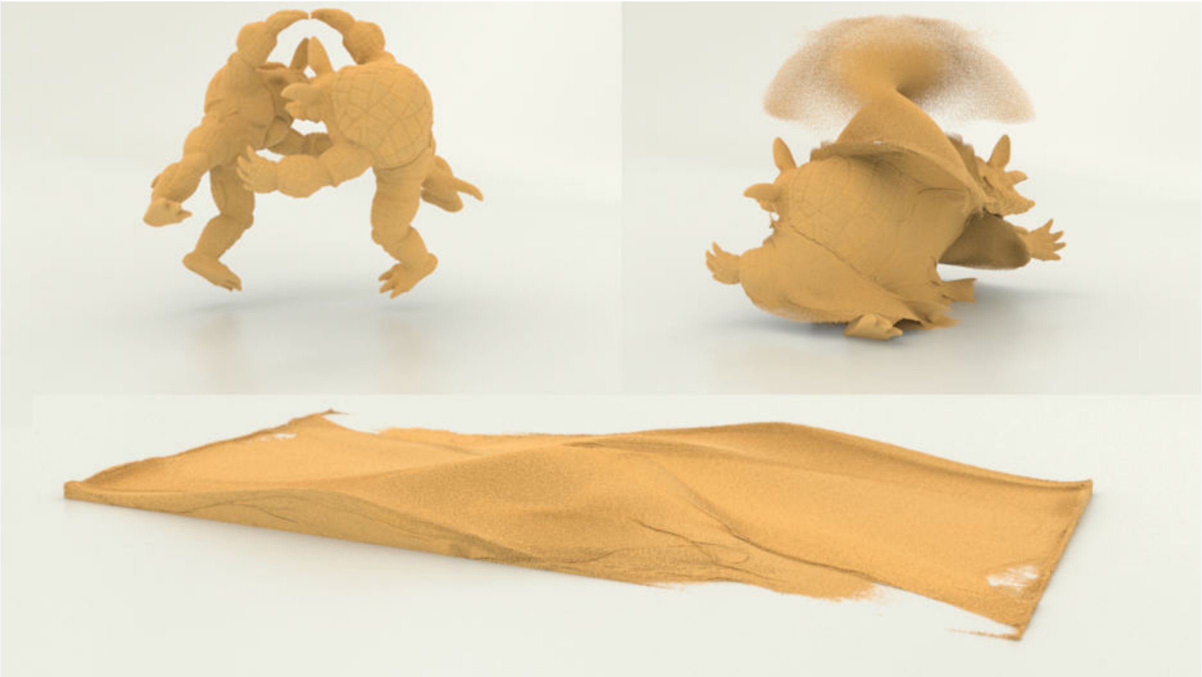


Figure 3.11: **Sand armadillo.** In this simulation, two sand armadillos hug and smash together using 4 GPUs with 55M particles and a grid resolution of $512 \times 512 \times 1024$. Fine details are captured with a small Δx.

3.5.1 Multi-GPU Communication

Although there are multiple CUDA libraries (*e.g.*, OpenSHMEM (Chapman et al., 2010), NCCL (Nvidia, 2019)) for inter-GPU communications, we directly use the low-level memory APIs for better control over the double-buffering scheme and halo communication. The halo data can be manually copied through the host, peer-to-peer, GPU-Direct, or with the use of Unified Virtual Memory (UVM) and let CUDA handle on-demand requests of halo data in UVM. However, page faults during kernel executions are expensive, and pre-fetching block-by-block before kernel launching requires many CUDA API calls. Hence, we choose to manually initiate the data transfers through multiple streams.

3.5.2 Memory Consumption

Table 3.1 summarizes the memory footprint in our pipeline and in that of Hu et al. (2019a); each particle block contains 4096 particles in total. As shown in Fig. 3.7, there

	Particle Buffer		Grid	Bucket
	Fluid	Solid		
Conventional	278528 n	409600 n	1024 m	16384 n
G2P2G	131072 n	425984 n	2048 m	16384 n

Table 3.1: **Memory budget.** We compare the number of particle blocks and grid blocks with the block size $4 \times 4 \times 4$. Here, numbers followed by n stand for the number of particle blocks and m for the number of grid blocks. We use 64 as the maximum number of particle-per-cell inside each grid cell; this setting is more than sufficient for most MPM simulations (8 is the typical setting). We have not observed any violations in any examples. Note that the data in the second row represents the total memory cost from the double buffering required by the *G2P2G* kernel.

are 16 bytes for a fluid particle and 52 bytes for a solid particle in our *G2P2G* pipeline. In contrast, in the conventional pipeline, there are 68 bytes and 100 bytes, respectively. In our pipeline, the per-particle storage size is reduced substantially, especially in the fluid case, due to particle velocity being a kernel-local quantity. However, we need to maintain two copies of the data structures due to the double-buffer strategy.

3.5.3 Material-Dependent Computation

The constitutive model-related particle-wise operations (elasticity, plasticity, *etc.*) are implemented in separate device functions, and the correct function for a specific material to call is automatically handled when utilizing the C++ sum type *variant*. Thus, different materials are easily supported with little changes in our approach.

3.5.4 Generalizations to Other MPM Methods

Not only is the *AoSoA* particle data structure compatible with different algorithmic or material choices, but the *G2P2G* kernel is essentially a general operator evolving the grid state through transfers. With reasonable efforts, the majority of existing MPM methods (including those of Jiang et al. (2017) and Wolper et al. (2019)) can also be implemented with *AoSoA+G2P2G*. In the case of the implicit MPM, matrix-free linear solvers, as in (Gao et al., 2018b), can be implemented directly with the *G2P2G* fusion

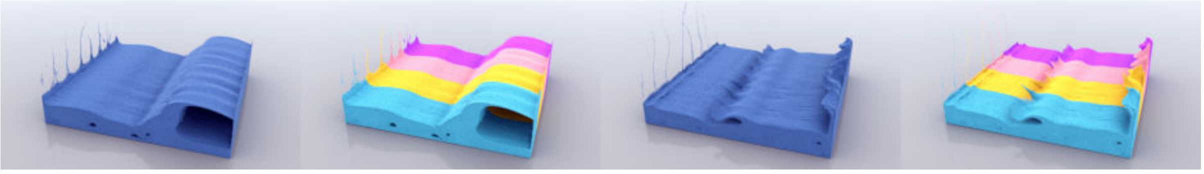


Figure 3.12: **Single dam-break with *MGSS*.** We simulate a fluid dam-break with 48M particles and the *MGSS* partition strategy. Each color denotes particles running on a single physical GPU device. With each device assigned approximately the same size spatial domain, all particles are evenly partitioned to 4 GPUs.

strategy used for a single matrix-vector product to improve performance. Moreover, our *AoSoA+G2P2G* design can benefit other hybrid particle-grid simulation methods, such as PIC/FLIP fluids, with better performance, more efficient memory usage, and flexible extension to multi-GPUs.

3.6 Benchmarks and Performance Evaluations

In this section, the *fixed corotated* constitutive model (Stomakhin et al., 2012a) is applied by default for all benchmarks unless stated otherwise. We use microseconds as units for all timings. The codebase used to generate these examples is made publicly available.

For experiments with different materials, we experimented with multiple parameters, which are set as easy-to-set compile-time constants. For instance, the *crashing concrete* scene is tested with Young's modulus ranging from $6e6$ to $6e8$ for grid resolution $512 \times 512 \times 512$ and $1024 \times 1024 \times 1024$, since concrete is really a mixture of materials with no absolute stiffness. We choose one of the material parameter settings we tested for the final results and list them in Table 3.5 for reproduction purposes.

In the following subsections, we start with the single-GPU performance comparison against the state-of-the-art methods. Two ablation studies are presented to analyze the efficacy of the proposed *AoSoA+G2P2G* design. We then move to multi-GPU settings with discussions of scalability, comparisons of two partitioning strategies, and demonstrations of large-scale simulations.

	Quadro P6000			RTX2080		
	(Hu et al., 2019a)	(Hu et al., 2019a)*	Ours	(Hu et al., 2019a)	(Hu et al., 2019a)*	Ours
Dragons	4.3	5.0	2.0	3.0	3.5	1.5
Dragons*	2.6	3.0	1.3	1.8	2.0	0.9
Bomb Falling	6.4	6.7	3.4	4.2	4.4	2.5

Table 3.2: **Single-GPU performance comparison.** All candidate single-GPU MPM methods use the MLS-MPM transfer method in explicit time integration. The timing results are run on NVIDIA *RTX 2080* and *Quadro P6000* and gathered after objects hit the ground for better evaluation. In addition, the initial reordering is disabled in the (Hu et al., 2019a)* benchmark; and the *dragons** scene reduces particles per cell by half.

3.6.1 Single-GPU Performance

3.6.1.1 Speedup over State-of-the-art Methods

When comparing with the state-of-the-art method (Hu et al., 2019a), we apply the optimal settings listed by Hu et al. (2019a), *i.e.*, *AoS* for particles, and SPGrid for grid blocks. Moreover, we set up the following scenes for performance evaluations.

- **dragons**. 775196 particles, 256 × 256 × 256 grid.

- **bomb falling**. 984018 particles, 256 × 256 × 256 grid.

As shown in Table 3.2, our pipeline reaches around 2× speedup compared to the state-of-the-art approach of Hu et al. (2019a). Under a more fair setting with the initial sorting of particles in (Hu et al., 2019a) disabled, we further achieve a 2.5× speedup. Measured speedups show consistencies on NVIDIA GPUs for both gaming (RTX series) and computing (Quadro series) and for different generations.

We also compare the timing against an open-source, heavily optimized CPU-based MPM codebase (Fang et al., 2019) (a SIMD vectorized implementation provided by its authors). The experiment is conducted in an elastic sphere colliding scene with particle counts ranging from 5 to 40 million. On a workstation with an Intel 8086K CPU and a single Quadro P6000 GPU, our GPU MPM achieves 110 to 120 × per-time-step speedup, as summarized in Table 3.3.

Particles #	$5M$	$10M$	$15M$	$20M$	$25M$	$30M$	$35M$	$40M$
CPU Time	667.70	1442.90	2110.20	2949.60	3875.01	4671.09	5148.63	5925.24
GPU Time	5.97	11.91	18.22	27.38	32.30	38.54	43.67	50.15

Table 3.3: **Performance comparison between an SIMD implementation *vs* our GPU pipeline.** (in microseconds) CPU: Intel 8086K. GPU: Quadro P6000 GPU.

3.6.1.2 Ablation Studies

***G2P2G* Speedup** We implement (Hu et al., 2019a) with the proposed *G2P2G* kernel. As shown in Table 3.4, all of the test cases have achieved around 40% speedup, except for the *cube* case where the model is generated with uniform sampling rather than Poisson sampling. With perfectly balanced particle distribution in the *cube* case, the negative impact of redundant particle data access pattern in *P2G* and *G2P* pipelines is mitigated. Moreover, the *G2P2G* kernel may lessen the latency-hiding capability (Laine et al., 2013) compared to conventional separate transfer kernels (*i.e.*, *P2G* and *G2P*), limiting the performance gain in the *cube* case. In addition to improving performance, the proposed *G2P2G* pipeline also decreases the storage size required for each particle, making it more favorable for particle migrations in the multi-GPU pipeline.

AoSoA Speedup On top of the *G2P2G* pipeline, we further change the *AoS* in (Hu et al., 2019a) to our proposed *AoSoA* layout. As shown in Table 3.4, the combined improvements enhance the transfer kernel with around 3× speedup without introducing any additional overheads of the maintenance or the storage of particle data.

3.6.2 Multi-GPU Scalability

The scaling with multi-GPU devices is an essential aspect of evaluating the efficacy and robustness of the algorithm. Ideally, the performance should scale with the number of devices and remain robust when simulating scenes that have different patterns for the halo regions. We perform scaling benchmarks on a workstation with one *Intel Core i7-8086K* CPU, four *NVIDIA Quadro P6000* GPUs, and 64GB RAM assembled on a

	(Hu et al., 2019a)	*G2P2G*		*AoSoA+G2P2G*	
	ref time	time	speedup	time	speedup
Dragons (775,196)	3.98	2.91	1.37×	1.33	2.99×
Dragons (619,916)	3.18	2.3	1.38×	1.15	2.77×
Dragons (388,950)	2.04	1.47	1.39×	0.78	2.62×
Bomb Falling (3,193,038)	16.95	12.25	1.38×	7.00	2.42×
Cube (262,144)	0.99	1.10	0.9×	0.74	1.34×

Table 3.4: **Ablation study.** The first timing column is the sum of the timings of *P2G* and *G2P* kernels. The timing in the second timing column is measured by replacing *P2G* and *G2P* kernels with the proposed *G2P2G* kernel. The timing in the third timing column is measured by replacing the *AoS* layout with the proposed *AoSoA* layout on top of the *G2P2G* kernel. The speedup is calculated by comparing it with the reference time (Hu et al., 2019a). Both *Bomb Failing* and *Dragons* scenes use irregular geometries; all *Dragons* scenes have the very same geometry but are sampled with different numbers of particles per cell, and *Bomb Failing* scene is much denser in space. The *Cube* scene is a uniformly sampled cube with particles ordered. All timings are computed using an *NVIDIA RTX 2080* graphics card.

Z390 motherboard.

Weak Scaling We assign each GPU device with one giant cube containing 4,096,000 particles. All cubes are either arranged compactly or side-by-side. In the compact layout, each partition shares a certain amount of halo regions with partitions from all the other GPU devices. In the side-by-side layout, each partition is only in contact with at most two neighboring partitions. The weak scaling comparisons are shown in Figs. 3.13 and 3.14.

Strong Scaling Four cubes of the same size that contains 4,741,632 particles are used to form a long cuboid. The scene is evenly partitioned and assigned to multi-GPU devices. The strong scaling comparisons are shown in Fig. 3.15.

Results Taken together, the scaling results indicate that the *G2P2G* kernel, as the bottleneck of the algorithm, is scaling almost linearly when each GPU is saturated by enough computations. Additionally, our multi-GPU MPM pipeline scales almost perfectly with the increasing number of GPUs. The improved efficiency with respect to memory

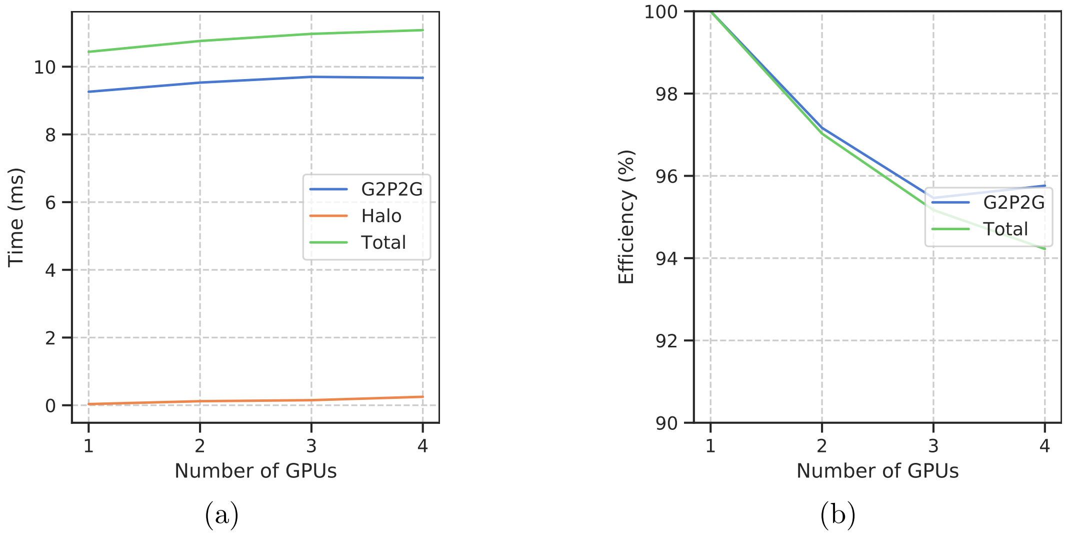


Figure 3.13: **Weak scaling; compact layout.** (a) The per time-step wall time remains steady with an increasing number of GPUs for the *G2P2G* and overall performances. The additional overhead of halo tagging is growing but remains insignificant. (b) Both the *G2P2G* kernel and the overall efficiencies still stay around 95% even when employing 4 GPU devices.

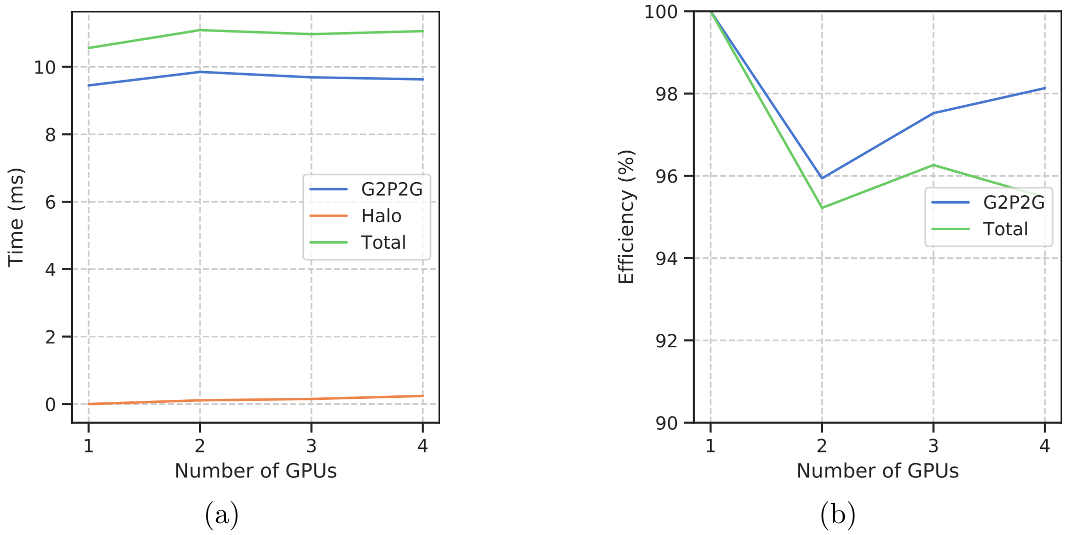
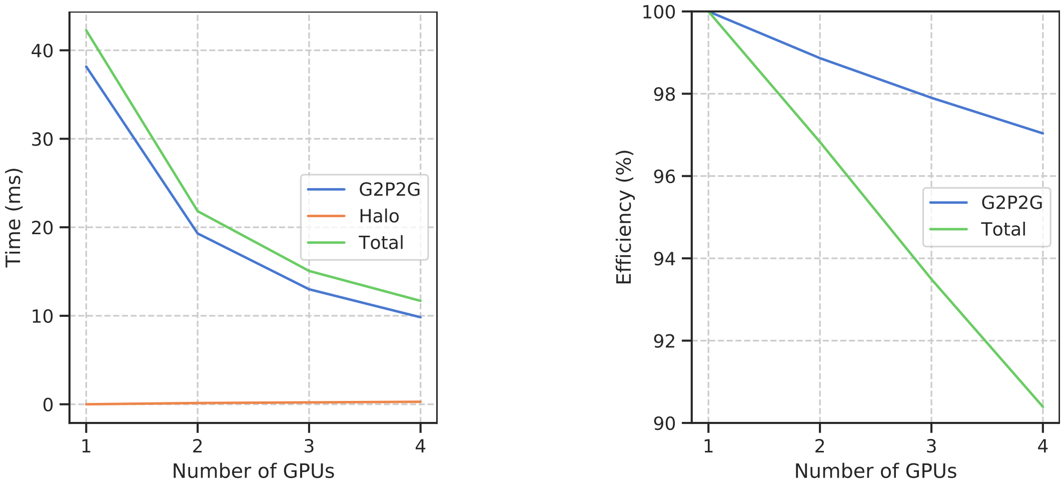


Figure 3.14: **Weak scaling; side-by-side layout.** (a) The per time-step wall time remains steady with an increasing number of GPUs for the *G2P2G* and overall performances. The additional overhead of halo tagging is growing but remains insignificant. (b) Both the *G2P2G* kernel and the overall efficiencies stay above 95% even when employing 4 GPU devices.



(a) (b)

Figure 3.15: **Strong scaling; side-by-side layout.** (a) Both *G2P2G* and overall performances scale well with an increased number of GPUs. The additional overhead of halo tagging is growing but remains insignificant. (b) The loss of the strong scaling of *G2P2G* is trivial even when employing 4 GPU devices, while the overall strong scaling drops to around 90%.

access and data communication (*e.g.*, fewer attributes stored, coalesced data accessing, and particle data locality) is also preserved in multi-GPU systems.

3.6.3 Partitioning Comparisons

Although *MGSP* is a perfectly balanced partitioning method in terms of the number of particles, the overhead due to halo block tagging would increase with more GPU devices employed. Moreover, when the size of the halo regions becomes large enough, the memory latency will increase and become the dominant factor compared to the latency of the *G2P2G* kernel. Such a performance degradation may frequently happen in fluid simulations where fluid may significantly mix together as time goes by, resulting in an increasing number of halo region storages and computations. In such cases, it would be more efficient to use *MGSS* where the halo region size stays the same throughout the simulation time; we demonstrate the partitioning using *MGSS* throughout a dam-break scene in Fig. 3.12.

3.6.4 Large-scale Simulations

We showcase a suite of simulations with various materials to demonstrate the scalability of our multi-GPU MPM algorithm. The following constitutive models with plasticity are implemented to demonstrate the applicability of our methods to diverse materials: 1) fixed corotated (Stomakhin et al., 2012a) to simulate elastic jello, 2) Non-Associated Cam-Clay (NACC) (Wolper et al., 2019) to reproduce soil and concrete, 3) Drucker-Prager elastoplasticity (Klár et al., 2016) for sand animation, and 4) weakly compressible fluid (Tampubolon et al., 2017) to generate water. All timings and spatial resolution settings are summarized in Table 3.5. Additionally, we also provide material related parameter settings for reproduction purposes.

We first demonstrate the scalability of the proposed multi-GPU MPM in Fig. 3.2, wherein 13,346 bombs fall onto the ground. This example is run on 8 GPUs, with the grid resolution $512 \times 2048 \times 512$, 134M particles, and each frame finished within 1 minute on average. To the best of our knowledge, no prior work has achieved such a large-scale simulation with MPM on with a single machine. In addition to the 8-GPU test, we also evaluate this scene on 4 GPUs with 6,688 bombs (67M particles). In a 4-GPU context, proper scaling is achieved with each frame simulated in 49.14 seconds on average.

Using the fixed corotated elastic material, we fill the bowl in Fig. 3.3 with 6,786 candies (23M particles) with each frame finished within 5 seconds on average. In other words, one only needs 20 minutes to obtain the results of a 200-frame simulation with 20M particles, which usually would take several days if only CPU-based MPM algorithms were adopted.

We crush concrete in Fig. 3.1 with NACC models, showing hydraulic press experiments on a concrete cylinder. The simulation domain is discretized into a $1024 \times 1024 \times 1024$ grid with $\Delta x = 1/1024$, while the concrete cylinder is represented by 93.8M particles. On a 4-GPU workstation, each frame is finished within 4 minutes. Note that only 4 (instead of 8) GPUs are employed to simulate 96M particles, indicating a strong potential of the proposed *AoSoA+G2P2G* in simulating large-scale scenes with limited memory resources.

Moreover, we further test the same scene with different settings of resolutions, particle numbers, and material parameters. Timing statistics show that it takes only 17 seconds to simulate the same scene with 12M particles and grid resolution $512 \times 512 \times 512$.

As another NACC example, three soil chunks fall, fracture, and mix together in Fig. 3.6; each frame with 52M particles is finished under 1 minute. In comparison, as reported by Wolper et al. (2019), a NACC example with only 1.67M particles consumes at most 10 minutes on a CPU-based MPM implementation. Similar to Fig. 3.1, we visualized the NACC-α to indicate the crack propagation.

Sand material is used to create two armadillos smashing together with fine details captured in Fig. 3.11. This scene has 55.5M particles with grid resolution $512 \times 512 \times 1024$. Simulating each frame takes less than 30 seconds using the proposed multi-GPU MPM pipeline.

We demonstrate a large-scale fluid simulation with the *MGSS* strategy in a *single dam-break* experiment, shown in Fig. 3.12. The topology of the fluid changes substantially as the simulation evolves, resulting in different portions of the fluid to mix together as time goes by. The size of the halo region would increase substantially as the simulation proceeds should we utilize the *MGSP* strategy; it would lead to significant performance degradation as most of the run-time would be spent in inter-GPU communication. In contrast, with the *MGSS* strategy, even though different portions of fluid are permeating into each other, the multi-GPU partitions are still relatively well balanced with a fixed-size halo region.

3.7 Limitation and Future Work

Limitation Our *G2P2G* kernel inherently requires a double buffer strategy for simultaneous read and write of particle and grid data. This fact could offset some of the savings of memory from the per-particle storage size. Although we use compact storage for particle attributes, their indices are still managed in the corresponding buckets that are pre-allocated with a uniform and conservative size. This design imposes restrictions

Example	**Particle#**	**Ave s/frame**	$\mathbf{\Delta t}_{\text{frame}}$	**GPU#**	**Grid Resolution**	max $\mathbf{\Delta t}_{\text{step}}$	$\mathbf{\Delta x}$	**Material Parameters**
(Fig. 3.2) Bomb Falling	134,007,186	59.56	1/48	8	$512 \times 2048 \times 512$	2.71×10^{-5}	1/256	Fix-corotated: $(100, 3 \times 10^5, 0.2)$
(Fig. 3.3) Candy Bowl	22,900,536	4.15	1/48	4	$1024 \times 1024 \times 512$	2.10×10^{-4}	1/256	Fix-corotated: $(100, 3 \times 10^3, 0.2)$
(Fig. 3.1) Crushing Concrete	93,790,217	236.89	1/240	4	$1024 \times 1024 \times 1024$	1.74×10^{-6}	1/1024	NACC: $(2240, 6 \times 10^8, 0.2, -0.01, 0.5, 0.8, 1.85)$
(Fig. 3.6) Soil Falling	52,904,854	57.38	1/48	4	$512 \times 512 \times 512$	1.65×10^{-5}	1/256	NACC: $(2, 3 \times 10^4, 0.3, -0.006, 0.3, 0.5, 1.85)$
(Fig. 3.11) Sand Armadillo	55,508,474	34.39	1/48	4	$512 \times 512 \times 1024$	3.58×10^{-5}	1/512	Sand: $(20, 1 \times 10^4, 0.4, 30, 0)$
(Fig. 3.12) Single Dam-break	48,608,497	15.17	1/240	4	$512 \times 2048 \times 512$	1×10^{-5}	1/256	Fluid: $(1000, 4 \times 10^4, 7.15)$

Table 3.5: **Parameters and timings.** We summarize the parameters of particle numbers, grid resolutions, Δx, the average time per frame, and the maximum Δt for various experiments described in Section 3.6.4. These examples are simulated with different materials; material-related information is recorded in the last two columns. Specifically, FC denotes the fixed corotated material, NACC for Non-Associated Cam-Clay, and Sand for the Drucker-Prager elastoplasticity. In addition to the basic settings of the material (density ρ, Youngs Modulus E, and Poisson Ratio ν), we also include other material-specific parameters. The material parameters are listed in the following order: 1) FC: (ρ, E, ν), 2) NACC: $(\rho, E, \nu, \alpha_0, \beta, \xi, M)$, 3) Sand: (ρ, E, ν, fa, co), and 4) Fluid: (ρ, k, γ). We recommend review the corresponding papers for further information about parameters.

on more irregular MPM simulations where the number of particles per cell is significantly larger.

Future Work For simplicity, we adopt the "pre-allocation for all" strategy for all spatial data structures specified in our codebase due to the lack of a dedicated allocator. A more customized allocator could provide more flexibility in terms of memory management, *e.g.*, on-demand allocation. There is also room for improvement in terms of robustness. We will work on an adaptive and unified framework that supports multi-material simulations, including both solids and fluids, and more flexible load balancing by allowing for dynamic re-partitioning of the whole domain, which would change the method of halo-region identification and memory preservation for halo particles. Deploying to distributed systems, *e.g.*, cloud or multi-GPU clusters, is another challenging yet promising direction worth of research efforts.

In the robotics community, we recently observe a growing amount of work that exploits physics-based simulation to facilitate robot learning in navigation (Xie et al., 2019), embodiment mapping (Liu et al., 2019a), soft robot locomotion (Hu et al., 2019b), tool-using (Zhu et al., 2015), inferring human utility (Zhu et al., 2016), and causality (Edmonds et al., 2020). These tasks are traditionally considered to be extremely challenging. With the capability to run large-scale simulations on multi-GPUs with a relatively short simulation time, we expect the robot learning community would start to adopt high fidelity simulations to enable robots acquiring knowledge and skills swiftly with minimal human intervention or supervision.

CHAPTER 4

A Sparse Distributed Gigantic Resolution Material Point Method

We present a four-layer distributed simulation system and its adaptation to the Material Point Method (MPM). The system is built upon a performance portable `C++` programming model targeting major High-Performance-Computing (HPC) platforms. A key ingredient of our system is a hierarchical block-tile-cell sparse grid data structure that is distributable to an arbitrary number of Message Passing Interface (MPI) ranks. We additionally propose strategies for efficient dynamic load balance optimization to maximize the efficiency of MPI tasks. Our simulation pipeline can easily switch among backend programming models, including OpenMP and CUDA, and can be effortlessly dispatched onto supercomputers and the cloud. Finally, we construct benchmark experiments and ablation studies on supercomputers and consumer workstations in a local network to evaluate the scalability and load balancing criteria. We demonstrate massively parallel, highly scalable, and gigascale resolution MPM simulations of up to 1.01 billion particles for less than 323.25 seconds per frame with 8 OpenSSH-connected workstations.

4.1 Introduction

High-resolution simulations are of high demand in both the VFX industry and scientific research. In recent years, the Material Point Method (MPM), due to its flexibility and versatility, has shown a great potential for modeling a wide range of continuum materials.

To reduce computational cost and programming efforts, researchers have explored modern computational platforms and improved MPM in both parallelization schemes

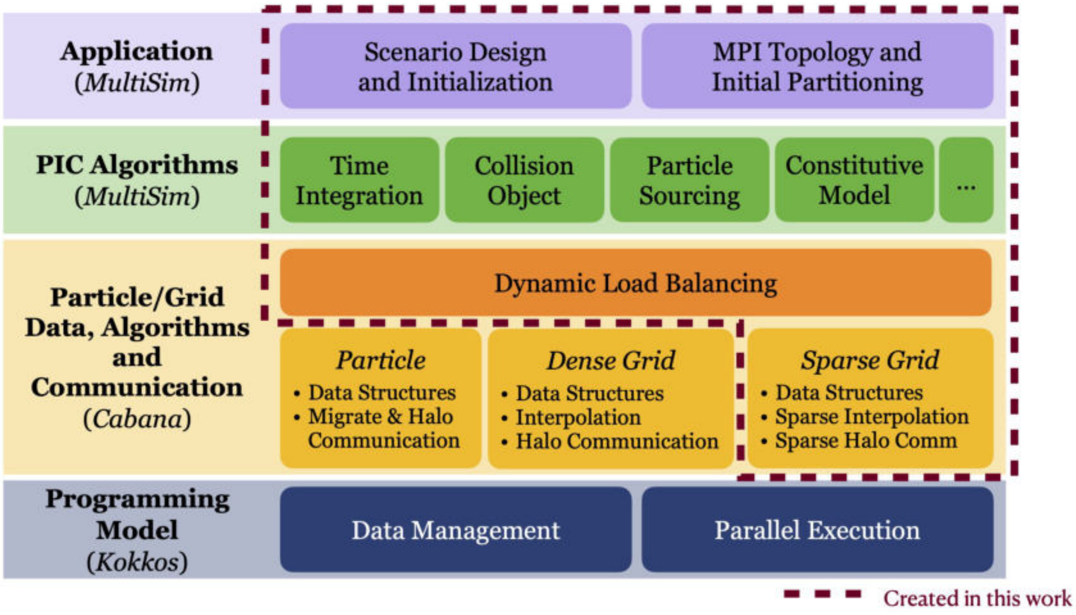


Figure 4.1: **Hierarchical System Architecture.** Our distributed MPM simulation system is designed and implemented hierarchically. In the chapter, we will use *Particle/Grid data layer* in short for the *Particle/Grid Data, Algorithms and Communication layer*. Different layers are implemented in separate codebases. Library names are labeled below layer names.

and latent particle/grid data management. Dedicated code design examples in graphics include threaded CPU MPM (Fang et al., 2018), single-GPU MPM (Gao et al., 2018b; Hu et al., 2019a, 2021) and multiple-GPU implementations (Wang et al., 2020; Fei et al., 2021). These state-of-the-art solvers still focus on exploiting a single machine with limited memory and computing power, leading to restrictions from several perspectives. On one hand, CPU-based computation is less efficient due to the limited number of threads despite the hundred-GB memory to support large-scale data. GPU-based computation, on the other hand, can significantly reduce the simulation time, but the onboard memory makes it challenging to go large-scale. While using additional GPUs can relieve the intense memory usage (Wang et al., 2020; Fei et al., 2021), the number of GPUs that a single motherboard can hold is still capped. Furthermore, both CPU and GPU MPM require skillful programming and dedicated design efforts.

Together, these restrictions motivate our exploration of a device-portable distributed simulation system, which allows researchers with minimal software experience to customize large-scale simulations and maximally leverage their devices. Specifically, we aim to build a distributed MPM system that pursues the following design goals:

- **Device portability for high performance.** Many existing simulations using only CPU or GPU resources require dedicated design, implementation, and optimization of code. It's also challenging to perform device-related code migration if new needs arise. Our design goal is to support effortless hardware switching according to users' needs, *i.e.*, to allow switching the latent programming models and parallel platforms for the simulations by modifying very few lines of code.

- **Distributed dispatch for large-scale simulation.** We allow the simulation system to scale up according to available hardware. To achieve this goal, we need to establish reliable and efficient grid/particle data structures and build communication machinery among multiple separate-memory computing nodes. To further improve the scalability, we aim to reduce unnecessary memory usage by developing new sparse data structures.

- **Dynamic workload decomposition.** Distributing computations to multiple workers is challenging from two standpoints. First, from the performance perspective, calculation time is bounded by the device with the highest workload. While other nodes are busy, idle nodes with tasks completed earlier simply wait, wasting time and resources. Second, robustness and system stability are crucial. An imbalanced partitioning strategy may cause run-time failure by exhausting the memory of some overloaded node. In simulations, the topology of the activated grids and the particles can dramatically differ from their initial settings. Thus, static partitioning can become extremely ineffective and non-robust, working only for carefully designed scenes as in (Wang et al., 2020; Fei et al., 2021). Therefore, we demand dynamic workload partitioning for better distributed performance and robustness.

- **Programming simplicity.** A typical parallel simulation code requires great programming effort in memory management and parallel execution. We prefer the system's users with different simulation and programming skill levels can all focus

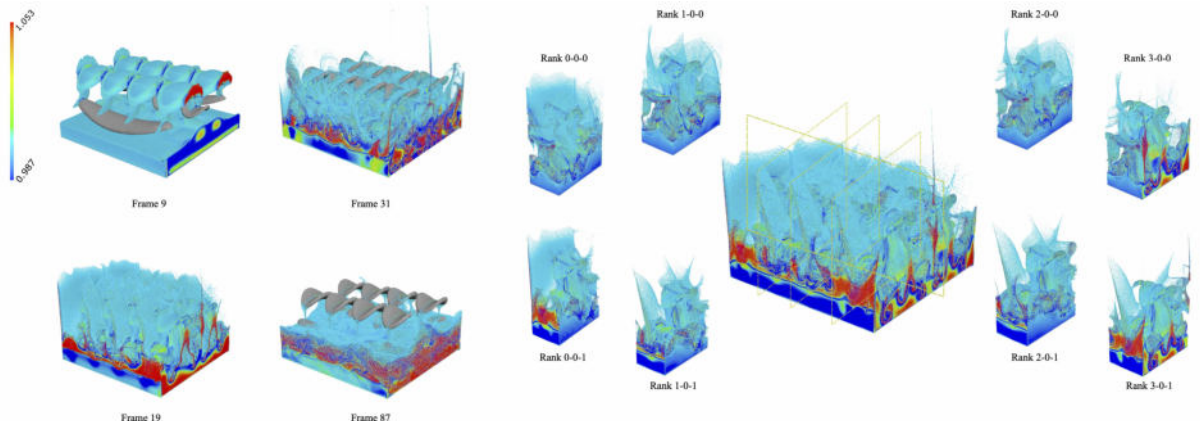


Figure 4.2: **1B-Fluid** with more than **1.01B** particles. We use 4 × 1 × 2 MPI ranks (8 in total) to handle the simulation domain. The weakly compressible fluid particles are colored by the volume change ratio. We show representative frames (left) and sub-domains handled by eight workers (right).

on their primary goals, ranging from setting up scenes and designing numerical algorithms to exploring novel data structures.

4.1.1 Key Insight

These assumptions and design goals lead us to a hierarchical architectural design principle as shown in Figure 4.1. We divide the whole system into four layers: the *programming model layer*, the *particle/grid data layer*, the *PIC algorithm layer*, and the *application layer*. This hierarchical design allows users with various experimental goals to focus on distinct layers and extrapolate the system's potential. Below, we discuss each layer in more detail.

Programming Model. This bottom layer focuses on developing device-portable specializations on 1) memory allocation and access and 2) parallel execution operations. Specifically, it allows upper layers to use unified interfaces to perform parallel computations with the desired backend computational models (*e.g.*, OpenMP or CUDA) and manipulate data stored on user-preferred computing devices (*e.g.*, CPU or GPU).

Particle/Grid Data. Generally, Lagrangian particles, Eulerian grids, and/or their combinations are used for the simulation schemes considered in this work. However, designing and implementing these data structures and related algorithms on distributed systems require intensive effort. Thus, we use an independent layer to implement particle/grid distributed data structures that allow users to customize the latent memory layout and the attributes stored for each element. Furthermore, particle and grid inter-rank communications are integrated for distributed systems. In addition, since particle/grid number determines the total workload on each rank, we also attach dynamic load balancing as another crucial component in this layer.

PIC Algorithm. Simulating dynamic physical systems typically requires a time integration scheme. In our case for example, MPM adopts a Particle-In-Cell (PIC) paradigm. Users can switch to other schemes by modifying the integration strategy. Additional components in this layer include constitutive models for material versatility and a particle sourcing module for time-dependent particle injection.

Application. Users can customize the scene setup and material parameters inside this layer based on all lower-layer components. The user also chooses an MPI topology according to the host hardware.

4.1.2 Background

To avoid reinventing the wheel, we employ two libraries, `Kokkos` (Edwards et al., 2014; Trott et al., 2022) and `Cabana` (Slattery et al., 2022; Mniszewski et al., 2021) to satisfy part of the requirements of the bottom two layers.

`Kokkos` provides support for basic data structures on all major heterogeneous and high-performance computing architectures (Edwards et al., 2014; Trott et al., 2022). Users can allocate multidimensional arrays on different computing devices such as CPUs and GPUs in a relatively easy and unified manner. In addition, `Kokkos` contains abstractions for most general parallel execution patterns that are portable across hardware. `Kokkos` fully satisfies the design goal of our *programming model layer*, enabling adoptions on modern hardware including NVIDIA GPUs and multi-core CPUs which are both used in

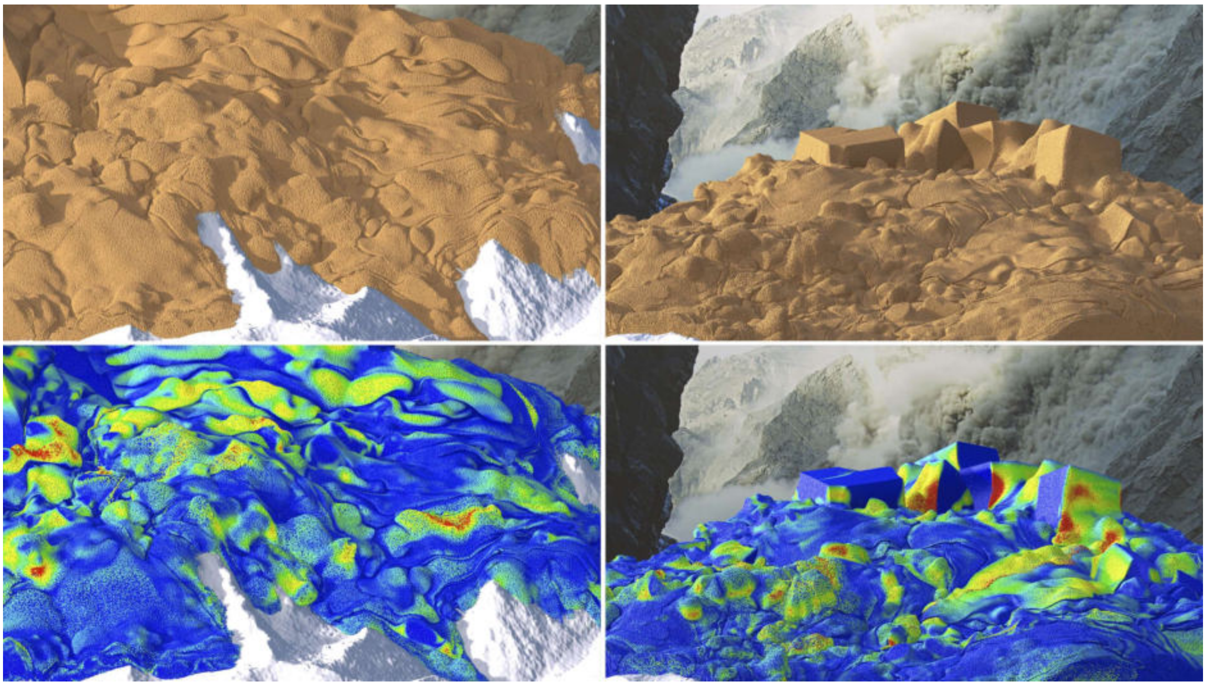


Figure 4.3: **Mudflow.** Our distributed MPM enables this over-207.8M-particle mudflow simulation using averagely only 159.34 seconds per frame. Here we employ four workstations connected through OpenSSH.

this work.

`Cabana` is a particle-specific library based on `Kokkos`. It provides particle data structures, particle algorithms, and MPI communication operations. It also supports dense-grid and dense-particle-grid operations with a static partitioning. Thus, `Cabana` satisfies the particle-related requirements in our *particle/grid data layer*; however, we require extra components for the grid components. First, the dense grid is not suitable for simulations with significant empty space since a large amount of memory and resources would be wasted. Second, static partitioning limits the performance and scalability as analyzed in the design goals.

4.1.3 Contributions

Following the hierarchical approach above, we develop a distributed simulation framework specialized for MPM kernels, emphasizing scalability and performance portability. Our system is built on top of a modern C++ programming model (`Kokkos`) and allows users to write and dispatch performant code on HPC platforms with CPU- and GPU-based parallelization. In order to support the generality for users to switch back-end devices effortlessly, we do not pursue extensive performance improvement on problems that can be well-solved by dedicated-designed single-rank CPU or GPU devices, as did Klár et al. (2017); Gao et al. (2018b); Wang et al. (2020); Fei et al. (2021). Instead, we concentrate on properly resolving large-scale scenarios where inter-communication is unavoidable and single-rank machines are unable to handle.

In addition, for the *particle/grid data layer*, we utilize `Cabana` for particle-related operations. We extend the `Cabana` library by designing and implementing a novel distributed sparse grid data structure with highly efficient allocation, access, and communication algorithms. Furthermore, we customize a dynamic load balancing partitioner to improve the simulation performance by ensuring a balanced workload distribution on all MPI ranks. Based on these implementations, we develop a fully open-source simulation library that supports multiple MPM-related algorithms and application designs, leading to gigascale resolution simulations for a wide range of solid and fluid materials. We further provide comprehensive computational experiments that demonstrate

- the scalability of the proposed distributed system,
- the benefits of dynamic load balancing on sparse simulations,
- the performance variance with different MPI topologies.

In addition, we demonstrate large-scale simulation examples for designers to customize their scenes with versatile application-level components.

4.1.4 Overview

Following the hierarchy proposed in §4.1.1, in this chapter, we introduce each system layer in different subsections. First, in §4.3, we overview the background needed to understand this chapter and corresponding implementations, including an introduction of MPM (§4.3.1), the `Kokkos` programming model (§4.3.2), and `Cabana` particle-related

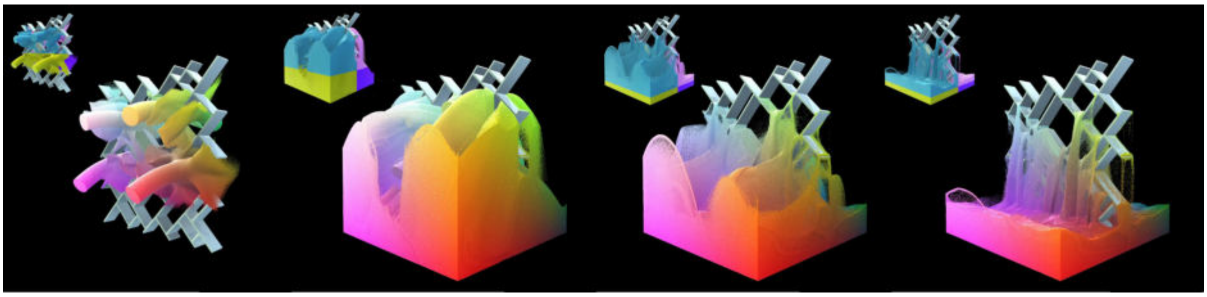


Figure 4.4: **High-resolution Sand Injection** with grid resolution 512 × 512 × 512 and 266.5M particles. Sand particles are rainbow-colored by their positions. We use four MPI ranks (2 × 2 × 1) for computation and show the partition status on the left-top corner of each sub-figure.

implementations (§4.3.3). This section thus covers the *programming model layer* and part of the *particle/grid data layer*. Next, we introduce two new features we integrated into the *particle/grid data layer*: §4.4 presents the proposed MPI-dedicated distributed sparse grid and §4.5 shows our distributed dynamic load balancing scheme. After that, we describe additional details related to the *PIC algorithm* and *application* layers in §4.6. We then offer performance analysis in §4.7: 1) weak and 2) strong scaling of the proposed distributed MPM scheme, 3) the performance improvement with our distributed dynamic load balancing algorithm, 4) performance comparison with different MPI rank topologies, and 5) large-scale simulation results with up to 1B particles. Finally, we conclude this chapter with limitations, discussion, and possible future work.

4.2 Related Work

4.2.1 HPC-oriented Simulation Programming Model

Modern hardware makes it possible to improve simulation performance with dedicated data structure and parallel kernels. One primary attempt is to use multiple CPU cores with tools like OpenMP (Dagum and Menon, 1998) and Intel TBB (Willhalm and Popovici, 2008). Further explorations are built upon GPUs for faster computations. For example, GPU-based schemes were designed for Eulerian and Lagrangian fluids (Chentanez and Müller, 2011, 2013; Cohen et al., 2010; Pfaff et al., 2010; Goswami et al., 2010; Vantzos et al., 2018; Winchenbach et al., 2016; Amada et al., 2004), as well as for hybrid solvers (Gao et al., 2018b; Hu et al., 2019a, 2021; Wu et al., 2018; Chentanez et al., 2015). Multi-GPU platforms (Wang et al., 2020; Fei et al., 2021) were developed for MPM as well.

For scalability, researches such as (Shah et al., 2018a; Qu et al., 2020; Liu et al., 2016; Bauer et al., 2012) also explored algorithms and data structures for distributed simulations. Kale and Krishnan (1993) introduced `Charm++`, an object-oriented portable C++-based parallel programming language that is still being actively maintained by researchers from multiple fields. Additionally, supportive systems such as Canary (Qu et al., 2018) and Nimbus (Mashayekhi et al., 2017, 2018) distribute tasks onto computing nodes. For most systems, MPI (Snir et al., 1998) is adopted as the message communication library. It provides various communication primitives for sending and receiving data among ranks. For example, Lesser et al. (2022) propose a multi-physics framework named *Loki*, which can be used as a generalized tool to simulate various material phenomena ranging from elastic solids to fluids with multi-CPU-core clusters. However, *Loki* leaves GPU usage as future work. Similarly, for other systems, whether single-machine-based or distributed, data arrangement and computations are limited to specific back-end devices, and the performance optimization is only architecture-oriented.

There have been many efforts for performance portability, *i.e.*, enabling high performance across different architectures with a single source code. For example, Hu et al. (2019a) developed a compiler that allows users to switch CPU/GPU backend by changing a single line of code. Medina et al. (2014) provided a unified API for interacting with backend devices with a C-extended kernel language. Additionally, Zenker et al. (2016) implemented an abstract hierarchical redundant parallelism model that supports applications on many hardware types ranging from multi-core CPUs to GPUs. Furthermore, libraries such as `Kokkos` (Edwards et al., 2014; Trott et al., 2022) with its extensions like `Cabana` (Slattery et al., 2022; Mniszewski et al., 2021) support the manipulation of array-based data structures and their corresponding parallel patterns on multiple underlying computing devices in a distributed manner. These libraries relieve researchers' effort in backend-oriented maintenance and their usage is becoming a trend for next generation high-performance simulations.

4.2.2 Sparse Grid Data Structures

In many Eulerian and hybrid simulations, the grids are sparsely activated, *i.e.*, only part of the grids contains non-zero entries. Thus, sparse grid data structures have been developed to improve memory bandwidth and data access efficiency. For instance, OpenVDB (Museth, 2013), sparse paged grids (Setaluri et al., 2014), and Bifrost's volume tools (Bojsen-Hansen et al., 2021) enable efficient interactions of time-varying sparse quantities over large grid with dedicated grid representation design on CPUs. Furthermore, Museth (2021); Hoetzlein (2016) and Gao et al. (2018b) broaden the sparsity idea of VDB and sparse paged grids to GPUs. These extensions vastly improve the simulation scalability and efficiency on NVIDIA GPUs with limit-sized RAMs. Moreover, developing a data hierarchy is an important addition to improve sparse data access efficiency, such as in (Hu et al., 2019a; Liu et al., 2018).

4.2.3 Load Balancing for Simulations

Load balancing and workload distribution are crucial for the performance of distributed systems. Traditional load balancing algorithms perform either geometric-based (Berger and Bokhari, 1987) or graph-based (Karypis and Kumar, 1997; Catalyurek et al., 2007) optimization. Some other works consider the temporal aspect when deciding partition boundaries. Shah et al. (2018a) proposed speculative balancing for fluid simulation. It computes partition-to-worker assignments by performing a low-resolution simulation substitution and predicting the high-resolution workload distribution in the upcoming steps. Their partitioning overhead is polynomial in the number of ranks. Additionally, Qu et al. (2020) proposed a birdshot scheduling method for partitioning. It splits the simulation domain into many micro-partitions and assigns them to nodes randomly. Based on cloud computing nodes' high latency, high throughput, and full bisection bandwidth, birdshot scheduling was shown to outperform static partitioning in many fluid simulation schemes including SPH, Eulerian and hybrid methods.

4.2.4 Fast MPM in Computer Graphics

MPM was introduced to graphics by Stomakhin et al. (2013) for simulating snow dynamics. Scaling MPM to higher resolution is promising since a regular Cartesian grid is used to discretize fields (Jiang et al., 2016). Many research efforts investigated techniques to accelerate MPM. For example, Klár et al. (2017) constructed production-ready GPU MPM solvers in the Dreamworks animation pipeline with adaptive particle advection. Gao et al. (2018b) studied design choices for explicit and implicit MPM parallelism utilizing GPU. Based on that, Wang et al. (2020) harnessed the power of multiple GPUs and achieved one-hundred-million-particle simulations on an eight-GPU workstation. Recently, Fei et al. (2021) summarized various principles for accelerating single- and multi-GPU MPM implementations. They achieved real-time performance for a one-million-particle simulation on four NVIDIA GPUs with NVLinks.

Taking a different path towards performance optimization, Hu et al. (2019a) proposed the Taichi programming language as a high-level interface to process spatially sparse multi-level data structures. By decoupling data structures from computations, users can perform experiments using different data structures without changing much code. Hu et al. (2021) further improved this compiler by introducing low-precision numerical data types for reduced memory occupation and bandwidth consumption. It enabled faster and higher-scale simulations by sacrificing numerical accuracy.

4.3 Background

4.3.1 Material Point Method (MPM)

MPM is a hybrid simulation method that uses particle and grid representations to discretize the simulation domain. Typically, physical attributes including mass (m_p),

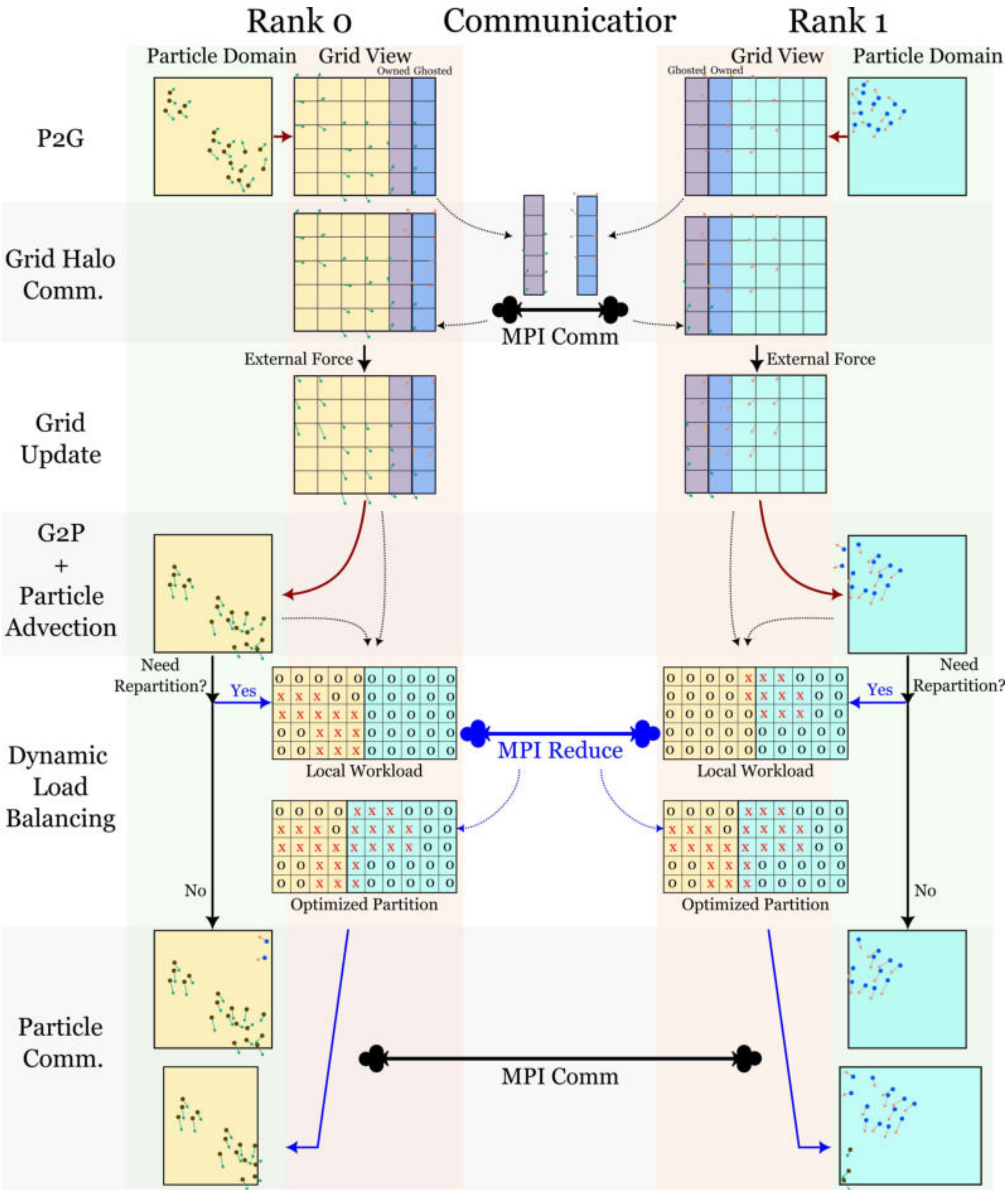


Figure 4.5: Our distributed MPM simulation pipeline, using 2 ranks as an illustrative example. In the figure, Comm. is short for Communication.

velocity (v_p), deformation gradient (F_p), and affine velocities (C_p) are stored on particles; grid nodes that stores mass (m_i) and momentum ($m_i v_i$), transferred from particles, are treated as auxiliary scratchpad variables to perform spatial derivative computations and boundary condition enforcement.

To demonstrate our programming model without loss of generality, we implement the most basic first-order MPM time integration scheme with the following essential steps for incremental dynamics.

1. **Particles-to-Grid (P2G)**.
   Compute grid mass and momentum from particles: $\{m_p, m_p \boldsymbol{v}_p^n\} \rightarrow \{m_i, m_i \boldsymbol{v}_i^n\}$. In addition, transfer force contributions to grid nodes from elastic stresses of the nearby particles and project particle deformation gradients for plasticity (if any).

2. **Grid Update**.
   Update grid velocities with either explicit or implicit time integration: $\boldsymbol{v}_i^n \rightarrow \boldsymbol{v}_i^{n+1}$, taking boundary conditions and collision objects into account.

3. **Grid-to-Particles and Particle Advection (G2P)**.
   Transfer velocities from grid nodes to particles, evolve particle strains, and then update particle positions with their new velocities: $\{\boldsymbol{v}_i^{n+1}\} \rightarrow \{\boldsymbol{v}_p^{n+1}, \boldsymbol{F}_p^{n+1}\}$, $\{\boldsymbol{p}_p^n, \boldsymbol{v}_p^{n+1}\} \rightarrow \{\boldsymbol{p}_p^{n+1}\}$.

These three steps are the major computing components in MPM. We show how these computations are performed on each MPI rank in our distributed system in Figure 4.5.

4.3.2 Performance Portable Parallel Programming with `Kokkos`

As introduced in §4.1.2, we employ the `Kokkos` library (Edwards et al., 2014; Trott et al., 2022) as the device-portable *programming model layer* that supports multidimensional array allocation and access and parallel execution patterns. Using `Kokkos`, our simulation pipeline can switch among different backend programming models, including OpenMP and CUDA, using C++ template arguments. For example, we can make the following definitions and pass them into both particle and grid data structures and related parallel kernels to invoke an NVIDIA GPU for data management and computation. The comments show how we can quickly switch to CPU with OpenMP.

```
using EXECSPACE = Kokkos::Cuda; // Kokkos::OpenMP
using MEMSPACE = Kokkos::CudaSpace; // Kokkos::HostSpace
using DEVICE = Kokkos::Device<EXECSPACE, MEMSPACE>;
```

The particle data structure, our new sparse grid, and all other supporting arrays are implemented based on `Kokkos::View`, which defines a multidimensional array based on user-specified memory space. We can set the array size at compile time or run time. One example of defining a 2-dimensional array with `Kokkos::View` is listed in the first line of the code patch below. It defines an NUM × 3 array, with the first dimension size (NUM) specified during run time and the second during compile time.

To dispatch parallel computations, one can use various `Kokkos` parallel patterns with a specified execution policy to perform defined kernels on different architectures, as shown below. In line 2, we get particle position data slices (detailed particle definitions are listed in §4.3.3), and then dispatch a `Kokkos::parallel_for` pattern with a range policy to assign particle positions to the pre-defined `Kokkos::View`. By changing the content of `KOKKOS_LAMBDA`, one can easily modify the behavior of the computing kernel. Finally, in line 12, we create a host copy of the `View` data so that the CPU-side (host-side) code can also access or further output the data to files for visualization.

```
1  Kokkos::View<T*[3], MEMSPACE> pos( "positions", NUM );
2  auto x_p = Cabana::slice<P::pos>( particles );
3  /* dispatch a parallel for to assign data from x_p to pos */
4  Kokkos::parallel_for(
5      Kokkos::RangePolicy<EXECSPACE>( 0, particle_num ),
6      KOKKOS_LAMBDA( const int idx ) {      // compute kernel
7          pos( idx, 0 ) = x_p( idx, 0 );    // element access
8          pos( idx, 1 ) = x_p( idx, 1 );
9          pos( idx, 2 ) = x_p( idx, 2 );
10     } );
11 Kokkos::fence();              // fence execution space
12 auto host_view =
13     Kokkos::create_mirror_view( Kokkos::HostSpace(), pos );
```

In addition to `Kokkos::parallel_for`, other parallel execution patters are supported in `Kokkos`, including `parallel_reduce` and `parallel_scan`. We refer to Trott et al. (2022);

Edwards et al. (2014) for further details.

4.3.3 Distributed Particles with `Cabana`

With the device-portable programming model, we are able to build the *particle/grid data layer*. As mentioned in §4.3, we utilize `Cabana` library for particle memory management and communication.

Built upon the `Kokkos::View`, the `Cabana::AoSoA` enables *Array-of-Structure-of-Array* (*AoSoA*) layout (Wang et al., 2020) to manage particle storage with user-specified properties. The *AoSoA* structure exploits the advantages of both *Structure-of-Array* and *Array-of-Structure* to conserve both coalesced threads calculations and performant random memory access patterns when parallelizing MPM. The following code sample shows how to declare particle storage with MPM-essential properties such as mass, position, velocity, deformation gradient, APIC transformation matrix, and plastic volumetric strain (lines 1-4). Additionally, lines 5-8 illustrate how to use `Cabana::slice` to access individual particle properties. Additional details are provided by Mniszewski et al. (2021).

```
1 using particle_members =
2     Cabana::MemberTypes<T, T[3], T[3], T[3][3], T[3][3], T>;
3 using particle_list = Cabana::AoSoA<particle_members, MEMSPACE>;
4 particle_list particles;
5 // access single particle properties with Cabana::slice
6 auto position = Cabana::slice<1>( particles );
7 auto velocity = Cabana::slice<2>( particles );
8 auto affine = Cabana::slice<4>( particles );
```

4.3.4 MPI Communication

Handling particle and grid data communication is another crucial ingredient of the *particle/grid data layer*. We use MPI (Snir et al., 1998), a message passing interface widely used for multi-node applications, to perform data communication among distinct ranks. In the rest of the chapter, we use MPI rank, rank, and worker as interchangeable terms of the logically independent computing unit that handles non-overlapped work.

In practice, we divide the whole simulation domain spatially and distribute the corre-

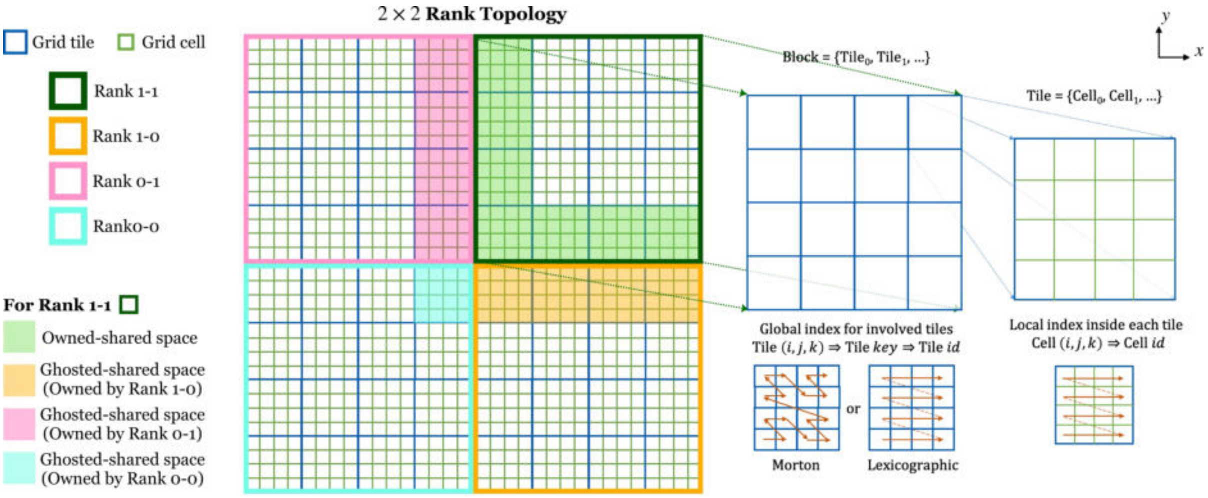

2 × 2 Rank Topology
Grid tile
Grid cell
Rank 1-1
Rank 1-0
Rank 0-1
Ranko-0
For Rank 1-1
Owned-shared space
Ghosted-shared space (Owned by Rank 1-0)
Ghosted-shared space (Owned by Rank 0-1)
Ghosted-shared space (Owned by Rank 0-0)
Block = {Tile$_0$, Tile$_1$, ...}
Tile = {Cell$_0$, Cell$_1$, ...}
y
x
Global index for involved tiles
Tile (i, j, k) ⇒ Tile *key* ⇒ Tile *id*
Local index inside each tile
Cell (i, j, k) ⇒ Cell *id*
or
Morton
Lexicographic

Figure 4.6: **Hierarchical Sparse Grid Representation**. We use a 2D MPI topology as an example. The entire simulation domain is divided into four *blocks*, each handled by a unique MPI rank. The *blocks* are further divided into *tiles*, which contains $N \times N$ grid *cells* (in this example, $N = 4$). Local cell indexing inside each *tile* is lexicographical. The tile ijks are mapped to 1D keys through a user-specified manner (either lexicographical or using a Morton curve). In addition, the halo regions, *i.e.*, the *shared spaces* of different MPI ranks, are classified as *owned-shared space* and *ghosted-shared spaces* as illustrated in shaded colors.

sponding workload (particles and grids) to ranks with a user-specified MPI communicator topology. MPI ranks can be mapped to a single or multiple computing devices according to the hardware setup and the options provided when running the simulation executable with `mpirun/mpiexec` command. Generally, one individual process will handle one rank during execution. In our system, we use non-blocking `MPI_Isend/MPI_Irecv` pairs for particle and grid data exchanges among workers. Also, `MPI_ALLReduce` with operators like `MPI_SUM` or `MPI_MIN` are employed for inter-rank grids/particles reductions. Moreover, `MPI_Barrier` is called for synchronization to ensure data consistency and computation correctness.

4.4 Distributed Sparse Grid

In MPM simulations, the valid domain is generally sparsely occupied by material particles. As a result, it may cause an unnecessary waste of computing time and memory occupation in large-scale simulations if a dense grid is used. Therefore, we develop a distributed sparse grid data structure to represent the sparsely populated uniform grids in the *particle/grid data layer* to more effectively leverage the computing resources on multiple MPI ranks. Our sparse-grid approach shares kernel-level interfaces with the dense grid data structures implemented in `Cabana`, making it effortless for `Cabana` users to switch in their simulation implementations.

We distribute the simulation work to multiple MPI ranks by dividing the entire domain into rectangular partitions. Each MPI rank needs to have panoramic information to guide its local computations. Some essential global knowledge includes the size and position of the entire simulation domain and the rectangle range each MPI rank handles. To clarify the descriptions, we propose the following concepts to represent the logical simulation domain and uniform grid. Each concept is implemented as a separate C++ class in practice.

- **Global Mesh:** The actual position and size of the entire simulation domain.
- **Global Grid:** The entire logical uniform grid, indexing from 0 to the grid resolution in each dimension. The global grid also contains the domain partition information, indicating the grid range that the current MPI rank is in charge of.
- **Local Mesh:** The position and rectangle sub-domain size of the current MPI rank.
- **Local Grid:** The valid *owned* and *shared* grid indexing space of the current worker. The *owned space* represents all the grids exclusively accessed by the rank, while the *shared space* indicates the halo range with which multiple MPI ranks may interact. As illustrated in Figure 4.6, we have two *shared space* types: 1) *owned-shared space* to represent all the grids that are owned and managed by the current MPI rank but may interact with particles residing on the neighbor ranks, and 2) *ghosted-shared space* to denote the grid range that owned by some other MPI ranks, but the current worker may read from or write to.

To further improve the flexibility of grid data management, we propose a hierarchical *block-tile-cell* representation of the simulation grid domain. *Block* is defined as the reference to the entire *local grid* domain on a single MPI rank. It is further divided into *tiles*, as shown in Figure 4.6, where each *tile* contains a user-defined number of *cells* ($4 \times 4 \times 4$ in our examples). This hierarchical design allows users to customize the grid data allocation and access with coalesced data access patterns that could potentially benefit parallel particle-grid interpolations. It can also fit the special design needs in user-customized simulation pipelines such as (Wang et al., 2020; Gao et al., 2018b).

Before performing the grid array allocation, we define a sparse *grid layout* to specify the following information:

1. The entity type on the sparse grid (*i.e.*, whether to store the value on grid nodes, cell centers, faces, or edges);
2. The valid grid *tiles* in the current simulation step;
3. The halo status in the current simulation step.

The first piece of information is consistent throughout the entire simulation process. In practice, we define multiple overloading functions in the *local grid* concept to deal with the minor indexing and grid ownership disparity caused by different entity types. By contrast, the second and third status varies along with the simulation and, thus, require recalculation in every time step. In the following subsections, we explain how the sparsity is registered (§4.4.1) and how the halo communications are achieved (§4.4.2).

4.4.1 Sparse Map

In MPM simulations, the valid/activated grids, *i.e.*, the grids that will be allocated and accessed in the upcoming step, are the grids that will interact with particles. The grid range each particle will activate is determined by the particle position and the Eulerian interpolating functions. We adopt the quadratic kernel for particle/grid data transfer in all examples. Thus, we can ensure that each particle will activate only 27 grid *cell*s nearby. Since we use grid *tile* as the minimum unit of actual allocation, each corresponding *tile* of these *cells* is mapped to an array index inside the grid memory by spatial hashing before MPM time integration in each step. We first map the global 3D *tile* index to a hashing key using either the lexicographical order or a space-filling Morton curve (Figure 4.6) (Setaluri et al., 2014; Gao et al., 2018b; Wang et al., 2020) according to user's choice. This process ensures that every logically independent grid *tile* has a unique identifier on whichever worker. Then the *tile* key is registered in a device portable hash table (`Kokkos::UnorderedMap`) in a specified execution manner. This way, the 3D indices of all valid *tiles* will be mapped to a linear memory span indexing from 0 to the total valid-*tile* number.

4.4.2 Sparse Halo

To decide whether two adjacent MPI ranks need to exchange grid data and how much to communicate, we need to consider the following factors:

- Entity type stored on the grids.
- Particle-grid interpolation kernel size.
- Whether the grid halo region contains valid *tiles*.
- Halo size.

Concretely, the first two factors correspond to how the MPI neighbor topology is defined by the entity type and the kernel size to transfer particle-grid data. Specifically, in 3D with kernel size 1, the workers would share data with all 26 neighbor ranks if data is stored on grid nodes or cell centers, while only six neighbors require communication for edge and face cases. Our MPM system stores all the attributes on grid nodes with a quadratic kernel, and thus each worker needs to communicate data with all topologically adjacent ranks.

The next two factors correspond to the following. Considering the sparsity of the grid data, halo communication happens only when there are commonly registered *tiles* in the *ghosted-shared spaces* and *owned-shared space* of two neighboring ranks. And the size of the *owned-* and *ghosted- shared space* is decided by the halo size. Under this circumstance, we introduce two types of halo communications:

- *Halo Scatter.* Scatter the data in the *ghosted-shared space* of the current MPI rank to their owner rank and perform the specified grid reduction (such as summation or computing the minimum/maximum value). Note that the reduction happens on the *owned-shared space* of the owner worker.
- *Halo Gather.* Gather grid attributes in the *ghosted-shared space* of the current worker from the *owned-shared space* of the neighboring owners.

The *halo scatter* happens after the *P2G* transfer in MPM time integration. The grid owner ranks collect and reduce all valid grid data during this process. Afterwards, all owner ranks will contain complete grid information transferred from simulation particles, including in *owned space* and *owned-shared space*s. Then *halo gather* is performed before *grid update* to ensure all MPI ranks hold the entire and correct grid data in *shared space*s.

To reduce the MPI communication overhead, we first count the valid *tiles* in the *ghosted-* and *owned-* shared space before *halo gather/scatter*, and broadcast the counting results to the neighbor ranks. For *halo scatter*, workers will send halo data to a specific neighbor only if both the counting in its *ghosted-shared space* and the counting in the neighbor's *owned-shared space* are non-zero. Additionally, a worker will wait to receive data from a neighbor only when the *owned-shared space* and the corresponding neighbor's *ghosted-shared space* are non-empty. Similar verification is also performed before actual data transfer in *halo gather* operation, with the role of *ghosted-* and *owned-* shared space switched.

4.4.3 Sparse Array Allocation

Based on the information provided by the *grid layout* (specifies entity type, grid activation, and sparse halo), the sparse *grid array* is created and allocated. The grid is managed in

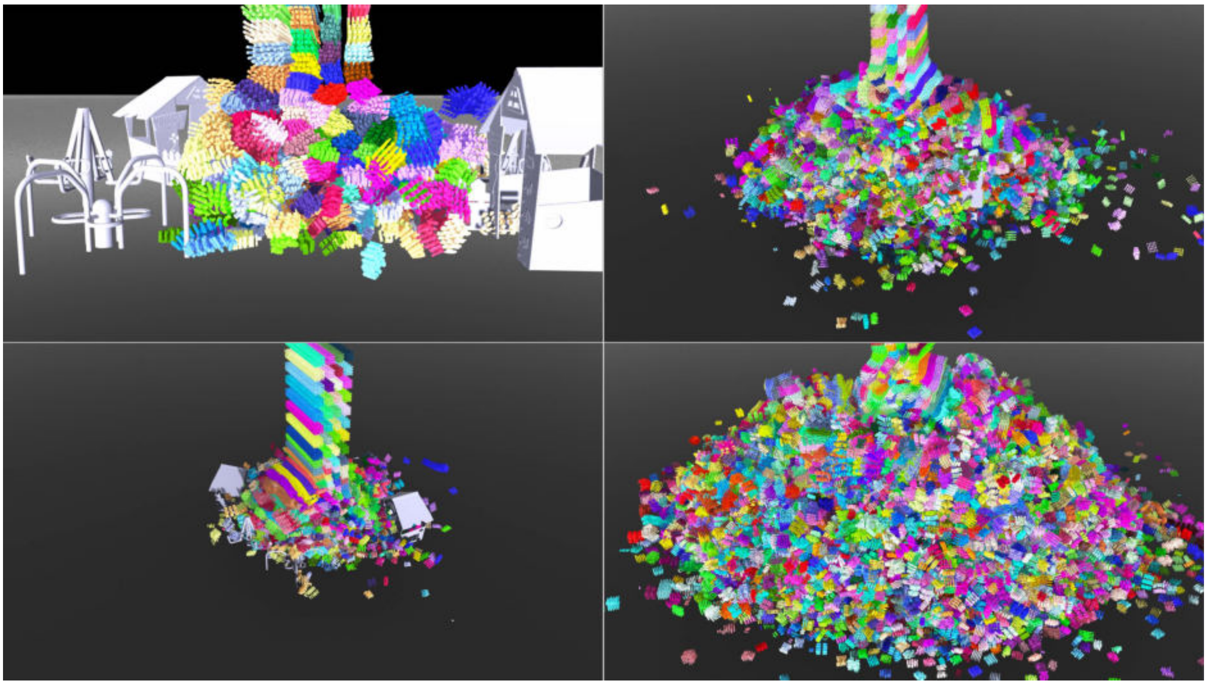


Figure 4.7: **Elastic Playground.** Numerous letters, symbols, and numbers are poured onto the toy playground. At most, 394M particles are involved, and the average simulation time is 229.56 seconds per frame.

an *AoSoA* manner, with each *tile* serving as a basic *Structure-of-Array* unit. *I.e.*, the *cell* properties inside each *tile* is organized in an *SoA* manner while the *tile structures* are listed in an outer array. By controlling the *tile* size, user can switch grid data to either *SoA* (when *tile* size equals to the *block* size) or *AoS* (when *tile* size equals to $1 \times 1 \times 1$ *cell*). As proposed in (Wang et al., 2020), this design helps improve data vectorization inside a contiguous array of member variables and overall device cache efficiency with small grid tiles.

The following code example shows how to define and allocate the sparse grid in the proposed programming model. In line 1, we specify the primary value type of grid attributes. Moreover, in lines 2-3, we define the attributes stored on grids by listing all member types (mass (1D) and grid momentum/velocity (3D)). Users can easily adjust data channels by modifying the template definitions. Then, in lines 4-5, we create a sparse map (§4.4.1) to record valid grid *tiles* in each simulation step. Here, the `MEMSPACE` indicates whether the hashing data is on CPU or GPU and further decides whether the hash insertions or queries are performed or paralleled within the host or device kernels.

```
1  using T = float;      // or other types like double
2  using node_members = // mass and momentum in MPM simulation
3    Cabana::MemberTypes<T, T[3]>;
4  auto sparse_map =     // hash table, Sec 4.1
5    Cajita::createSparseMap<MEMSPACE>(global_mesh, reserve_size);
6  /* create grid array layout, contains sparse halo (Sec 4.2); the entity
       type Cajita::Node() indicates values are stored on grid nodes */
7  auto layout =
8    Cajita::createSparseArrayLayout<node_members>(local_grid, sparse_map,
       Cajita::Node());
9  auto nodes =          // allocated grid AoSoA
10   Cajita::createSparseArray<DEVICE>("nodes", layout);
11 nodes.reserve(pre_allocate_cell_num);   // optional
```

Later in lines 7-8, we need to specify the *grid layout* from the *local grid*, *sparse map*, and the entity type to support the actual array allocation. In detail, entity type guides the halo communication and *array* allocation (§4.4); while *local grid* computes the *owned-/ghosted-tile* ranges. Note the *tile* ranges in *shared spaces* require update once the simulation domain is partitioned (§4.5) to ensure communication correctness. Finally, an *AoSoA* array is created with the pre-prepared information in lines 9-10. Automatic reallocation will be triggered during simulation if the valid grid *array* size exceeds the allocated capacity. We recommend explicitly reserving spaces for grid data by providing an estimation of the maximum valid cell number (line 11) to reduce performance drop caused by unnecessary reallocation.

4.5 Distributing and Load Balancing

For performance-portable large-scale simulations, evenly distributing the workload to multiple ranks is essential. Considering the significance of load balancing, simulation communities have formulated partitioning as a domain optimization problem (Surmin et al., 2015). However, most existing works focus on the theory and formulation. In this section, we propose and demonstrate a detailed dynamic load-balancing algorithm as an essential component of most distributed computing systems. The corresponding implementation is integrated into our *particle/grid data layer*, *i.e.*, into the `Cabana` library. It will be fully open-sourced with detailed documentation and unit tests. In the rest of

Algorithm 2 Dynamic Load Balancing

```
Input: Sparse map $map$                                   ▷ for dynamic grid partitioning
Input: Particle positions $pos_p$                         ▷ for dynamic particle partitioning
Input: MPI communicator $comm$
Output: Optimized partition $P = \{I, J, K\}$
Output: Optimization iteration times performed $n$
  computeLocalWorkload($map$ or $pos_p$)                  ▷ §4.5.1.1
  computeGlobalWorkload($comm$)                           ▷ §4.5.1.2
  computePrefixSum                                        ▷ §4.5.1.3
  $n \leftarrow 0$
  while $n < n_{max}$ do                                  ▷ $n_{max}$: max iteration time
      $dim\_sequence \leftarrow$ random permutation of $\{0, 1, 2\}$
      for all $d \in dim\_sequence$ do
          $is\_changed \leftarrow false$
          $is\_dim\_changed \leftarrow$ optimization1D($d$)
          $is\_changed \leftarrow is\_changed \, || \, is\_dim\_changed$
      end for
      $n \leftarrow n + 1$
      if NOT $is\_changed$ then
          return $n$
      end if
  end while
```

this chapter, we first introduce two definitions of simulation workload in §4.5.1, and then explain our 3D partition optimization in §4.5.2. The complete dynamic load balance optimization algorithm is summarized in Algorithm 2. More implementation details and the unit test of the proposed dynamic load balance algorithm can be found in section B.1 and section B.2, respectively.

4.5.1 Workload Computation

As introduced in §4.4, the entire work is distributed by partitioning the simulation domain into non-overlapped rectangular sub-regions according to the MPI topology, with every independent MPI rank handling each sub-region. To perform the partition optimization, we need to evaluate the workload on each MPI worker, *i.e.*, inside each rectangular sub-region, which changes dynamically throughout the simulation. In addition, the optimization process requires frequent workload analysis of the partitioned attempts.

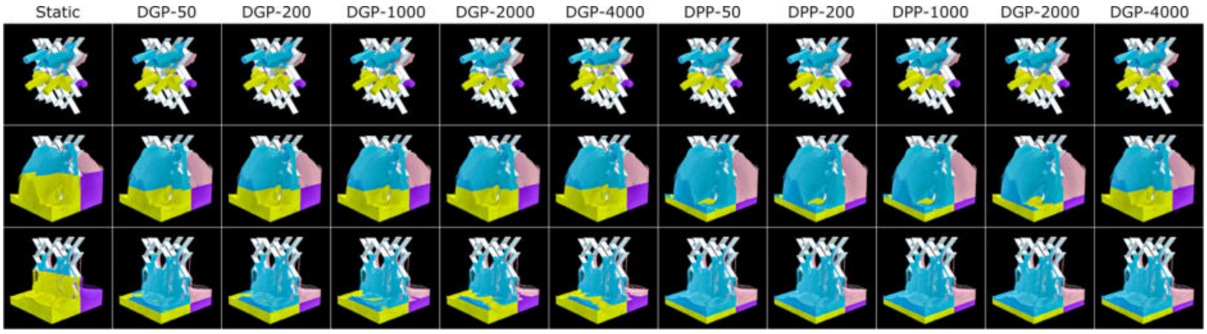


Figure 4.8: **Sand injection** with different partition algorithms (133.2M particles in total, grid resolution 256 × 256 × 256). The first column shows static partitioning result, and *DGP-N* refers to *Dynamic Grid Partitioning* per *N* simulation steps. Similarly, *DPP* refers to *Dynamic Particle Partitioning*. Row 1 to 3 shows the results for frames 25, 99, and 145, respectively. Different color refers to particles (simulation sub-domain) handled by different MPI ranks.

Thus, we need a representation that supports efficient workload computation within any rectangle region.

We construct a 3D matrix to realize this goal, with each element referring to the workload value inside the corresponding area. The granularity of the workload matrix influences the accuracy and performance of the load balancing optimization. A matrix with elements representing smaller-sized regions helps the optimizer to make a more accurate and flexible choice but may increase the computation and communication overhead.

4.5.1.1 Local Workload Computation

First, we count the workload handled locally on each MPI rank. In hybrid simulation methods, particles and grids are two crucial representations. Thus, both can measure the work amount, leading to two types of workload computing methods.

Particle-based workload computation. In this case, each particle is treated as a work unit. We make a parallel loop over particle positions and perform atomic addition to corresponding elements in the workload matrix. This method finally leads to a partition where all MPI ranks contain a similar number of particles. Generally, hybrid methods use dramatically more particles than valid grids (typically, each grid *cell* contains at least eight particles in 3D, and sometimes more to increase details and reduce numerical fractures). Thus, computations involving particles and atomic additions consume more computing and time resources for workload statistics. Nevertheless, balanced particle distribution can potentially benefit the timing of particle-grid data transfer if particles per cell are similar all over the domain because particle number decides the number of parallel kernels and the majority of memory accesses.

Grid-based workload computation. In order to improve the load balancing time efficiency, we also support the workload computation based on valid grid *tiles*. The compute kernel loops over the hash table in the *sparse map* and set the workload matrix element to 1 if the *tile* is valid. *I.e.*, no atomic additions are required, and fewer matrix elements are involved compared to particle-based computation.

Discussion on the choices. Generally speaking, both methods estimate the workload distribution from different perspectives in a given domain. When particles are relatively evenly distributed in grids, *e.g.*, in elastic simulations, these two representations will generate similar load balancing results. In this situation, grid-based workload outperforms as it uses fewer computing resources. However, in simulations for granular media and fluids, the particles can splash out dramatically or gather locally. In this case, particle-based method standouts because valid grid *tiles* are likely to contain significantly different numbers of particles, leading to distinct *P2G* and *G2P* time and memory requirement on different ranks. This may influence the simulation performance, as demonstrated in §4.7, or cause run-time particle memory allocation errors after *particle communication* for large-scale scenes.

In this chapter, we refer to the load balancing algorithm as *dynamic grid partitioning* if the workload is computed from the valid grid *tiles*, and as *dynamic particle partitioning* for the particle-based case. See §4.7 for detailed comparison results.

4.5.1.2 Global Workload Computation

All MPI ranks need to know the workload distribution in the whole simulation domain to perform global optimization. Thus, we need to gather all the computed local workload matrices to form a global matrix. We achieve this calculation by performing MPI reduction among all ranks with the `MPI_Allreduce` interface. Note that CUDA-aware MPI is required if the simulation uses CUDA memory and GPU execution space.

4.5.1.3 Global Workload Prefix Summation

We must scan all dimensions to perform load balancing optimization and analyze if the current partition is optimal. This process requires frequent workload counting inside any arbitrary rectangle sub-regions. Inspired by Surmin et al. (2015), we compute the 3D prefix summation of the global workload matrix, pursuing a constant-time workload estimation. Specifically, we adopt `Kokkos::parallel_scan` interface as an efficient solution for dispatching parallel inclusive/exclusive scans with a user-defined functor and a parallel execution policy. We scan the 3D workload matrix in three dimensions separately to compute the 3D prefix summation matrix. *I.e.*, the first scan is in the x direction, and then the second and third scans are based on the intermediate matrices in y and z direction individually. The concrete algorithm is listed in the supplemental document.

4.5.2 Partition Optimization

In this section, we first summarize the formulation of the 3D partition optimization process and then introduce the detailed algorithm we used for dynamic load balancing implementation. We use I, J, K to represent the partition in dimension x, y and z, with $I = (i_0, i_1, ..., i_{N_x})$, $J = (j_0, j_1, ..., j_{N_y})$, and $K = (k_0, k_1, ..., k_{N_z})$ indicating the dividing boundary sets. Here, N_x, N_y and N_z are the total number of MPI ranks in corresponding dimensions, and i_*, j_*, k_* refers to the *tile* indices. Specifically, $i_0 = j_0 = k_0 = 0$ and $i_{N_x}, j_{N_y}, k_{N_z}$ equals to the total number of *tiles* in x, y and z dimension, respectively.

Note that the workload matrix granularity will influence the unit of the pre-mentioned indices in the proposed implementation. In practice, to make implementations easily understandable and consistent, we use grid *tile* as the atomic unit of 1) workload matrix, 2) partition boundary index, and 3) grid data communication.

For any given rank (α, β, γ), the local grid domain it in charges is given by grid *tiles* $\{(i,j,k)|i_\alpha \leqslant i < i_{\alpha+1}, j_\beta \leqslant j < j_{\beta+1}, k_\gamma \leqslant k < k_{\gamma+1}\}$. Suppose the starting *tile* of the current rank are optimized and fixed; the proposed load balancing algorithm will find the optimal ending *tile* indices by solving the following optimization.

$$\min_{i_{\alpha+1}, j_{\beta+1}, k_{\gamma+1}} : \sum_{i=i_\alpha}^{i_{\alpha+1}} \sum_{j=j_\beta}^{j_{\beta+1}} \sum_{k=k_\gamma}^{k_{\gamma+1}} |W_{i,j,k} - \overline{W}|$$

Here, $W_{i,j,k}$ is the workload in *tile* (i,j,k) and $\overline{W}$ refers to the average rank workload computed by $\frac{\sum_{i=0}^{N_x} \sum_{j=0}^{N_y} \sum_{k=0}^{N_z} W_{i,j,k}}{N_x \times N_y \times N_z}$.

As discussed in (Surmin et al., 2015), this optimization is an NP-complete problem. With previous partitions $(i_0, ..., i_\alpha)$, $(j_0, ..., j_\beta)$ and $(k_0, ..., k_\gamma)$ fixed, there are three degree-of-freedoms to decide, *i.e.*, the optimal partition $i_{\alpha+1}, j_{\beta+1}$ and $k_{\gamma+1}$, for the current rank. In addition, the results will influence the computation for later ranks with larger rank indexing values. To solve this problem with three unknowns, we iteratively alternate among each variable and perform 1D optimizations. The iteration will stop when the partitioning results are unchanged or the maximum iteration number is reached, as shown in Algorithm 2. In the supplemental document, we present a validation example to show that this iterative algorithm can generate the optimal solution with several iterations when the ground truth is unique.

4.5.2.1 1D Load Balancing Optimization

Inside each 1D optimization, we randomly choose one dimension of interest that is never covered in the current iteration. This randomness reduces the possibility for the algorithm to be trapped into local optimal and potentially reduces the iteration times. Then, the

Algorithm 3 Compute Workload in a Given Sub-domain

```
Input: Dimension label $d_i$ and tile range $l_i, h_i$
Input: Dimension label $d_j$ and tile range $l_j, h_j$
Input: Dimension label $d_h$ and tile range $l_h, h_h$
Input: Workload prefix sum matrix $WS$
Output: Workload in tile range $[l_i, h_i] \times [l_j, h_j] \times [l_k, h_k]$
  function ComputeWorkload($d_i$, $l_i$, $h_i$, $d_j$, $l_j$, $h_j$, $d_k$, $l_k$, $h_k$)
      $s[d_i] \leftarrow l_i$, $s[d_j] \leftarrow l_j$, $s[d_k] \leftarrow l_k$
      $e[d_i] \leftarrow h_i$, $e[d_j] \leftarrow h_j$, $e[d_k] \leftarrow h_k$
      return $WS(e[0], e[1], e[2]) - WS(s[0], e[1], e[2])$
          $-WS(e[0], s[1], e[2]) - WS(e[0], e[1], s[2])$
          $+WS(s[0], s[1], e[2]) + WS(e[0], s[1], s[2])$
          $+WS(s[0], e[1], s[2]) - WS(s[0], s[1], s[2])$
  end function
```

partition in the non-chosen two dimensions is fixed. All partition boundaries in the dimension of interest will be reanalyzed individually for a more even workload division. The detailed algorithm is summarized in the supplemental document.

In practice, it is possible to have a range of consecutive *tiles* where there are no valid particles. In theory, any *tile* indices in this range can be treated as optimal partition positions. However, because dynamic load balancing is not performed in every simulation time step, if we choose the pre-mentioned *tile* range boundaries as the partition position, the particles may move over the range boundaries before the next round of partitioning. This choice will cause extra particle communications in the upcoming steps, especially for solid simulations, where many particles tend to gather together, and the particle communication overhead would be considerable. Therefore, we always set the partition point as the middle point of the equivalent *tile* range where there are no particles. This simple operation reduces the potential particle communications among MPI ranks and improves the overall performance. For

4.5.2.2 Workload Computation

Once the 3D prefix summation matrix of the global workload is obtained, we can calculate the workload in any given rectangle domain within constant time as shown in Algorithm 3.

4.5.3 Partition Optimization

We apply three separate 1D optimizations to approximate the optimal solution of the 3D partition optimization. Details of the 1D case is shown in Algorithm 4.

4.6 Distributed MPM Implementation

4.6.1 Time Integration

As mentioned in §4.3, we implement the first-order MPM time integration scheme including three basic *computation kernels*, *i.e.*, **P2G**, **Grid Update**, and **G2P**. For distributed systems, another two *communication kernels*, **Grid Halo Communication** and **Particle Communication** are required to guarantee the correctness. In detail, **Grid Halo Communication** is performed before *Grid Update* to ensure the completeness of the grid data on each MPI worker. It consists of the *halo scatter* and *gather* operations introduced in §4.4.2. Then, after updating particle positions in **G2P**, we end up time integration step with **Particle Communication** to distribute particles to the ranks in charge of the corresponding grid sub-domain. Additionally, **Dynamic Partition Optimization** is performed right before *Particle Communication* at certain time steps to ensure a relatively balanced load distribution. The entire pipeline is illustrated in Figure 4.5.

4.6.2 PIC Algorithms and Application Implementation

To make a more complete distributed MPM simulation system, we add the *PIC Algorithm layer* to further support the *application layer* in building up versatile large-scale scenes. In addition to the core time integration routines, we add components like device-portable sparse collision object, particle sourcing, analytic/VDB-based shape, and multi-material constitutive modeling with elasticity and plasticity, forming the `MultiSim` library (Figure 4.1). We support four types of constitutive models, including fixed-corotated model (Stomakhin et al., 2013) for elasticity, Drucker-Prager elastoplasticity (Klár et al., 2016) for sand simulations, Non-Associated Cam-Clay (NACC) (Wolper et al., 2019; Li et al.,

Algorithm 4 1D Rectangle Partition optimization

```
Input: Dimension-of-interest: $d$
Output: Optimized partition: $P$, with $P_0 = I, P_1 = J, P_2 = K$
Output: If partition is updated: $is\_changed$
  function OPTIMIZATION1D($d$)
   ▷ solve 1D rectangle optimization given partitions in the other two dimensions fixed
    $d_i \leftarrow d$
    $d_j \leftarrow (d + 1) \mod 3$
    $d_k \leftarrow (d + 2) \mod 3$
    for all $j \in [0, N_j)$, $k \in [0, N_k)$ do                    ▷ in parallel
      $W_{all}(j, k) \leftarrow$
        COMPUTEWORKLOAD($d_i$, 0, $N_i$, $d_j$, $P_{d_j}(j)$, $P_{d_j}(j + 1)$, $d_k$, $P_{d_k}(k)$, $P_{d_k}(k + 1)$)
      $W_{ave}(j, k) \leftarrow W_{all}(i, j) \,/\, N_i$
    end for              ▷ $N_i$, $N_j$ and $N_k$: rank number in corresponding dimensions
    $p_{i-1} \leftarrow 0$
    $p_i \leftarrow 1$
    $eq_{start} \leftarrow 1$                       ▷ record equivalent partition range
    $last\_diff \leftarrow INT\_MAX$
    $P_{d_i}(0) = 0$
    for all $rank = 1, ..., N_i - 1$ do
      while $true$ do
        for all $j \in [0, N_j)$, $k \in [0, N_k)$ do                ▷ in parallel
          $W(j, k) \leftarrow$
        COMPUTEWORKLOAD($d_i$, $p_{k-1}$, $p_k$, $d_j$, $P_{d_j}(j)$, $P_{d_j}(j + 1)$, $d_k$, $P_{d_k}(k)$, $P_{d_k}(k +$
  $1)$)
        end for
        $diff \leftarrow \sum_{j,k} |W(j, k) - W_{ave}(j, k)|$              ▷ parallel reduce
        if $diff < last\_diff$ then
          $eq_{start} \leftarrow p_k$
          $last\_diff \leftarrow diff$
        else
          if $P(d_i, rank) \neq (p_i - 1 + eq_{start})/2$ then
            $P(d_i, rank) \leftarrow (p_i - 1 + eq_{start})/2$
            $is\_changed = true$
          end if
          $p_{i-1} \leftarrow p_i$
          break while loop
        end if
        $p_i \leftarrow p_i + 1$
      end while
    end for
  end function
```

2022b) for snow/mud-like behaviors, and furthermore, weakly compressible fluids (Tampubolon et al., 2017) for liquids. Users can easily specify the component or even extend the current *PIC Algorithm layer* for more applications. In the section B.3, we provide a concrete example showing how to use the components in *application level*. More demos can be found in our open-source code.

4.7 Results and Evaluations

This section evaluates the proposed distributed MPM framework with scaling tests, load balancing comparisons, MPI Cartesian topology comparisons, and large-scale demonstrations. We use at most eight workstations (each as one MPI rank) in our experiments. The workstation has one Intel Core i9-10920X (12 core, 24 threads, base clock 3.50Hz) and one NVIDIA GeForce RTX 3090 GPU. We adopt 10-Gigabit bandwidth Ethernet to support inter-rank communications, with OpenSSH (developers, 2021) and CUDA-aware OpenMPI 4.1.2 (Members, 2021). All the evaluations and demonstrations are conducted under this setup unless stated otherwise. In addition, the *fixed-corotated* constitutive model (Stomakhin et al., 2012b) and dynamic grid partitioning (per 200 simulation steps) are adopted for all scaling tests.

4.7.1 Multi-MPI Scalability

Scalability with increased computing resources is a widely adopted test to evaluate the effectiveness and robustness of a distributed algorithm. Ideally, performance should scale up with the number of involved MPI ranks. However, perfect scaling is not practical. Specifically, Amdahl's law and Gustafson's law demonstrate the limitation of parallel computing; furthermore, the communication bandwidth also constrains the upper bound of multi-rank acceleration. To analyze the performance of the proposed distributed MPM system, we present the scaling results on local workstations with both CUDA and OpenMP as the latent programming models respectively.

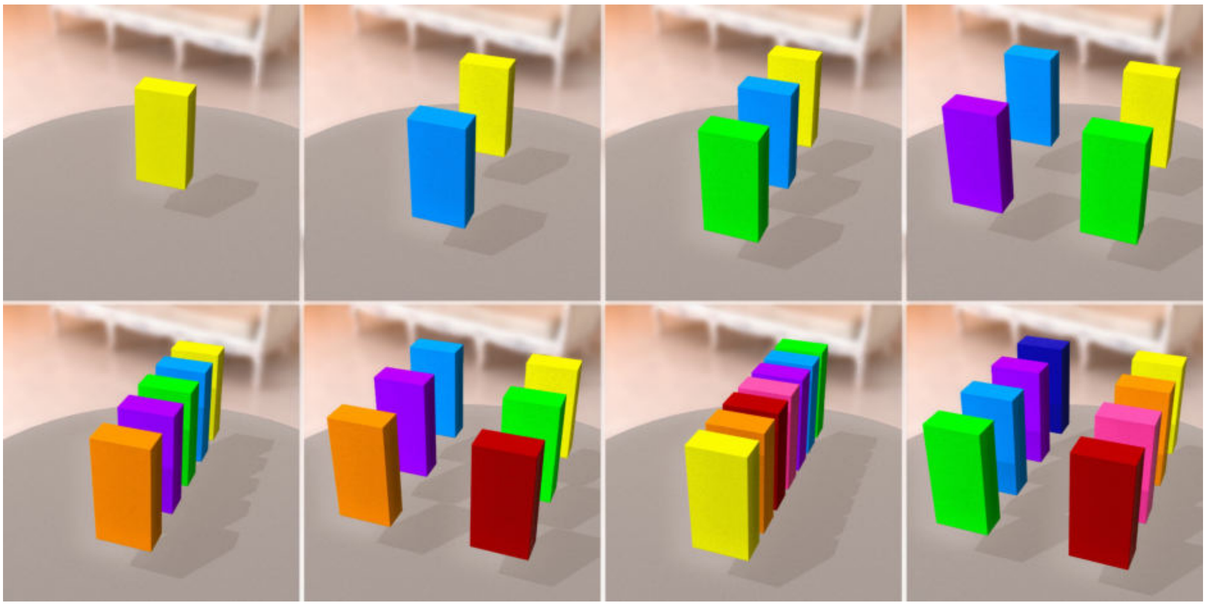


Figure 4.9: **Weak Scalability** scene setup with 1-8 workstations. Different colors refer to different MPI ranks in each sub-figure.

4.7.1.1 Weak Scaling

Inspired by Gao et al. (2018b), we set up the experiment by placing an elastic cuboid at the center of each rank's *local mesh* and let it fall with gravity, as illustrated in Figure 4.9. All cuboids are of the same size with 28.8M particles. The MPI rank topology is $n \times 1 \times 1$ for rank number $n = 1, 2, 3, 5, 7$, and $n/2 \times 1 \times 2$ for $n = 4, 6, 8$. We summarize the experiment timing and efficiency of each computing/communication kernel in Figure 4.20. For communication kernels (*Particle Communication*, *Dynamic Partition Optimization* and *Grid Halo Communication*), efficiency is computed with 2-rank timings as the 100% base, since there's little communication overhead for the 1-rank case. As demonstrated, the aggregated weak efficiency is over 95% regardless of the rank numbers.

To further illustrate the scaling potential of the proposed model, we run the weak scaling test with more cuboids on the Summit supercomputer with up to 120 ranks (20 nodes). Each compute node on the Summit contains six NVIDIA Tesla V100 GPUs (six MPI ranks), and these GPUs are grouped into two sockets. Thus, various levels of latencies are incurred for both cross- and within-node communications. The results are shown in Figure 4.24.

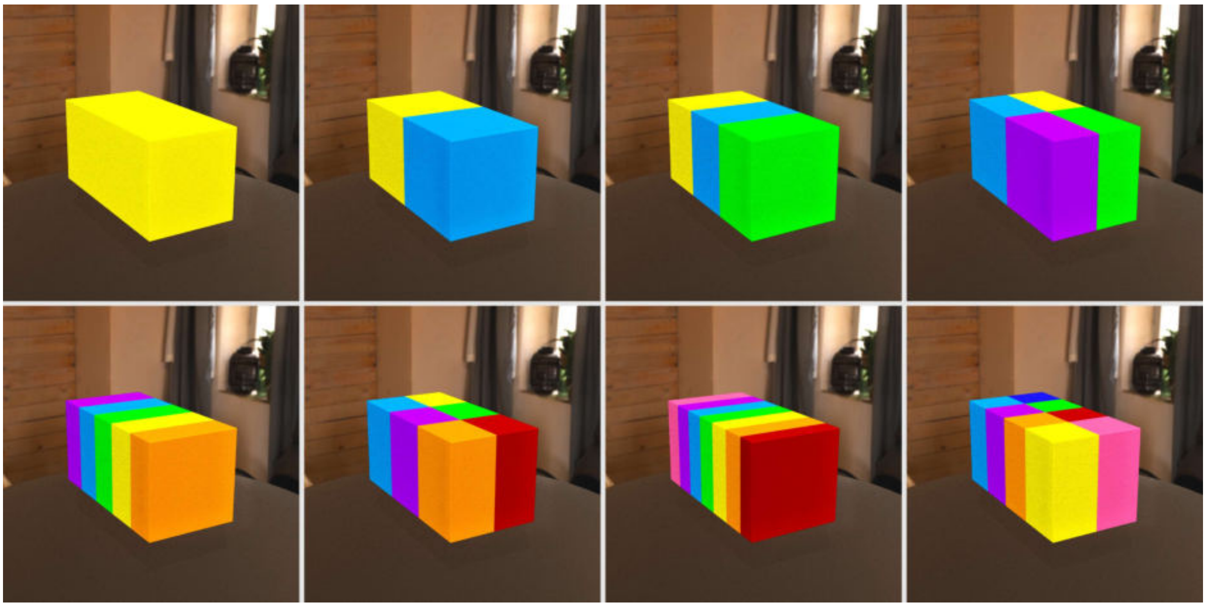


Figure 4.10: **Strong Scalability** scene setup with 1-8 workstations. Different colors refer to the particles handled by different MPI ranks in each sub-figure.

The scalability tests on local workstations with OpenMP as the latent programming model are shown in Figure 4.22. While the scene configurations remain consistent with the CUDA cases, we lower the particle size of each cuboid to 1M to speed up the evaluation process.

4.7.1.2 Strong Scaling

For the strong scaling test, we assign a falling cuboid at the center of the *global mesh* as a fixed-size problem and bring different numbers of MPI ranks into the computation. The cuboid contains 159M particles for the CUDA test. Using the proposed dynamic load balancing algorithm with a user-specified MPI topology, as shown in Figure 4.10, the whole workload is automatically split up and given to ranks. We conduct the experiments with the same MPI rank topology settings as in §4.7.1.1. The timing and speedup analysis are illustrated in Figure 4.21. Our system can pursue an almost linear overall speedup as the MPI rank increases.

The strong scalability tests on local workstations with OpenMP as latent programming model are shown in Figure 4.23, and the cuboid is resized to 10M particles.

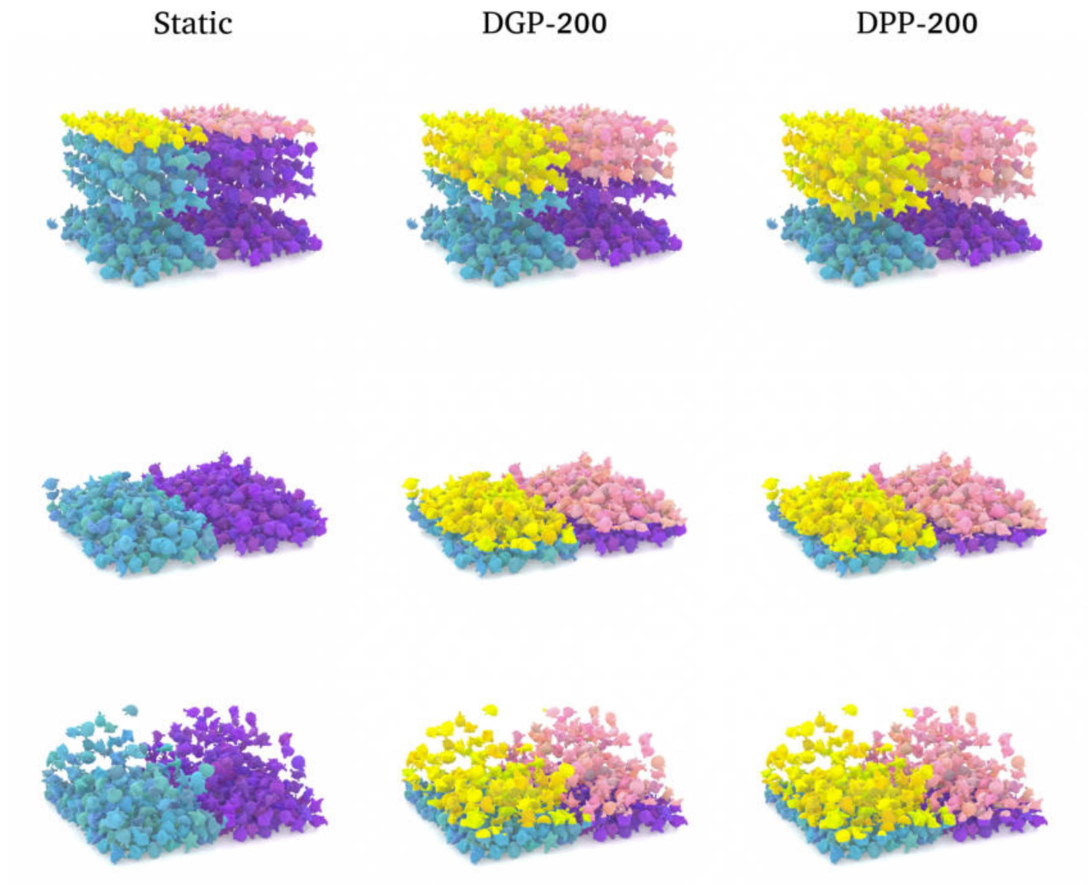


Figure 4.11: **Elastic toys** with different partition algorithms distributed on 4 (2 × 2 × 1) MPI ranks (22.4M particles in total, grid resolution 256 ×256×256). Row 1 to 3 shows the results for frames 6, 12, and 22, respectively. Particles are colored yellow, blue, purple, and pink to indicate the MPI ranks they belong to. Minor hue differences are applied to separate toys.

4.7.2 Load Balancing Studies

Dynamic load balancing generally boosts the simulation performance of a distributed system from several perspectives, as stated before. However, partition optimization and commensurate particle relocation may require significant computation and communication time. In addition, various material behaviors may lead to divergent partition results when using different workload elements. Therefore, we conduct several experiments with multiple material behaviors to evaluate the proposed load balancing algorithms in this section. The results and discussions can help users find the best choice for their simulation objectives.

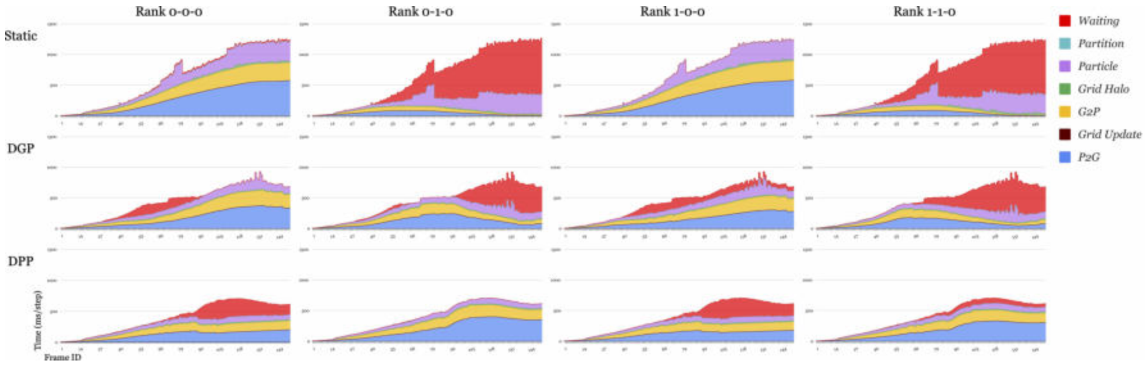


Figure 4.12: Detailed timing per step (in milliseconds) of sand injection with *static partitioning*, *dynamic grid partitioning* (DGP) and *dynamic particle partitioning* (DPP).

4.7.2.1 Sand Injection

In this experiment, we focus on comparing the behavior of the *static*, *dynamic grid*, and *dynamic particle partitioning* methods and analyzing how partitioning frequency influences the performance. As displayed in Figure 4.8, we design a 4-MPI-rank *sand injection* scene, where each rank injects sand from two sourcing points with random velocities pointing towards the shelf (collision object) sitting at the domain center. Throughout the simulation, sand material sometimes splashes and finally settles down, leading to dynamically varying workload distribution.

Dynamic Partitioning V.S. Static Partitioning For a thorough analysis, we illustrate the detailed timing on all 4 ranks for *static*, *dynamic grid*, and *dynamic particle* partitions in Figure 4.12. In addition to the timing of each separate kernel, we also show waiting time, which refers to the duration when faster ranks finish computation/communication and wait for other ranks. We show the data with dynamic partitions performed every 50 steps without loss of generality. In the first row of Figure 4.12, static partitioning pushes more work to lower ranks (rank 0-0-0 and 1-0-0) as the sand particles fall to the ground. The upper ranks (rank 0-1-0 and 1-1-0), on the other hand, contain fewer and fewer particles and thus sit idle, wasting time waiting for the lower ranks. This issue is mitigated when the dynamic partition is adopted (rows 2-3 in Figure 4.12).

In this test, some sand particles splash out in the upper sub-domains while the others pile up at the bottom. This uneven particle-per-grid-*tile* distribution leads to different behaviors of *dynamic grid partitioning* and *dynamic particle partitioning.* When applying *dynamic grid partitioning*, each rank contains a similar number of grids, but the upper ranks need to handle more particles. This fact means that more parallel work is required for the upper ranks to perform *P2G* and *G2P*, and thus the lower ranks become idle, especially after frame 90. For *dynamic particle partitioning*, the roles reverse as the lower ranks need to handle more of the grid, making the upper ones wait.

Dynamic Partition Frequency We compare the speedup of *dynamic grid/particle partitioning* with different frequencies to static partitioning in Figure 4.13. This specific simulation takes around 233 steps per frame, and the sand particles are continuously injected until frame 80. We choose partitioning step intervals to be 50, 200, 1000, 2000, and 4000 for testing, *i.e.*, performing dynamic load balancing per about 0.25, 1, 4, 8, and 17 frames.

As illustrated in Figure 4.13, all choices achieve over 1.4x speedup and can reach 2.3x in some frame ranges. In theory, the best speedup would be about 2x, as the extreme case is that the lower two ranks handle all workloads and the upper two do nothing but wait. This speedup can be better in practice when considering the overhead of parallel scheduling, memory access, and communication.

Overall, *dynamic particle partitioning* outperforms the grid-based method for the splashing materials. Moreover, each partitioning frequency has a different speedup trend through frames 0-25, 25-80, and 80-150. This indicates that the particle/grid number (problem scale), material behavior (sourcing, splashing, and falling), as well as motion (if particles are moving towards the same direction as the partition boundaries) will all influence the actual performance. Despite the partitioning frequency, our dynamic load balancing algorithm, compared to the static case, can always accelerate the simulation process, as summarized in Table 4.1, and it can gain more speedup for large-scale cases that consume more time (after frame 80 when sourcing stops, as in Figure 4.13).

In particular, we observe that *dynamic particle partitioning* per 4000 steps behaves

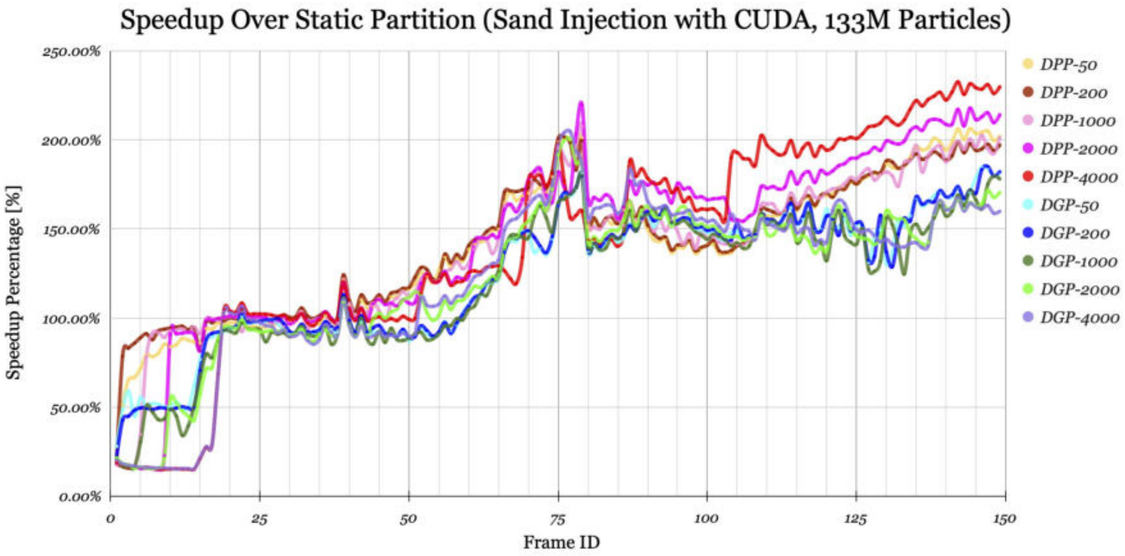


Figure 4.13: Aggregated time speed up of dynamic load balancing over *static partitioning*. *DGP-N* and *DPP-N* refer to *dynamic grid partitioning* and *dynamic particle partitioning* performed every N steps, respectively.

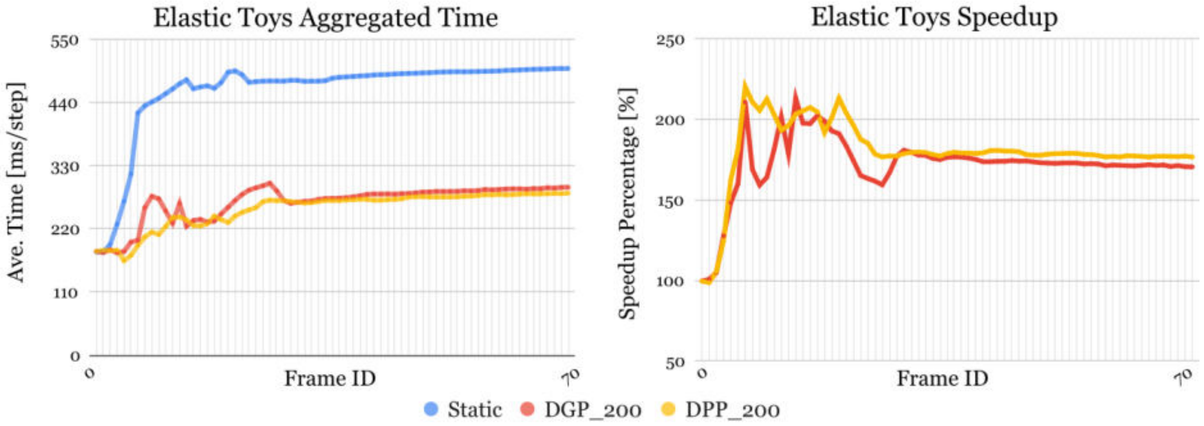


Figure 4.14: **Elastic Toys.** (Left) Aggregated simulation time statistic, averaged on each step. (Right) Speedup of dynamic partitioning over static partitioning.

Method	Static	DPP-50	DPP-200	DPP-1000	DPP-2000	DPP-4000
Time (h)	**6.15**	3.83	3.82	3.83	**3.62**	3.64
Speedup (%)	-	160.49%	160.87%	160.30%	***169.89%***	169.10%
Method (h)	-	DGP-50	DGP-200	DGP-1000	DGP-2000	DGP-4000
Time (h)	-	4.26	**4.25**	4.37	4.25	4.26
Speedup (%)	-	144.28%	***144.68%***	140.69%	144.49%	144.13%

Table 4.1: **Sand Injection Speedup.** We summarize the total simulation time of the 150 frames in hours and the speedup of partition methods with different partitioning frequencies. *Dynamic particle partitioning* and *dynamic grid partitioning* achieve the best overall speedup 2000 and 200 steps, separately.

better than other cases after frame 80. There are two possible reasons. First, frequent partition changes prompt immediate particle relocation among ranks. It will also introduce extra *particle communication* work in the following steps, especially when particles and partition boundaries move in the same direction. Second, a relatively perfect particle partition leads to an undesirable grid partition for splashing sands. Nevertheless, delayed partitioning alleviates this situation by pushing more particles to lower ranks but more grids to the upper ranks, thus leading to more rank-balanced particle-grid computations. *dynamic grid partitioning*, however, cannot benefit from this partitioning delay.

4.7.2.2 Elastic Toys

This experiment shows how partition methods behave when materials splash considerably less. Initially, we assign 4 MPI ranks the same number and type of toys and thus the same amount of particles, and we drop them as shown in Figure 4.11. With toys falling down, particles are communicated to lower ranks if they pass the upper-lower partition interface. Here the overall acceleration rates (173% for *dynamic grid partitioning* and 180% for the particle case) are similar and are close to the best theoretical speedup (200%). In a detailed timing statistics (Figure 4.14), we notice better acceleration with *dynamic particle partitioning* before frame 30. It happens because the toys are of irregular shape and random orientation. Thus, some active grid *tiles* contain only a sharp toy corner with few particles. Lower toys reach the bottom while falling, but the upper ones are still

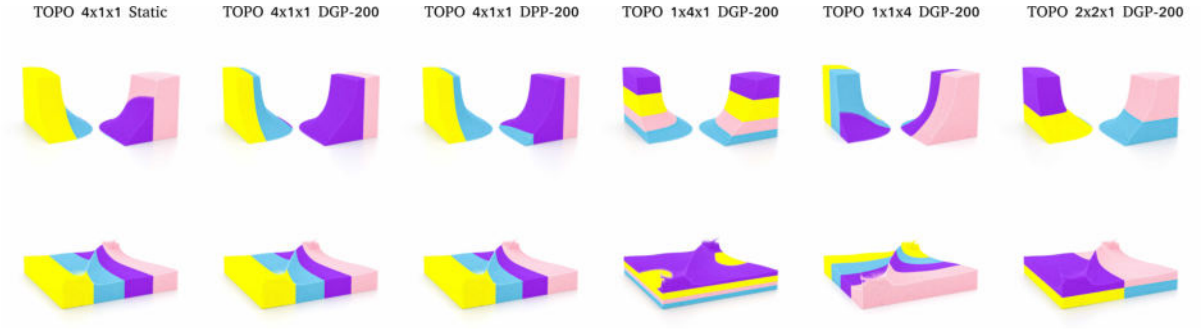


Figure 4.15: **Sand dambreak** with different partition algorithms and MPI topology settings (17M particles in total, grid resolution 256 × 256 × 256). TOPO in the figure refers to MPI Cartesian topology. Rows 1 and 2 show the results of frames 21 and 49, respectively. Different color refers to particles (simulation sub-domain) handled by different MPI ranks.

placed evenly in the sky. As a result, more grids will be activated in the upper domain, leading to a partition boundary closer to the upper toy group. The toy's falling direction makes the workload less balanced in the steps prior to the next round of partitioning. One of the representative frames is shown in the first row of Figure 4.11.

4.7.2.3 Sand Dam break

Another classic scene we test is the sand dam break illustrated in Figure 4.15. Initially, two sand columns are located at the diagonal corner of the entire domain. Unlike previous settings, we use MPI rank topology 4 × 1 × 1 to evaluate the behavior of dynamic load balancing algorithms. In this simulation, the sand flows towards the domain center and settles down onto the floor at last. There is no splashing or extreme deformations throughout this process. Thus the two dynamic partition methods achieve similar speedups as demonstrated in Figure 4.16. In addition, with more and more sands gathering into the middle domain, *static partitioning* gradually becomes a naturally appropriate approach (after frame 50). In this case, the dynamic load balancing methods exhibit only 10%-15% performance improvement.

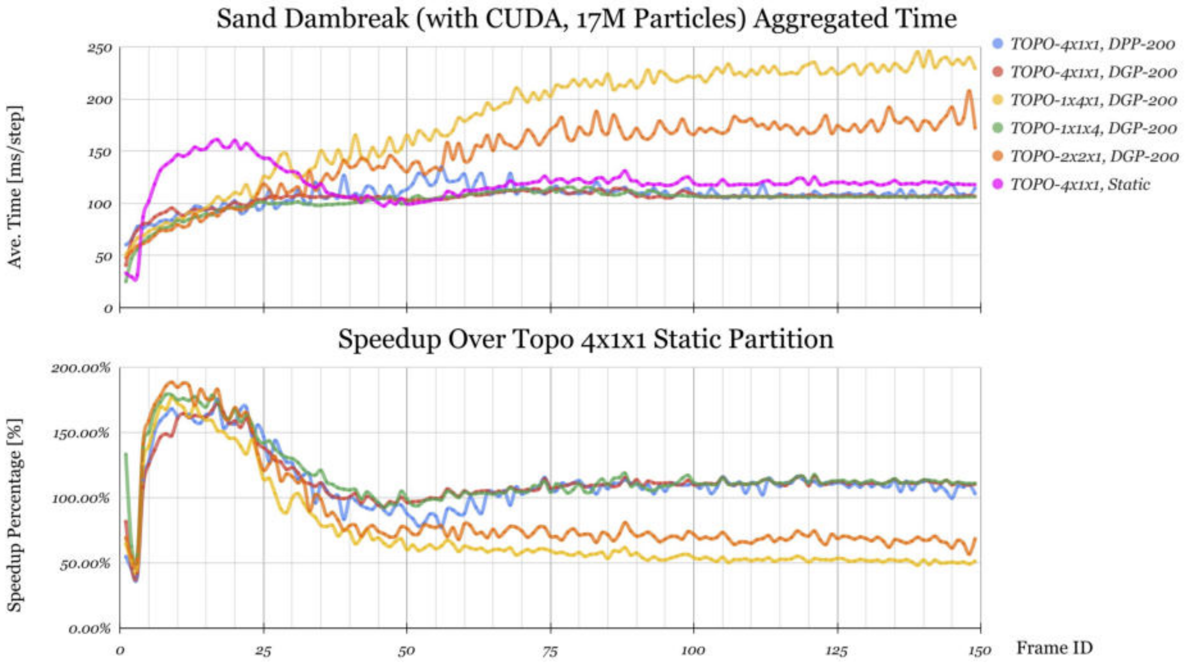


Figure 4.16: **Sand Dambreak.** (Top) Aggregated simulation time statistic, averaged on each step. (Bottom) Speedup over static partitioning with MPI rank topology 4 × 1 × 1.

4.7.3 Cartesian Topology of MPI Ranks

Here we observe another factor that strongly influences the performance of distributed simulators: the initial MPI Cartesian topology setting. In this section, we use the *Sand Dam break* example introduced in §4.7.2.3 for illustration. We rerun the simulation with MPI topology 1 × 4 × 1, 1 × 1 × 4, and 2 × 2 × 1 (Figure 4.15) and compare the timing to the case 4 × 1 × 1. To make a fair comparison, we adopt *dynamic grid partitioning* for all the new topology settings and summarize the timings in Figure 4.16.

With identical computing resources, the simulation performance reduces significantly with inappropriate MPI topologies (1 × 4 × 1 and 2 × 2 × 1). There are several reasons leading to this result in this specific simulation scene. First, sands move with gravity towards the $-y$ direction. If there are rank boundaries on the y dimension, particles must be relocated to other ranks when flowing down, increasing the *particle communication* overhead. Second, diverse rank topologies lead to a different number of neighbors and the size of halo areas, causing performance variance. Consequently, the takeaway is that

Example	Particle#	Ave s/frame	$\mathbf{\Delta t_{frame}}$	Rank#	Grid Resolution
(Figure 4.2) 1B-Fluid	$1,006,766,992$	323.25	1/4	8	$256 \times 266 \times 256$
(Figure 4.3) Mudflow	$207,810,349$	159.34	1/10	4	$256 \times 256 \times 256$
(Figure 4.4) High-Res Sand	$266,507,608$	$1,072.95$	1/60	4	$512 \times 512 \times 512$
(Figure 4.7) Elastic Ground	$393,954,516$	229.56	1/4	4	$512 \times 512 \times 512$
	max $\mathbf{\Delta t_{step}}$	**$\mathbf{\Delta x}$**	**Material Parameters**		
	1.01×10^{-3}	100/256	Fluid: (500)-$(3 \times 10^6, 3)$		
	3.36×10^{-4}	200/256	NACC: $(1500, 1.5 \times 10^7, 0.3)$-$(-0.007, 0.05, 30, 30)$		
	3.58×10^{-5}	1/512	Sand: $(20, 1 \times 10^4, 0.4)$-$(30.0, 0.0)$		
	8.42×10^{-4}	200/512	Fix-corotated: $(1000, 9 \times 10^6, 0.4)$		

Table 4.2: **Parameters and timings.** We summarize the parameters of particle numbers, grid resolutions, MPI rank numbers, grid *cell* size Δx, and the average time per frame for various experiments described in subsection 4.7.4. The material-related parameters are listed as well. In addition to the basic material settings (density ρ, Youngs Modulus E, and Poisson Ratio ν), parameters needed by specific materials are provided. We refer to the corresponding papers for a physical explanation of the parameters. For *fix-corotated* model, we simply show (ρ, E, ν); while for Non-Associated Cam-Clay (NACC), parameters are given with format (ρ, E, ν)-$(\alpha_0, \beta, \xi$,friction angle); sand model (Drucker-Prager elastoplasticity) includes parameters (ρ, E, ν)-(friction angle, cohesion), and finally, we provide (ρ)-(k, γ) for fluids.

we should always carefully consider the particle distribution and motion tendency in the scene to set the initial MPI topology for the best performance.

4.7.4 Large-scale Simulations

This part demonstrates the scalability of the proposed distributed MPM scheme with a suite of large-scale simulations. The corresponding settings and average time per frame are summarized in Table 4.2. In addition, we show detailed timing statistics for large-scale simulations in Figure 4.19, Figure 4.17 and Figure 4.18. The high-resolution sand injection has similar detailed timing proportions as the low-res version (illustrated in Figure 4.12) and is not repeatedly illustrated.

1B-Fluid. Example in Figure 4.2 exemplifies a complex fluid scene containing 1.01 billion particles falling onto chip-shaped boards. To the best of our knowledge, this is the first MPM simulation beyond the scale of 1B. Initially, there is a water layer on the ground and eight liquid cubes falling on top. We use eight workstations to solve this

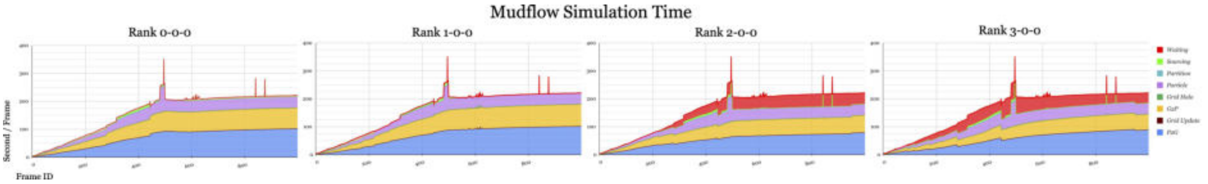


Figure 4.17: **Mudflow.** Detailed timing per frame (in seconds). Some Ethernet instability happened during particle communications in frames 495, 840, and 875, which can be regarded as external noises.

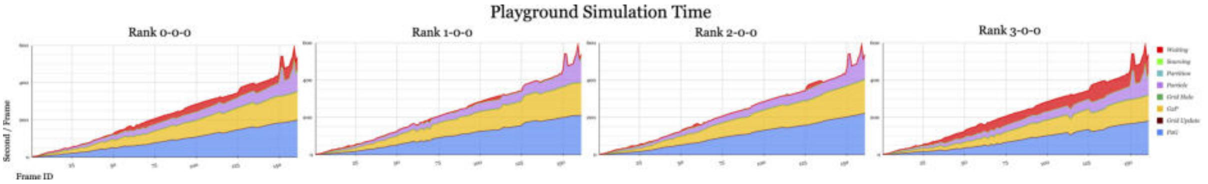


Figure 4.18: **Playground.** Detailed timing per frame (in seconds).

challenging problem with MPI topology 4 × 1 × 2 through CUDA parallelization. In order to maintain a balanced fluid distribution, we use the *dynamic grid partitioning* (per 50 steps). As a result, every MPI rank consistently uses 22 to 23 Gigabytes of GPU memory throughout the simulation. The fluid's turbulence is recorded in a large amount of detail, as shown in Figure 4.2.

Mud flow. We simulate natural mud flow (Figure 4.3) with NACC models. The domain is of size 200 × 200 × 200 meters, with a bumpy slope serving as a collision object. Mud particles are injected into the scene in the first 500 of the total 1000 frames and reach 207.8M at maximum. The simulation is performed with 4 × 1 × 1 MPI ranks and *dynamic grid partitioning* every 200 steps. Figure 4.3 also visualizes the mud flow damage propagation.

High-resolution Sand Injection. Figure 4.4 demonstrates the scalability with a high-resolution version of *Sand Injection*. We increase the spatial resolution to 512 × 512 × 512, and the scene reaches 266.5M particles at the middle point of the simulation time. Compared to Figure 4.8, there are more splashing and collision details in Figure 4.4. This demo has the same MPI rank topology as the low-res case.

Device	Method	Particle #	Timing (ms/step)
CPU	(Wolper et al., 2019)	10M	1034.98
	Ours	10M	2261.39
GPU	(Gao et al., 2018b)	20M	39.89
	Ours	20M	73.17

Table 4.3: **CPU/GPU SOTA comparison.** Note that the particle number is rounded.

Playground. Another scene involving more than 393.95M particles shows elastic jellos spreading onto the ground. We visualize some frames from different viewpoints in Figure 4.7. In this test, numerous elastic letters, numbers, and symbols are continuously poured into a toy playground. Four workstations ($4 \times 1 \times 1$) are adopted for computation.

4.7.5 Single-machine Performance

For completeness, we also compare our distributed MPM pipeline with the state-of-the-art (SOTA) C++ MPM simulation pipeline implementations that are heavily hand-optimized for single architectures: CPU (Wolper et al., 2019) and GPU (Gao et al., 2018b). As the test example, we use a simple scene, an elastic box falling down to the ground with a grid resolution of $256 \times 256 \times 256$. Table 4.3 summarizes the particle number, testing device, and timing results. Indeed, our distributed system cannot outperform the separate CPU- and GPU-implements, which have CPU-tailored SPGrid (Setaluri et al., 2014) and CUDA kernel optimizations (Gao et al., 2018b), but is near 50% performance for both. Nevertheless, our target is a generalized, distributed, and scalable system that can be executed on most HPC platforms without code modification, while the compared implementations have adopted sophisticated hardware-specific code optimizations focusing on their specialized single-machine platforms.

4.8 Conclusion and Discussion

We proposed a distributed simulation framework specialized for MPM computations. Based on the hierarchical system architecture design, we make it possible to achieve

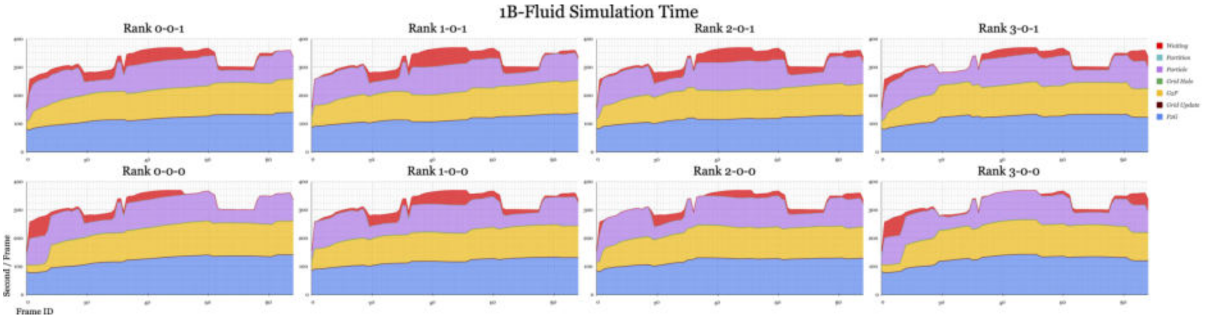


Figure 4.19: **1B-Fluid.** Detailed timing per frame (in seconds).

multiple advancements in each layer. The *programming model layer* uses `Kokkos` to enable fast switching between various latent devices and dispatch the MPM pipeline to many major HPC platforms. Additionally, *particle/grid data, algorithms, and communication layer* with our dedicated distributed sparse grid design makes it simpler for hybrid simulation mechanics. Furthermore, we propose the dynamic load balancing algorithm to improve overall performance. Multiple experiments and comparisons are conducted to demonstrate the effectiveness and serve as a reference for simulation setups. Finally, we demonstrated that the proposed distributed MPM system can handle extremely large-scale simulations of complex elastoplastic materials with more than 1 billion particles, which has never been achieved before in computer graphics or computational mechanics.

Limitations and Future Work. First, there is still space to improve communication efficiency. We adopt `MPI_Isend`/`MPI_Irecv` for data transmission among MPI ranks. However, better communication scaling could be achieved if more efficient MPI interfaces are studied and explored. Second, as discussed in §4.7, the initial settings of MPI topology and dynamic load balancing frequency will strongly influence the overall performance. Making this decision process automatic according to initial particle samples is a valuable direction to explore. Furthermore, the workload computation in dynamic load balancing can be improved. Grid or particle alone is both insufficient to form a perfect workload description to fit all general simulation scenes. Therefore, combining these two pieces of information and conducting better representation could provide more benefits. Finally, it is also meaningful to further improve the performance for individual parallel backends and within the *particle/grid* layers on specific devices to support faster application systems. Indeed, because the simulation framework is built on `Kokkos` and `Cabana`, testing and leveraging additional hardware architectures is straightforward. This includes AMD GPUs as deployed in the recent Frontier supercomputer, as well as future systems.

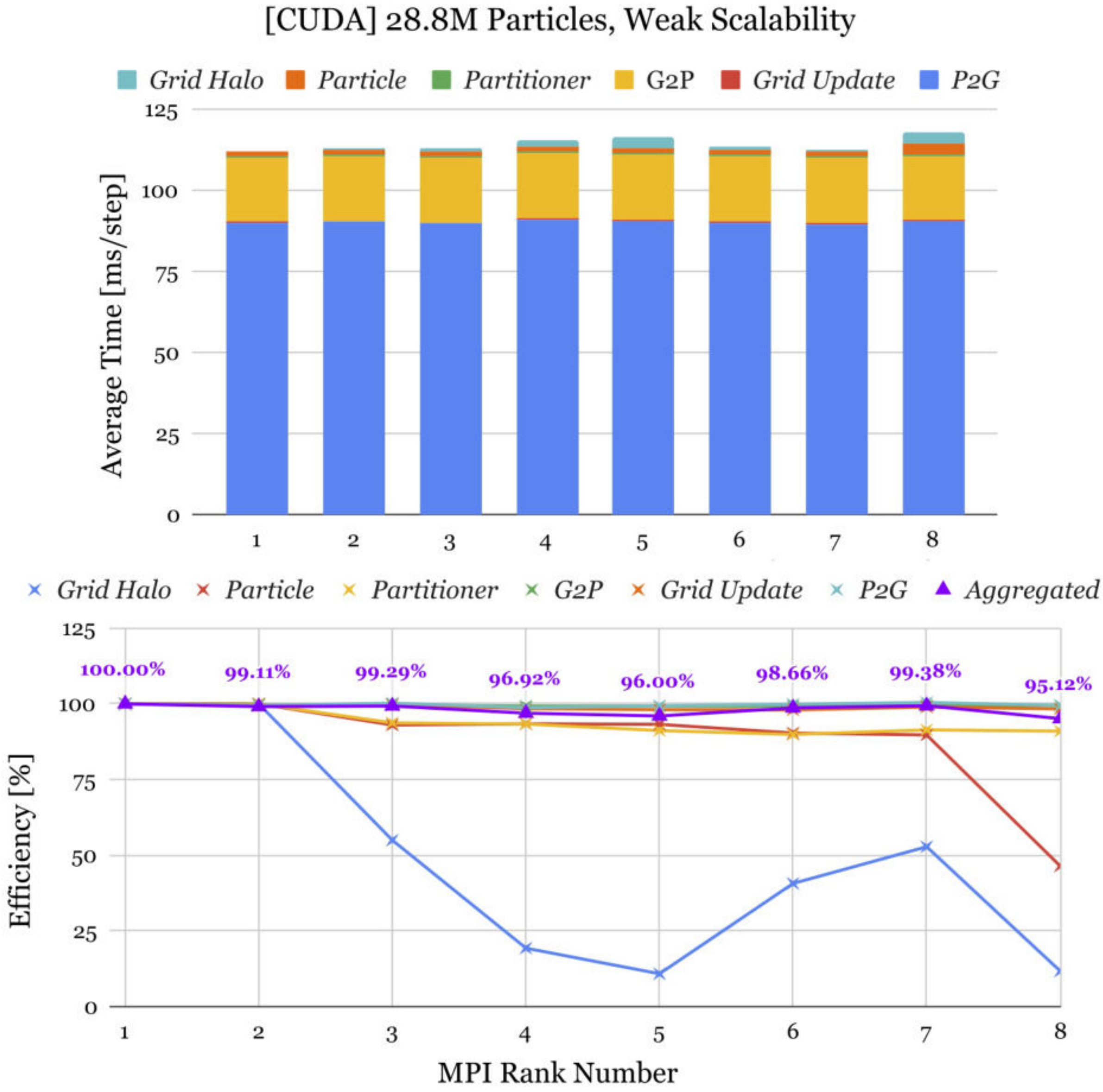


Figure 4.20: **Weak Scalability** on local workstations with GPU(CUDA). Here, *Particle* is short for *Particle Communication* kernel; and similarly, *Grid Halo* for *Grid Halo Communication* and *Partitioner* for *Dynamic Partition Optimization*. The listed numbers in the lower figure are the efficiency values for aggregated timing (summation of all six kernels).

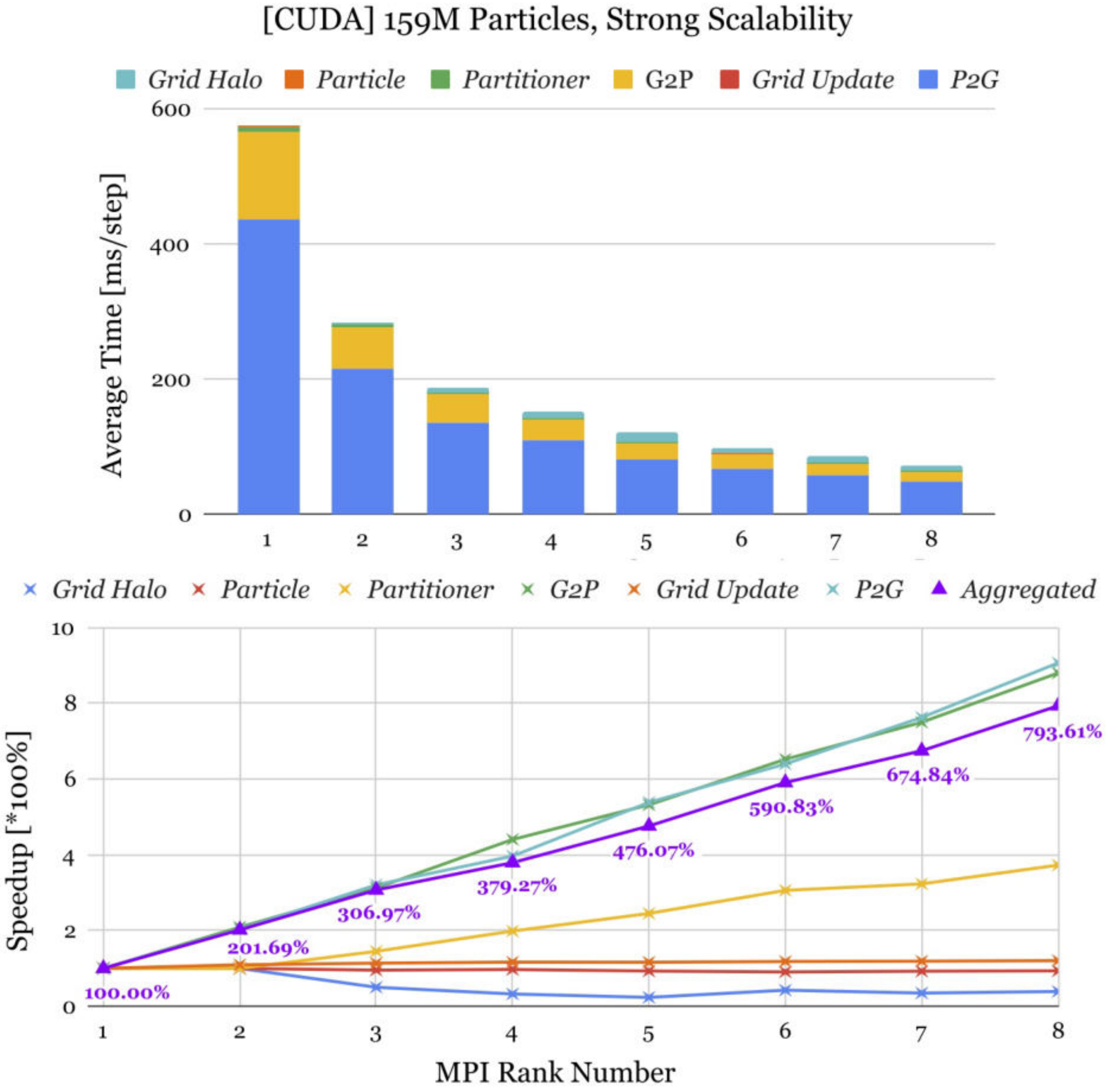


Figure 4.21: **Strong Scalability** on local workstations with GPU(CUDA). All ranks handle a huge elastic box with 159M particles.

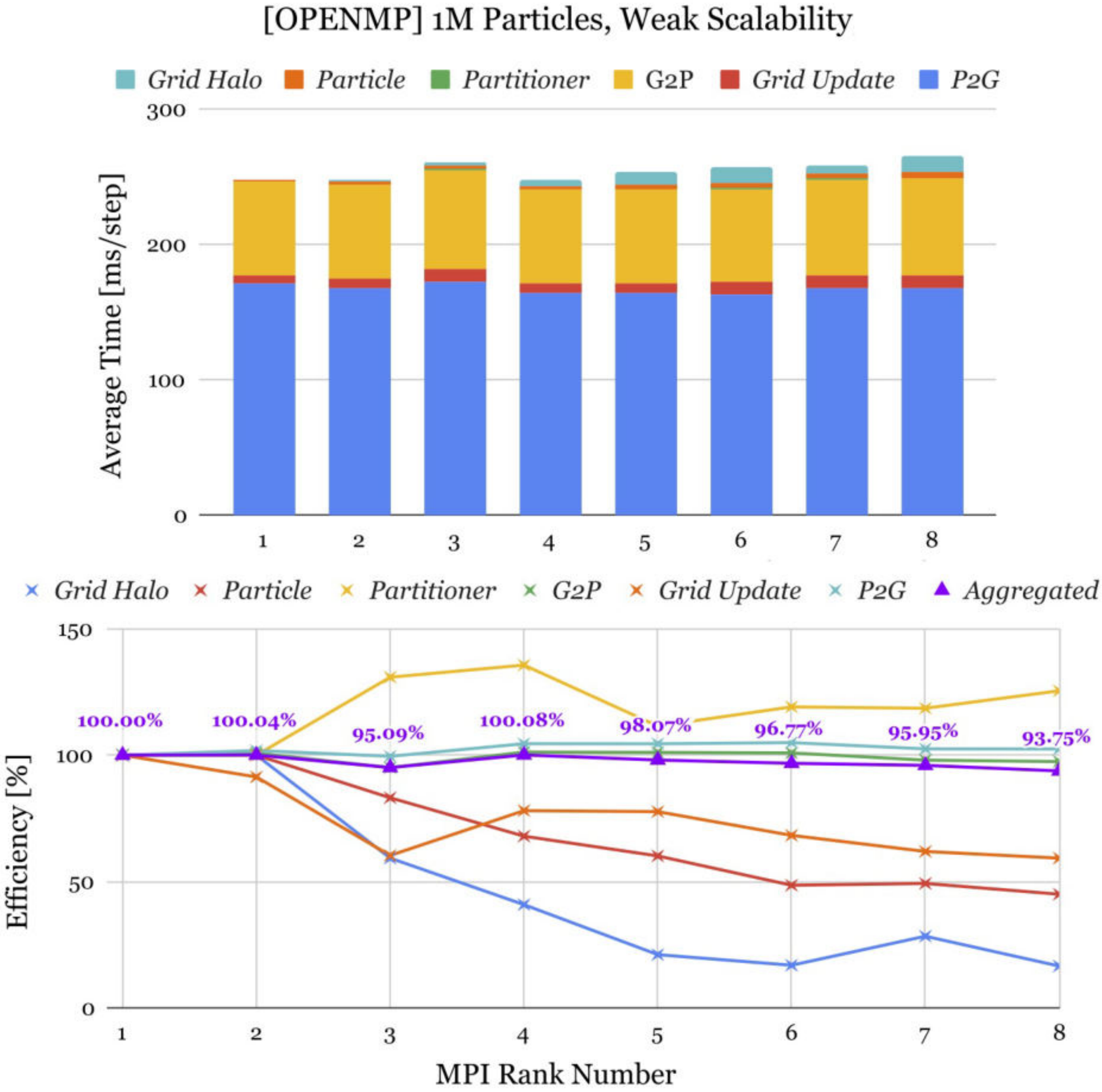


Figure 4.22: **Weak Scalability** on local workstations with CPU (OpenMP). Each rank handles an elastic box with 1M particles.

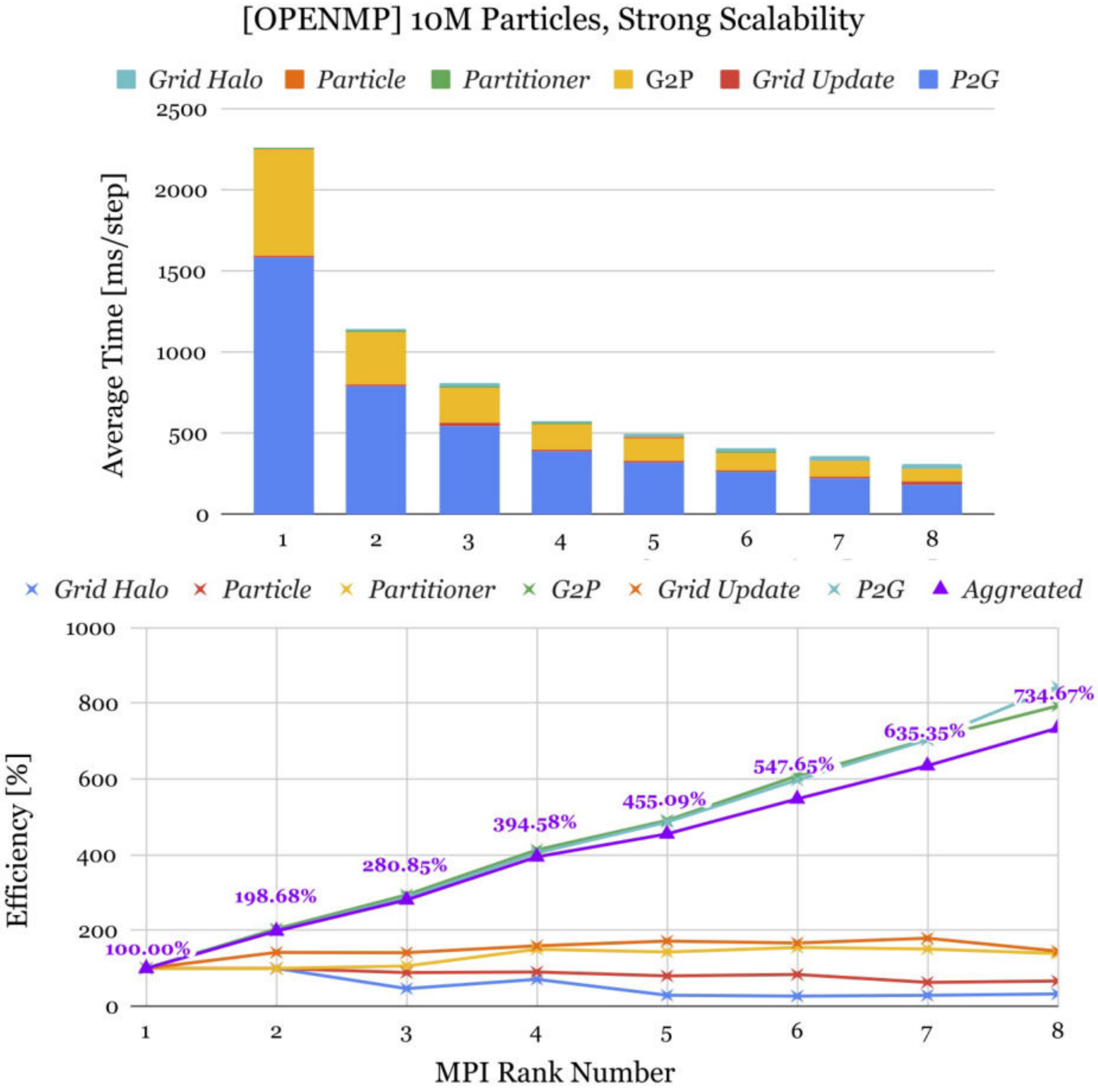


Figure 4.23: **Strong Scalability** on local workstations with CPU(OpenMP). All ranks handle a huge elastic box with 10M particles.

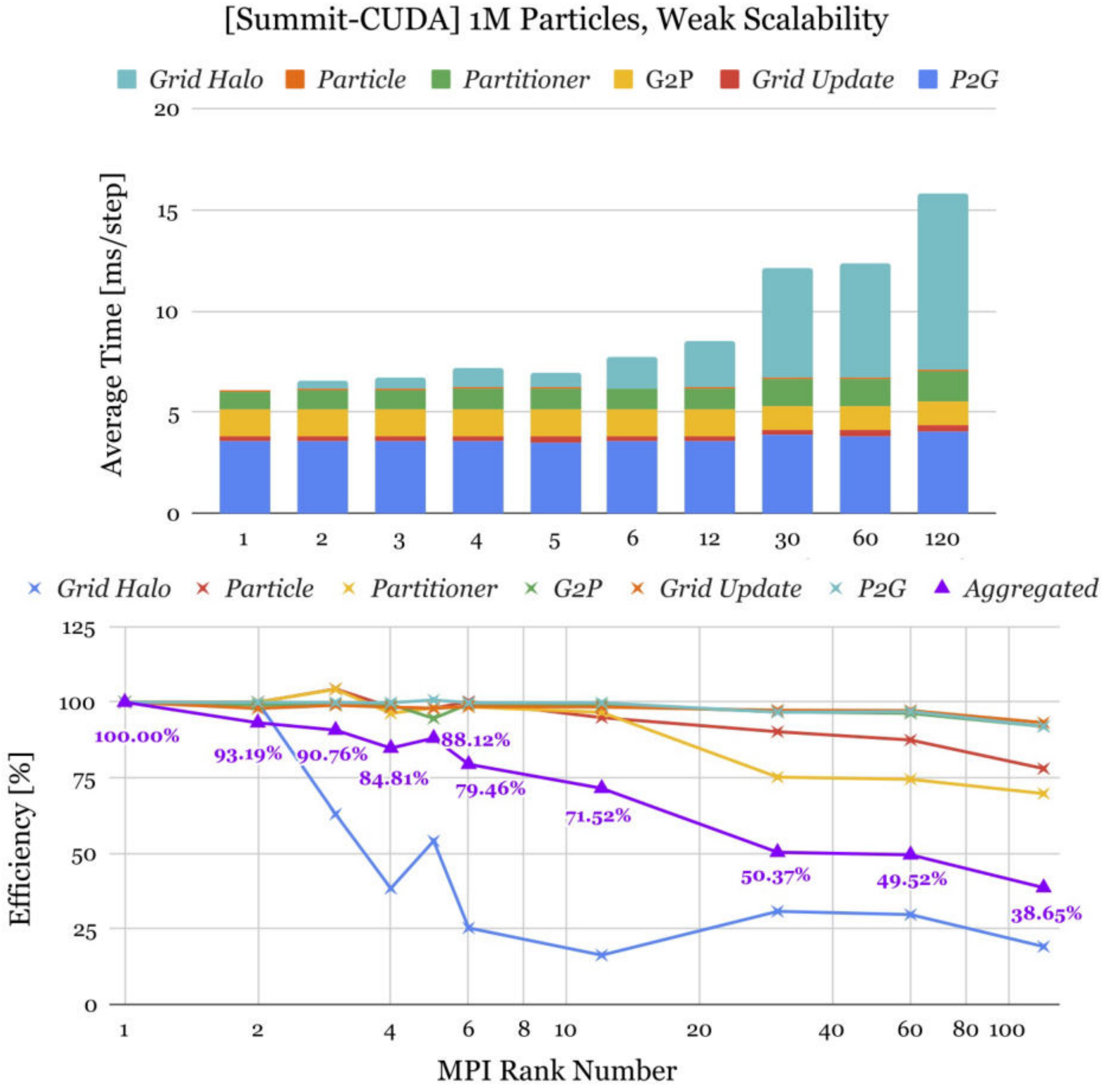


Figure 4.24: **Weak Scalability** on **Summit** with GPU(CUDA). Each rank handles an elastic box with 1M particles.

CHAPTER 5

Real-Time Simulation Applications

5.1 A Virtual Testbed for Physical and Interactive AI

We propose VRGym, a virtual reality (VR) testbed for realistic human-robot interaction. Different from existing toolkits and VR environments, the VRGym emphasizes on building and training both physical and interactive agents for robotics, machine learning, and cognitive science. VRGym leverages mechanisms that can generate diverse 3D scenes with high realism through physics-based simulation. We demonstrate that VRGym is able to (i) collect human interactions and fine manipulations, (ii) accommodate various robots with a ROS bridge, (iii) support experiments for human-robot interaction, and (iv) provide toolkits for training the state-of-the-art machine learning algorithms. We hope VRGym can help to advance general-purpose robotics and machine learning agents, as well as assisting human studies in the field of cognitive science.

5.1.1 Introduction

The past decade has witnessed a rapid development of categorical classification for objects, scenes, and actions, fueled by large datasets and benchmarks, discriminative features, and machine learning methods. Similarly, successes have also been achieved in many other domain-specific tasks, largely due to the ever-growing vast amount of labeled data and rapidly increasing computing power, combined with supervised learning methods (in particular, deep learning (Hinton and Salakhutdinov, 2006)). The performance of certain tasks has reached a remarkable level, even arguably better than human in control (Duan et al., 2016; Mnih et al., 2015), grasp (Mahler et al., 2017; Lenz et al., 2015), object recognition (He et al., 2015), learning from demonstration (LfD) (Argall et al., 2009), and playing the game of go (Silver et al., 2016) and poker (Moravčík et al., 2017; Brown and Sandholm, 2018).

Despite the impressive progress, these data-driven feed-forward classification methods have well-known limitations, hindering the advancement towards a more general Artificial Intelligence (AI) that can interact with human: (i) needing *large labeled training datasets*; (ii) often *task-specific* and view-dependent, which makes it difficult to generalize; (iii) lacking an *explicit representation and structure* to handle large variations exhibited in and outside of the training data.

In contrast, the hallmark of machine intelligence is the capability to rapidly adapt to new tasks and "achieve goals in a wide range of environments (Legg and Hutter, 2007)". To achieve such intelligence, recent years have seen the increasing use of synthetic data and simulation platforms. Advantages include: (i) the structure of the data is efficiently encoded *without the need for human labeling* as the simulation inherently comes with the ground truth; (ii) can accommodate different embodied agents (*e.g.*, humans, humanoid robots, or turtle-bots); and (iii) benchmark *generalization* in various tasks at a low cost.

Empowered by the gaming industries, tremendous amount of game contents, including scenes and objects, are made available for the virtual environment. Meanwhile, more sophisticated physics-based simulation engines and rendering techniques have enabled more realistic simulations. These characteristics allow a growing number of tasks to be performed using synthetic data in simulation platforms. Furthermore, some simulation platforms also become publicly available, such as AirSim (Shah et al., 2018b), AI2THOR (Kolve et al., 2017), Gibson (Xia et al., 2018), *etc.*, promoting the further explorations and applications. In short, it is both the research and the engineering efforts that make it possible to achieve considerable successes in some AI tasks and applications.

However, prior work often lacks the human involvement, especially in high-level tasks. For instance, although some virtual platforms (*e.g.*, OpenAI Gym (Brockman et al., 2016) and Mujoco (Todorov et al., 2012)) allow to train a virtual robot to perform many

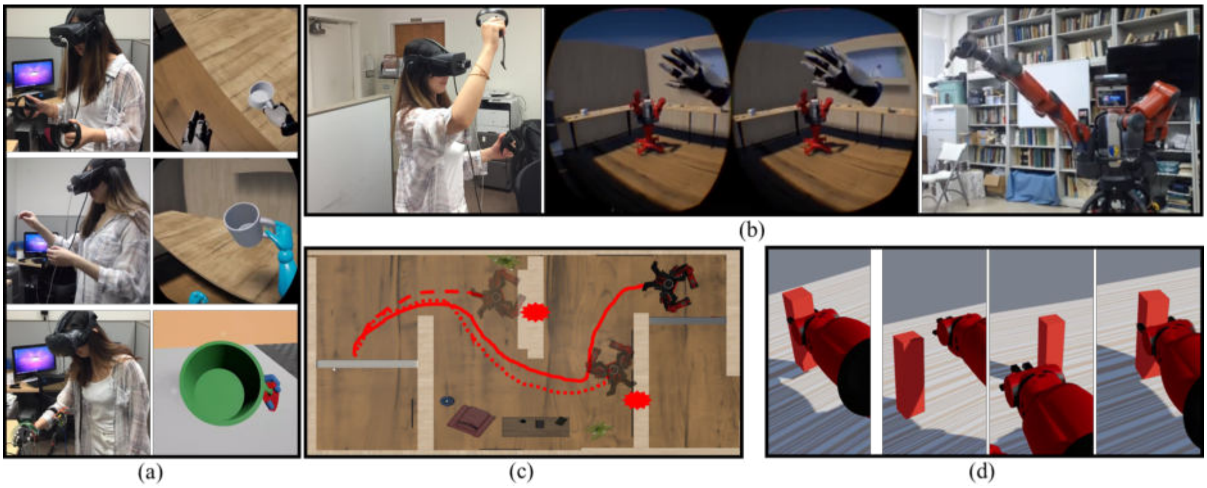


Figure 5.1: (a) VRGym integrates three types of input devices, providing human manipulation in an increasing resolution using Oculus Touch, LeapMotion, and a data glove, from top to bottom. (b) The VRGym-ROS bridge allows physical human/robot agent meet virtual agents inside a virtual world, providing the capability of social interactions. (c) The training of the robot navigation using reinforcement learning (RL) inside VRGym. The robot successfully navigates to the goal without collisions after about 10,000 episodes. (d) The learning of object manipulation using human demonstrations (leftmost) and inverse reinforcement learning (IRL) (right three) inside VRGym.

manipulation tasks, they lack a human in-the-loop, thus cannot handle critical tasks like intention prediction and social interaction. Hence, having a simulation environment where a robot can interact realistically with a human and evolve incrementally could facilitate the robotics developments.

In this section, we propose VRGym—a virtual reality testbed, which combines VR with virtual training for both physical and interactive AI agents. By putting human in-the-loop, VRGym goes beyond the traditional synthetic data and simulation platforms by simulating a human-robot co-existing environment.

Specifically, VRGym tries to fill in the gap between the new advancement of VR and the need for training virtual agents to collaborate with human. In particular, we hope to address three critical issues. First, what is the best way to reflect human embodiment in VR; *i.e.*, how humans can genuinely interact with robots and how the robots can perceive related data that are sufficiently close to those in real life? Second, how to take advantages of current well-developed algorithms and models? Third, to which

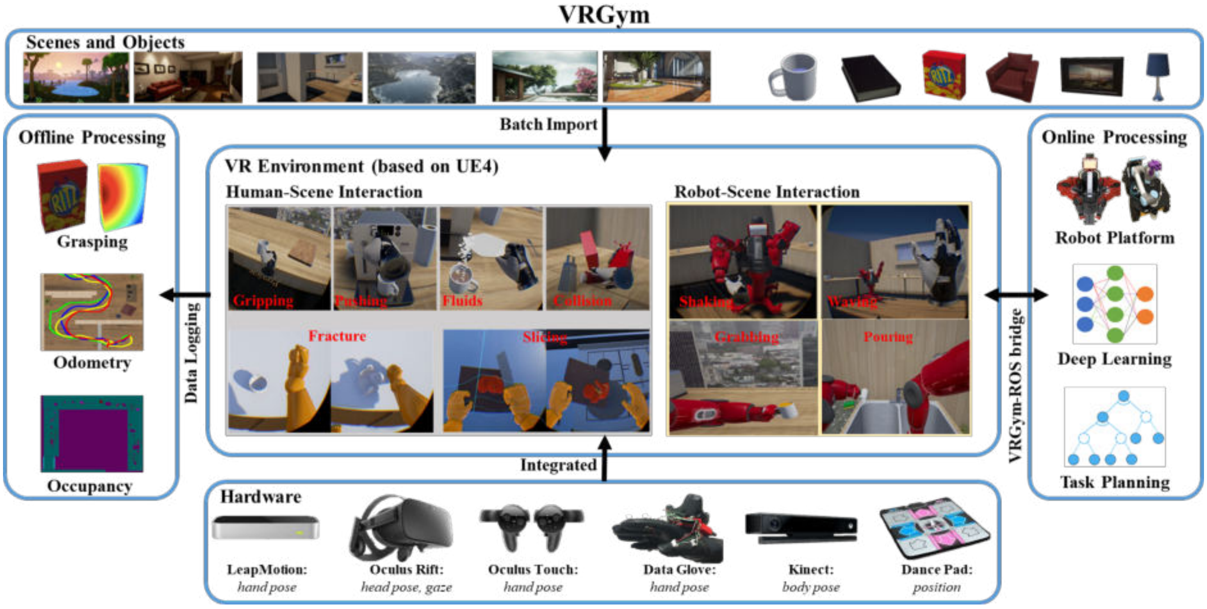


Figure 5.2: System architecture of VRGym, consist of three major components: (i) Hardware modules for human data input. (ii) Scene modules batch import various category of scenes as well as diverse objects, derived from different resources such as 3D modeling tools, scanned models, and automatically generated synthetic data. (iii) VR environment serves as an ideal testbed, where both a human and a robot can perform diverse tasks. The inherent physics-simulation engine enables realistic human-scene interactions and robot-scene interactions.

level of unique interactions the VR simulations can afford? To answer these questions, VRGym is designed to push the limits of current akin simulators by offering the following characteristics.

Fine-grained human embodiment representation Adding a real human in the simulation is not a trivial task. Most of the current simulation platforms only support either scripted or limited remote-controlled human models. In VRGym, we integrate a multi-sensor setup as alternatives to traditional VR input devices. Our setup is capable of providing a whole-body sensing and reflecting the measured data on a detailed human avatar. As a result, the simulation can account for both body and hand poses during interactions. Figure 5.1a shows different resolutions of manipulations in VRGym.

High compatibility with existing robotics systems and algorithms In VRGym, we build an efficient bi-directional communication interface with the Robot Operating System (ROS). Figure 5.1b depicts an example of how a person interacts with a robot in VRGym, supported by the VRGym-ROS bridge. As a result, all ROS-compatible resources can be used in VRGym with little effort, which allows easy setups, training, evaluations, and benchmark.

Multiple levels of interactions By providing the fine-grained human embodiment representation and the ROS integration, various interactions between humans and autonomous agents are made possible in different resolutions. VRGym supports interactions as simple as only providing visual/perception information and as sophisticated as learning complex robot grasping from human demonstrations. Figure 5.1c shows how an agent obtains a navigation policy using RL, and Figure 5.1d shows learning a grasp policy using IRL.

VRGym makes the three contributions:

- A comprehensive simulation platform that integrates UE4 built-in functions, *e.g.*, scene, physics-based simulation, rendering, basic human inputs, with customized developments, aiming to facilitate a variety of AI researches.
- A multi-sensor hardware and software setup that allows the whole body sensing and reflects human subjects to virtual embodiments with great details. The generated data can be seamlessly logged for online and offline training purposes.
- VRGym-ROS bridge enables a bi-directional data communication. Through this interface, AI researchers can take advantages of the existing robotics models and algorithms. Similarly, robotics researchers can utilize more sophisticated physics-based simulation.

5.1.2 VRGym System Architecture

Figure 5.2 illustrates the system architecture of the VRGym. VRGym offers a variety of realistic scenes and tasks for both humans and robots, and provide automatic logging of the data during agents performing tasks. This capability is provided by the integration of three main modules: (i) scene module which renders user-specified 3D scenes and objects, (ii) VR environment based on UE4 with physics-simulation engine, introducing various physical properties that enrich tasks and data, and (iii) VR hardware module that imports a human agent's state and command to the VRGym. We now further elaborate each module in the following subsections.

5.1.2.1 Scene Module

Scenes and objects are the building blocks for a simulation environment. In order to increase the variety of environments for VRGym, we develop several pipelines to import or create scenes into VRGym based on the users' specifications. The scene module enriches static environments for VRGym. Note that the ground truth of RGB image, depth image, surface normal, and object label come automatically with the scene module in real-time, enabling the training for machine learning models and robotics applications.

Specifically, VRGym can directly import the entire 3D scenes provided in large open-source datasets, either collected from the web (Song et al., 2017; Chang et al., 2017) or automatically generated from a given set of objects (Yu et al., 2011; Qi et al., 2018; Jiang et al., 2018) (see top of Figure 5.2). Additionally, VRGym also supports manually constructed scenes (see Figure 5.4) for more specific tasks, where neither the open-source scene dataset or the automatically generated scenes could satisfy such constraints.

Similarly, individual objects can be imported to VRGym from mesh files, which can be obtained from open-source CAD datasets (*e.g.*, (Chang et al., 2015; Calli et al., 2015)). Customized or complex objects can be manually created or scanned using a RGB-D sensor to import to VRGym for specific tasks. After the import, users can further adjust static meshes, textures, materials, and collision boundaries of the objects.

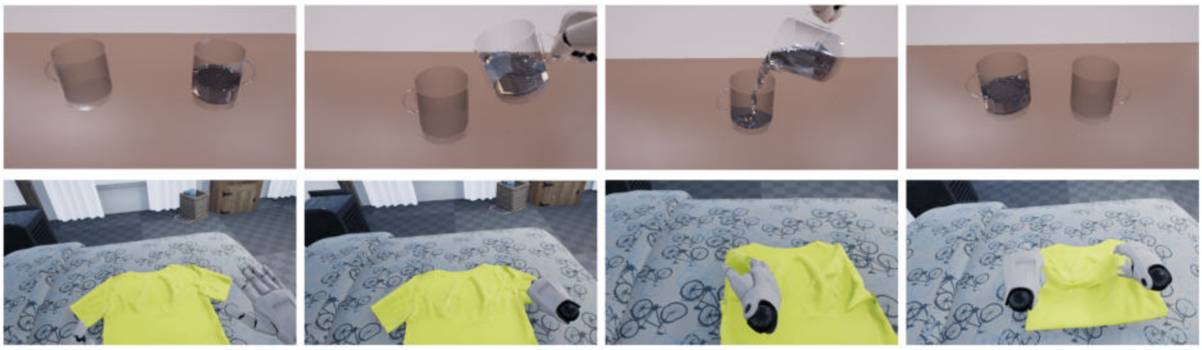


Figure 5.3: Examples of various physics-based simulation for diverse tasks in VRGym beyond merely rigid-body simulation in other 3D virtual environments. (Top) Pouring water. (Bottom) Folding clothes.

5.1.2.2 Real-time Physics-Based Simulation

We choose UE4 as the simulation engine for VRGym for its advanced real-time physics-based simulation. Unlike previous 3D virtual environments that mostly focus on rigid body simulation or symbolic-level event simulation, VRGym integrates the advanced simulation provided by UE4 to enable a large set of various simulations, including rigid body, soft body, collision, fluid, cloth, slicing, and fracture. Some examples are shown in Figure 5.3 and the center of Figure 5.2. As a result, subtle object state or fluent (Newton and Colson, 1736) changes due to the virtual agent's actions are realistic and diversify. Integrating with such sophisticated physics-based simulations, VRGym not only increases the task complexity and improves the visual experience of human agents, but also affords more complicated task simulations for both virtual and physical robots.

5.1.3 Human Embodiment in VRGym

Compared to other similar 3D virtual environments, VRGym has another distinct feature; *i.e.*, introducing the capability to represent the physical human agent's embodiment in real-time as an avatar in the virtual environment. To reflect human movements and manipulations accurately, the physical human agent is tracked in real-time, resulting in a humanoid mesh that can deform accordingly based on the underlying tracked body skeleton and the hand poses.

Specifically, the setup includes: (i) A Kinect One RGB-D sensor to map human skeleton to the avatar in real-time through a customized-built Kinect plugin developed in UE4, (ii) an Oculus headset to record the head pose, (iii) a dance pad to navigate the avatar inside a large virtual world, and (iv) three types of input devices that provide manipulation information in different resolutions. Compared to other platforms, VRGym emphasizes the capability for users to interact with virtual environments. Depending on the needs, the user can use one of the three input devices for manipulation:

- Oculus Touch Controller offers an attachment-based approach; *i.e.*, the virtual object will automatically attach to the virtual controller/hand once the user triggers the grasp event. It enables a firm-grip manipulation, providing a firm but the least realistic grasp during the human-object interaction. Such manipulation is effective in the event-level tasks where the fine-grained hand pose is not required; *e.g.*, pick and place.
- The commercial hand pose sensing products (*e.g.*, LeapMotion) provide the vision-based gesture recognition. It is a low-cost and off-the-shelf solution that can be easily set up by mounting the sensor on the head-mounted display. However, it is difficult to have a firm grasp due to occlusions and sensor noises. Note that the hand tracking will fail if the hand is not within the view.
- An open-sourced glove-based device (Liu et al., 2019b) is also compatible with VRGym to provide the finest-grained manipulation. It requires a Vive Tracker to provide global positioning of the hand, and an IMU network in the glove to measure the rotation of each phalanx and calculate the hand poses using forward kinematics. Although glove-based devices are costly compared to other alternatives, they allow reliable hand pose sensing, which is vital for the tasks with detailed, complex and subtle hand manipulations.

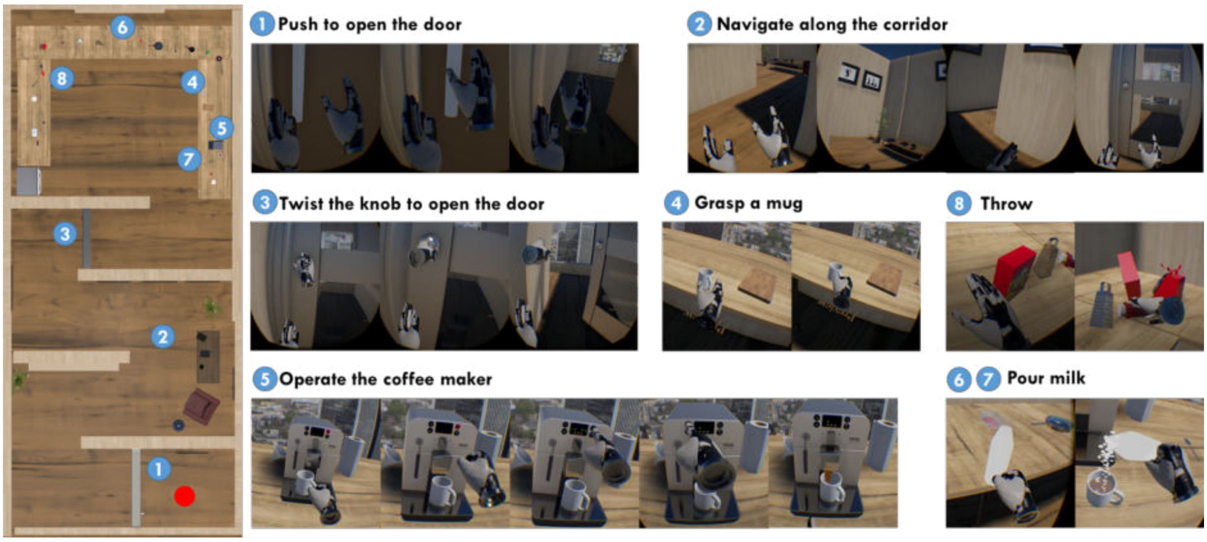

1 Push to open the door
2 Navigate along the corridor
3 Twist the knob to open the door
4 Grasp a mug
8 Throw
5 Operate the coffee maker
6 7 Pour milk
6
8
4
5
7
3
2
1

Figure 5.4: A human agent performs a series of actions in a virtual scene using Oculus Touch controllers. (Left) Action sequence from a top view of a virtual indoor environment. (Right) Sequences of the performed actions. Specifically, the human agent starts at the red dot as shown in the left, (1) pushes a door, (2) navigates along the hall, (3) twists a door to enter the kitchen, and (4)-(7) makes a cup of coffee. This process involves (i) large movements using the human embodiment provided in VRGym (navigating along the hallway), (ii) complex operations (operating the coffee maker), (iii) fine-grained manipulations (twisting the doorknob), and (iv) physics-rich controls (pouring milk).

5.1.4 Software Interface Design

VRGym has two major software interfaces developed to enable training and benchmarking both physical and interactive AI agents. The first interface is the human data logging system that builds on top of the hardware setups to collect the data generated during the interactions between the avatar and the environment. Another interface, a VRGym-ROS bridge, is introduced to allow seamlessly import of robot models and robotics algorithms from ROS. The collected data together with the VRGym-ROS bridge could be used for a variety of AI applications; see examples in subsection 5.1.5.

To demonstrate the functions of these two interfaces, we consider a task-rich environment built for the VRGym. Figure 5.4 depicts an environment in VRGym that provides semantically-diverse tasks to the agent. Note that although such environment could be constructed in the real world to perform the demonstrated tasks, sensing and logging the detailed data generated during the interactions between the agent and objects would be extremely difficult in practice.

In such a typical virtual environment in VRGym, an agent (a human as an avatar or a virtual robot) is initially placed on the starting point, indicated as the red dot in Figure 5.4. The final goal for the agent is to reach the kitchen located at the far-end, and accomplish several sub-tasks. At the beginning, the agent has to push to open the first door and navigate along the corridor, requiring *large movements*. Then the agent must go through another door to enter into the kitchen, and the only solution is to twist the doorknob using complex *manipulations*. Inside the kitchen, the agent is required to make a cup of coffee with milk, which needs to grasp and move a mug, operate the coffee maker by pushing several buttons in a certain order. The entire procedure requires the *task planning* empowered by the *physics-based simulation*.

5.1.4.1 Human Data Logging

When a user performs a task, data generated by the interactions between an avatar and the environment can be directly logged with ground-truth labels in VRGym. In this section, we showcase two scenarios where the data is logged and used in other applications.

Grasping Finer-grained manipulation is made feasible in VRGym using a glove-based device (Liu et al., 2019b); see Figure 5.5a for some results. By collecting a set of subjects' grasp data on a variety of objects, we can merge all the collected grasp data to form heat maps on different objects to visualize the likelihood of grasp points on man-made objects. Specifically, the grasp data shown in Figure 5.5b is the averaging data of heat maps collected from 10 human subjects, where the hotter the area is, the denser the grasp points are, and the more likely a human agent would grasp around that area.

Footprints VRGym provides the function to log an agent's footprints or the odometry data. Figure 5.5c shows the recorded odometry data from 5 human subjects who have limited VR experience. Each of the participants navigates from the starting point to the kitchen room along the corridor using Oculus Touch controllers.

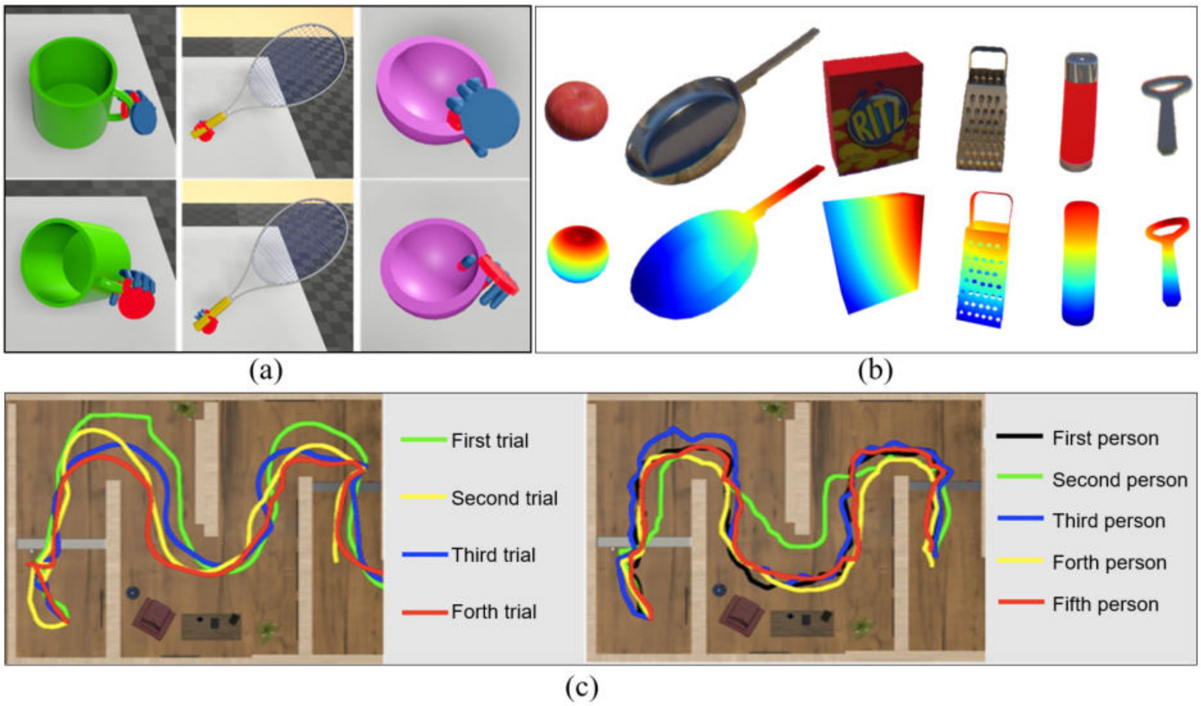
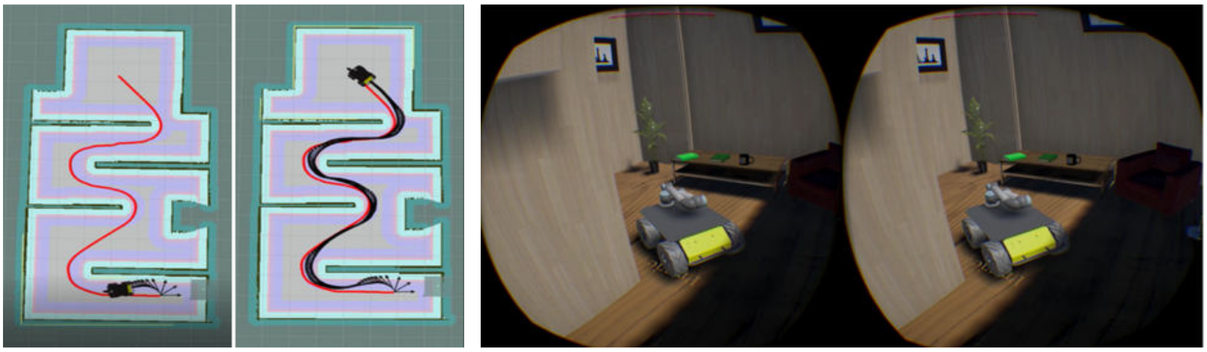


Figure 5.5: (a) Grasp a mug, a tennis racket, and a bowl. The red area indicates the contact force between the virtual hands and the object. (b) Visualization of the collected human grasp data. Top: a set of 3D objects. Bottom: the average grasp heat map generated by multiple subjects. (c) Visualization of footprint from different subjects.

5.1.4.2 ROS Interface

The VRGym is compatible with the popular ROS framework through a customized VRGym-ROS communication bridge. This bridge allows the off-the-shelf ROS robot models to communicate with the simulations and human agents in VRGym with minimal efforts; *e.g.*, the diverse scenes rendered in VRGym can also be exported to the Gazebo simulator, which is highly compatible with ROS.

Implementation We develop a ROS interface, VRGym-ROS bridge, based on the TCP/IP protocol in order to enable VRGym to communicate with the existing popular robotics platforms. Through this interface, robot body parts can be easily imported to VR environments as mesh files and control signals, and a data stream can be seamlessly transferred between the VRGym and the robot platforms using ROS to communicate with either physical or virtual robots. We organize all data types (*i.e.*, ROS topics) in a unified



(a) (b)

Figure 5.6: VRGym-ROS bridge. (a) The robot navigation in the scene imported into the Gazebo, exported from the VRGym. The red curve indicates the path planned by the robot's global planner. The black curve is the actual trajectory executed by the robot. (b) A Husky-UR5 robot is imported into VRGym from ROS to guide the way and open the door for a human agent.

JSON format and construct JSON parsers in both VRGym and ROS to further improve the compatibility. Each port in the protocol supports a stream of data, making it possible to present multiple agents from ROS into the VRGym. With the VRGym-ROS bridge, we present two examples of training and evaluating human robot interactions (HRI) inside VRGym in subsection 5.1.5, which incorporates direct human reactions and involvements. Such capability is largely missing in the current robotics simulators such as Gazebo or V-Rep. The benchmark in subsection 5.1.5 is also supported by this VRGym-ROS bridge.

Evaluation We evaluate the VRGym-ROS bridge on a navigation task (see Figure 5.4) using a Clearpath Husky robot. This navigation task is performed in VRGym, whereas the robot model is imported from ROS, making it possible to evaluate a number of SLAM algorithms and path planning approaches. In Figure 5.6a, the mapping result is obtained using the conventional GMapping package in ROS. The red curve indicates the planned path, whereas the black curve is the actual odometry of the Husky robot. Figure 5.6b shows the user's view when the robot is moving. This VRGym-ROS bridge fills in the gap between the diverse scenes in VRGym and the existing fine-tuned algorithms provided in ROS.

Communication Bandwidth To evaluate the reliability and efficiency of the VRGym-ROS bridge, we conduct an experiment by sending packages with the size of 512Kb[1].

5.1.5 Experiments

In this section, we demonstrate the performance and capability of the VRGym from four different perspectives[2]. Two human robot interaction (HRI) applications are conducted, including a human intention prediction task and a social interaction task. Like other testbeds, we also benchmark the performance popular machine learning algorithms (*e.g.*, reinforcement learning and IRL) in the VRGym.

5.1.5.1 Experiment 1: Intention Prediction

Predicting human intention is difficult when training on a physical robot since this task has very small error tolerance; wrong predictions may endanger both the human and the robot. It is particularly interesting to study human intention prediction in VRGym, since this problem involves complicated inference process that many types of data can be useful: human trajectories, human poses, object positions, object states, and first/third-person vision inputs, *etc.* Predicting intention is made possible in VRGym as our unique multi-sensor setup reflects human poses, and the odometry data provided by the data logging system indicates human's trajectories.

In the experiment, we analyze different human intention prediction algorithms to demonstrate the potential of VRGym as a testbed for both physical and virtual AI agents. Additionally, we show the unification of both the learning and the inference enabled by the VRGym. 20 subjects are recruited. The virtual environment is set up as a virtual kitchen, in which more than 20 objects are placed on top of three long tables. The layout of the kitchen is shown in Figure 5.7, where the agent starts from the entrance of the room (red dot) and performs the task with at least 4 steps: grasp a mug, operate the coffee maker, add milk, and add sugar. Note these tasks can perform in different orders.

[1] See a detailed evaluation in supplementary.

[2] See a video demo at Vimeo.

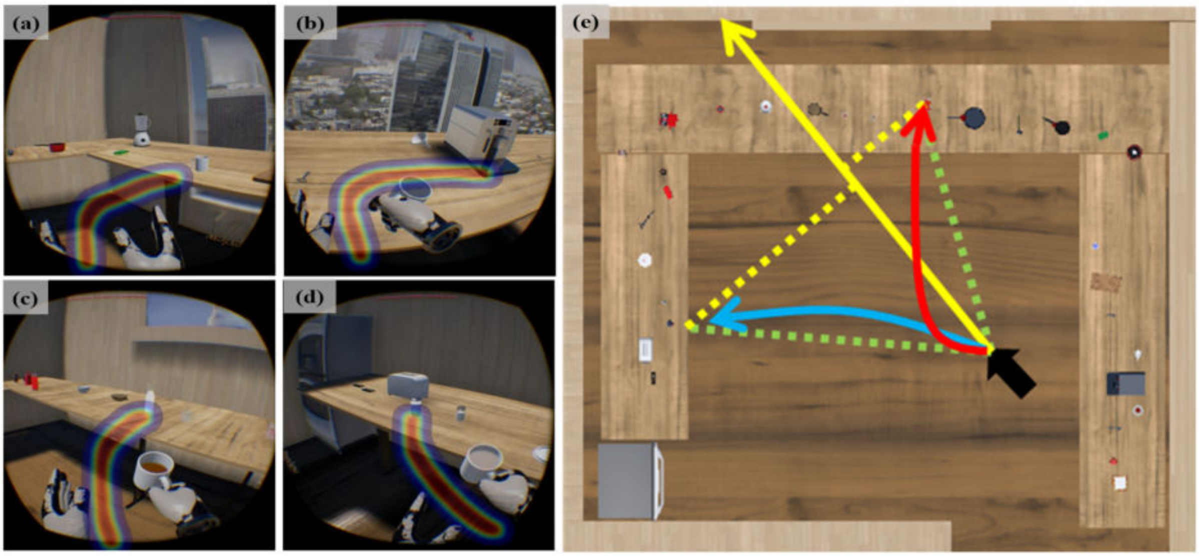


Figure 5.7: Intention predictions in a coffee-making task. (a) Grab a cup. (b) Use the coffee machine. (c) Pour milk. (d) Add sugar. (Right) Visualization of three intention prediction algorithms. Blue and Red: sampled paths from the grammar model (Qi et al., 2017). Green: straight-line distance. Yellow: prediction by shortest perpendicular distance (dashed lines) from objects to the ray direction (solid arrow) based on avatar's location.

The resulting footprint from one subject is plotted in Figure 5.7. All subjects are required to perform a coffee-making task—making a cup of coffee using the available objects.

Figure 5.7e illustrates the comparisons among these three methods. The qualitative results are shown in Figure 5.7a-d to reveal the intention of the agent as the heat maps during the process of making coffee, where hotter color (red) indicates higher probability. This high-level semantic prediction is inferred given multiple human demonstrations as logged navigation and grasp data collected from the agent using VRGym.

5.1.5.2 Experiment 2: Social Interaction

Social interactions or social HRI is a vital topic enabling human-robot co-existing environment, since the robot needs to understand and respond properly to human's social behaviors, such as waving and hand-shaking. Although the current robot simulators (*e.g.*, Gazebo and V-Rep) provide a suite of features, one key element these simulation platforms still largely missing is direct human involvement which is crucial for human-robot interaction studies.

Participants A total of 10 subjects were recruited. We implemented the algorithm proposed in (Shu et al., 2017) for robot learning social affordance. The algorithm is briefly described as follows; we refer the readers to the original paper for more technical details.

Results Qualitative results are shown in Figure 5.8. Concretely, the robot starts hand-waving in response to the agent's hand waving (Figure 5.8a), illustrated by a virtual hand model. The robot stretches out its manipulator to make a handshake with the virtual agent (Figure 5.8b). Technically, when the virtual Baxter inside the VRGym perceives the action signals from a virtual human such as *hand shaking* or *hand stretching out*, it sends the action signals to ROS through the VRGym-ROS bridge. In ROS, the motion planning will generate corresponding body parts transformations and send the computed transformation data back to the virtual Baxter inside VRGym, such that it will then act with the appropriate responses to the virtual human agent. In this sense, the proposed VRGym enables a new approach to study social human-robot interaction without using a costly physical robot or having a physical contact between a subject and robots, which in some cases could be dangerous.

5.1.5.3 Experiment 3: RL Algorithms Benchmark

We introduce a playground as a sub-module (Figure 5.9) inside the VRGym, aiming to train robots to navigate in a 3D maze-like indoor corridor. The overall goal is to teach the robot agent itself by trial and error to obtain a navigation policy, reaching the final goal of the maze. The learning strategy applied on the virtual robot follows the standard RL framework. A Baxter robot is integrated into the VRGym and controlled by off-the-shelf ROS packages.

Compared to other virtual playgrounds (*e.g.*, OpenAI Gym), the proposed VRGym differs in two primary aspects.

- *Sophisticated Interactions.* With the advanced physics-based simulator, the VRGym offers realistic interactions between the virtual agent and the virtual environment.

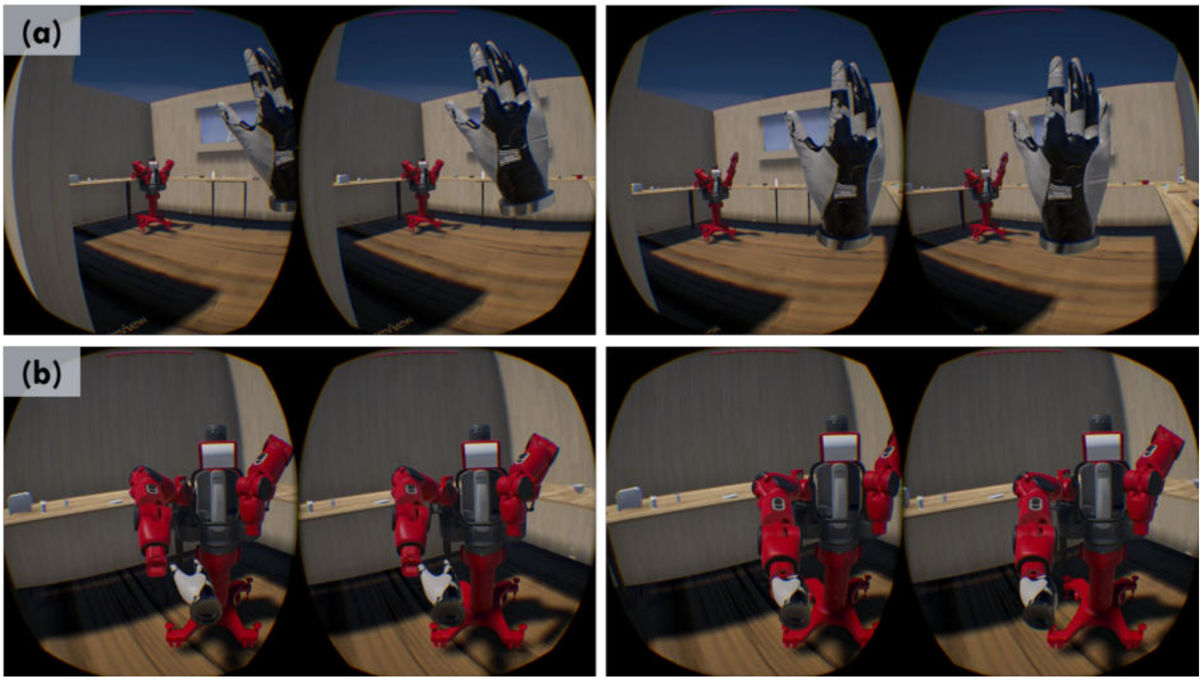


Figure 5.8: Human robot interactions in VRGym. A Baxter robot (a) waves hands and (b) shake hands with a virtual human agent.

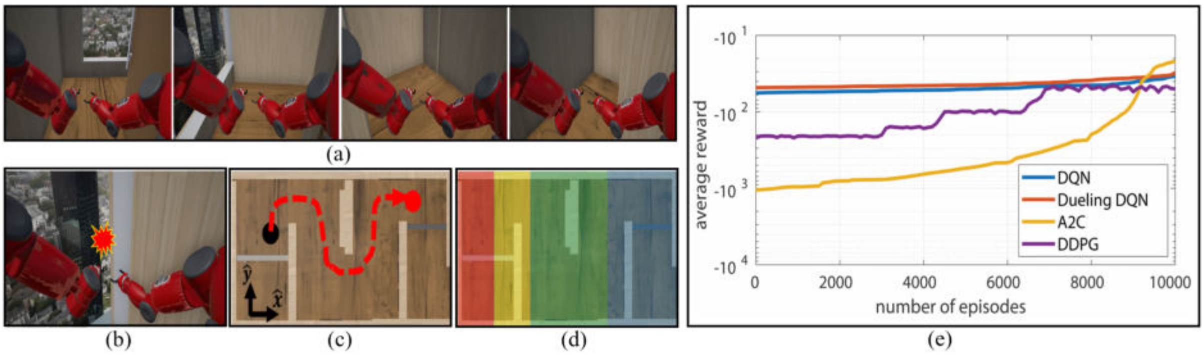


Figure 5.9: Settings for the RL training inside VRGym environment for an indoor maze navigation task. (a) First-person view of a virtual robot. (b) The robot collides with a wall, triggering negative rewards. (c) An eagle view of the indoor navigation task. (d) Rewards assigned in different color zones (red, yellow, green and blue) from low to high. (e) The performances of the tested RL algorithms.

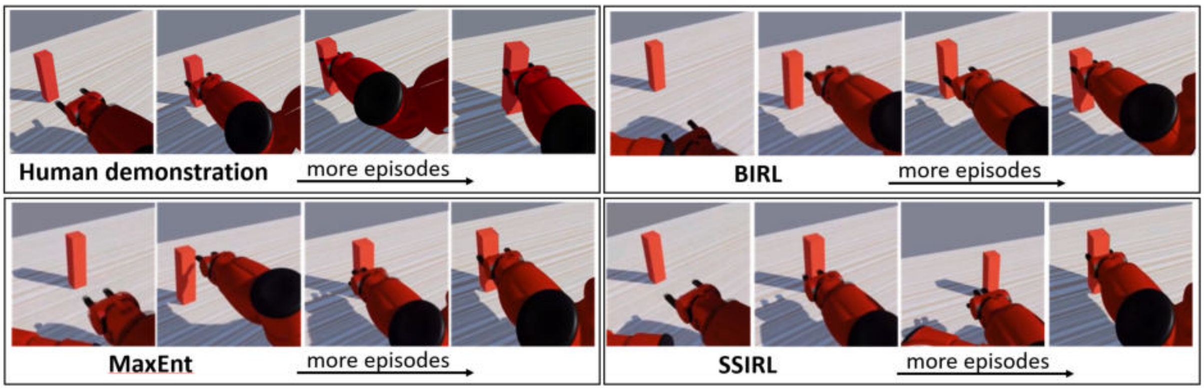


Figure 5.10: Learning human grasping demonstration with different IRL frameworks.

- *Physical RL Agent.* Since the VRGym is capable of importing both the physical and the virtual robot model to the virtual scene, it is feasible to transfer RL model trained inside the virtual environment directly to a physical robot agent.

We conduct four state-of-the-art deep RL algorithms to demonstrate the VRGym's capability in RL related tasks. These algorithms are **DDPG** (Lillicrap et al., 2016), **DQN** (Mnih et al., 2015), **Actor-Critic** (Mnih et al., 2016), and **Dueling DQN** (Wang et al., 2016). All four algorithms use the pixel-input from the first-person camera view. The quantitative comparison of the above four algorithms in VRGym is plotted in Figure 5.9e.

5.1.5.4 Experiment 4: IRL for Learning Grasp

Grasp is an imperative capability for an interactive agent. In this experiment, we adopt an inverse reinforcement learning (IRL) framework to enable a virtual robot learning to grasp from human demonstrations. This task primarily involves both the data logging function in VRGym and a ROS motion planer communicated by the VRGym-ROS bridge. The robot is expected to learn how to successfully grasp an unknown object based on the hand trajectories demonstrated by the human subjects, collected through tele-operations using the Oculus Touch Controller inside the VRGym.

The trajectories of the human demonstrations are logged and used to infer the model and its parameters. Later, with the learned model and its parameters, the robot can be executed using the motion planner in ROS to grasp an unknown objects in the virtual environment.

Three IRL algorithms are implemented in the VRGym: Bayesian-IRL (Ramachandran and Amir, 2007), Maximum Entropy-IRL (Ziebart et al., 2008), and Semi-supervised-IRL (Valko et al., 2012). Qualitative results are shown in Figure 5.10.

5.1.6 Conclusion

In this section, we introduce the VRGym as a promising simulation platform for training and evaluating autonomous agents to build the physical and interactive AI. VRGym can represent a fine-grained human embodiment as a virtual avatar using a range of hardware setups for body and manipulation sensing. Existing robotics systems and algorithms developed in ROS can also be integrated to VRGym through a customized VRGym-ROS bridge. Multiple evaluations indicate that the VRGym has a robust performance at the system level and in the communication with ROS. Our experiments have demonstrated that four different robotics interactive tasks can be successfully trained using RL and IRL inside VRGym. Specifically, we showcase how the data logged from the VRGym is useful in several interaction tasks, combining with the functions (*e.g.*, motion planners, robot models) provided by ROS through the VRGym-ROS bridge. The successful implements of RL and IRL for robotics interactive tasks in VRGym also support the training capability offered by VRGym in training robots with advanced machine learning methods. We believe VRGym could have further potential applications and it will benefit future research on the physical and interactive AI.

5.2 Real-time Material Point Method in Unreal Engine

The prior subsection highlighted the significance of real-time physics-based simulations in VR environments, both for interactive tasks and agent training. During the initial development of VRGym, our capabilities were limited to the Unreal Engine version 4 (UE4). The multi-material simulation (rigid and deformable solids and fluids, as shown in Figure 5.3) in VRGym was achieved through the integration of NVIDIA Flex (Macklin et al., 2014). However, the advent of Unreal Engine version 5 (UE5) marked a notable enhancement in engine performance. Additionally, the introduction of the Niagara VFX System in UE5, a leading tool for creating visual effects, offers new avenues for the implementation of real-time physics-based solvers.

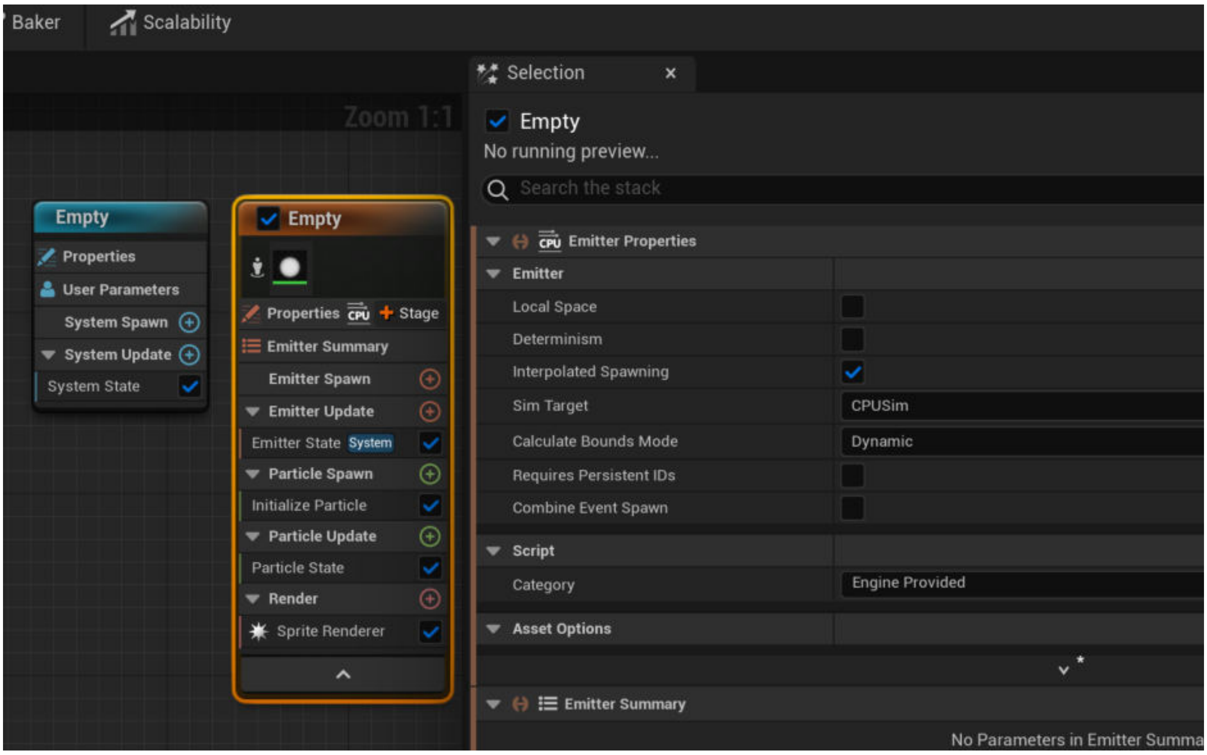
In this section, we discuss the redevelopment of the MPM solver utilizing the Niagara VFX system in UE5. We present an evaluation of our solver and provide a compilation of timing statistics. This redevelopment was undertaken during the internship at LightSpeed Studio. It is important to note that, due to copyright restrictions, an in-depth exposition of the method and its implementation details are omitted.

5.2.1 Background

This section is dedicated to elucidating the foundational elements of the Niagara System within Unreal Engine 5 (UE5), with an emphasis on contrasting traditional coding methodologies with the Niagara framework.

Niagara is a cutting-edge VFX system for UE5, specifically tailored for technical artists who may not possess extensive programming knowledge. Unlike conventional coding practices, Niagara facilitates the creation and customization of simulation pipelines through graphical user interfaces. This approach deviates from traditional text-based coding but offers a visual-oriented workflow where artists can define program behaviors by sequentially listing graphical modules or connecting graphical nodes. In addition, the system is equipped with a variety of templates, enabling artists to effortlessly modify and create desired visual effects. The contrast between the graphical user interface-driven Niagara system and traditional code-centric strategies marks a significant shift in the way visual effects are developed in modern gaming engines.

To commence the development of visual effects in Niagara, including those for physics-based simulations, the initial step is to instantiate a Niagara system. A key aspect of the Niagara system is its composition, which includes a system setup block and one or more Niagara emitters. In particular, one example of an empty Niagara system equipped with



☑ Empty

No running preview...

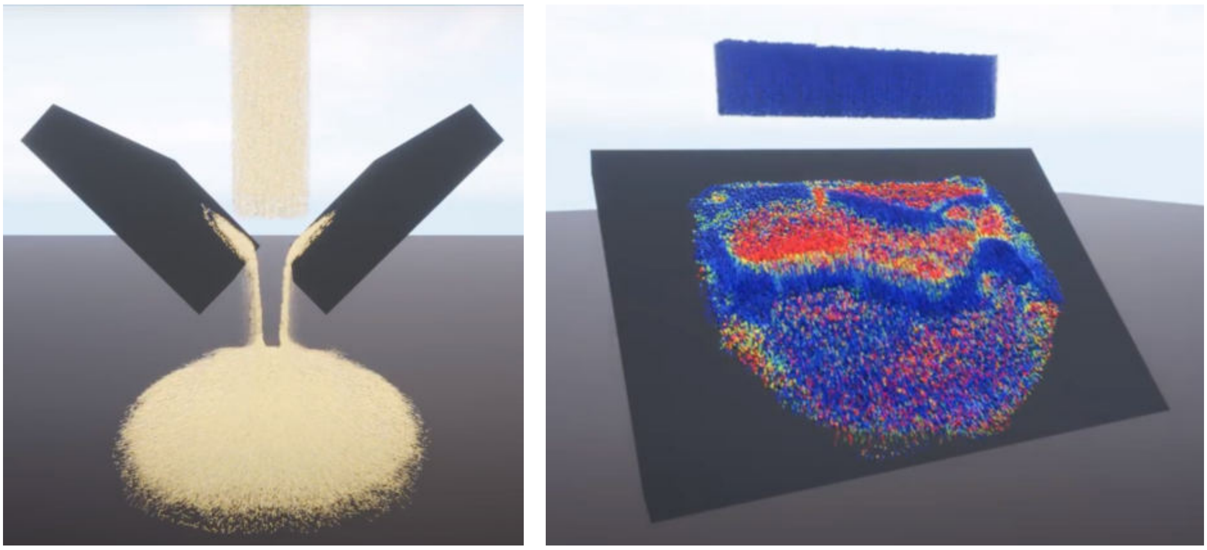
Local Space	☐
Determinism	☐
Interpolated Spawning	☑
Sim Target	CPUSim
Calculate Bounds Mode	Dynamic
Requires Persistent IDs	☐
Combine Event Spawn	☐
Category	Engine Provided

No Parameters in Emitter Summa

Figure 5.11: An Empty Niagara System with an empty Emitter.

a single empty emitter is illustrated in Figure 5.11. Generally speaking, the system setup block is designed to define overarching behaviors that are applied universally, impacting every emitter within that particular system instance. This block is also in charge of setting fundamental parameters and rules that govern the system-wide characteristics of the simulation. The Niagara emitters, on the other hand, define particle behaviors such as particle initialization, simulation steps, and rendering features. Within each of these components, both in the system setup and the emitters, a set of default stages are pre-integrated. Users are allowed to add additional stages to the emitters, enhancing their capability to customize the system. The update stages and any user-defined stages within the Niagara system can be executed on either CPUs or GPUs. This depends on the ”sim target” configuration of the respective emitters. When the ”sim target” is configured for GPU execution, both the particles and grids associated with that emitter are also stored and managed on the GPU.

Contrary to traditional coded simulation pipelines, the Niagara system, as exemplified



(a) (b)

Figure 5.12: Real-time Niagara MPM Simulation for (a) Drucker-Prager elastoplasticity (Klár et al., 2016) and (b) Non-Associated Cam-Clay (NACC) (Wolper et al., 2019).

in Figure 5.11, is not presented in a linear, time-sequential format. The list below outlines the sequential execution of these stages, with comments indicating the devices (CPU or GPU) on which each stage is run.

```
- System Spwan // CPU
- for each Emitter:
    - Emitter Spawn // CPU
    - Particle Spawn // GPU or CPU
- set TIME t = 0
- while true:
    - System Update // CPU
    - for each Emitter:
        - Emitter Update // CPU
        - Particle Update // GPU
        - User-defined Stages // GPU
    - t += dt
```

In order to implement an MPM solver based on the Niagara system, we also need to figure out how to define particle and grid data structures and how to perform data access and data transfers between different data structures. We refer (Tutorials) for more details.

Example	Sim-Par#	Render-Par#	Ave ms/run	fps	Grid Resolution
(Figure 5.12a) Real-time Sand (One Cylinder)	9, 635	77, 239 + 9, 635	0.43	120	64 × 64 × 64
(Figure 5.12a) Real-time Sand (Six Cylinder)	57, 810	521, 244	1.30	120	64 × 64 × 64
(Figure 5.12b) Real-time Soil (One Block)	46, 656	46, 656	1.25	120	64 × 64 × 64
(Figure 5.12a) Real-time Soil (Four Block)	186, 624	186, 624	3.63	110	64 × 64 × 64
(Figure 5.14) Real-time Fluid (One Block)	110, 592	110, 592	5.31	95	32 × 32 × 32

Table 5.1: **Parameters and timings.** We summarize the parameters of particle numbers (with "Sim-Par#" and "Ren-Par#" refers to particle numbers for simulation and rendering, respectively), grid resolutions, and the timing statistics for various experiments described in subsection 5.2.2. Here, "Ave ms/run" refers to the average timing for running a single round of Niagara GPU simulation, while the "fps" is the approximate average frame rate for running the test map, which contains not only the simulation but also the rendering and user-interaction controls.

5.2.2 Experiments

In this section, we demonstrate the performance of the UE5 Niagara-based MPM simulation system with various materials. We adopt the following constitutive models: 1) Drucker-Prager elastoplasticity (Klár et al., 2016) for sand animation, 2) Non-Associated Cam-Clay (NACC) (Wolper et al., 2019) for soil-like behaviors, and 3) weakly compressible fluid (Tampubolon et al., 2017) for fluid simulation. All materials are implemented through Niagara modules and can achieve simple switches by setting up a compile-time parameter in the Niagara system. All the experiments are performed on a desktop with one NVIDIA GeForce RTX 3080 GPU (10GB of GDDR6X RAM).

Real-time Sand. In this experiment, we interactively inject sand cylinders into the testing UE map by pressing some keyboard buttons, as shown in Figure 5.12a. Here, each sand cylinder contains 9635 particles for simulation purposes, and we additionally sample 77K particles that perform the G2P (Grid to Particle) step only. The detailed timing statistics are summarized in Table 5.1. Additionally, we test the interactive behaviors when setting different cohesion values for the sand model. As illustrated in Figure 5.13, the sands become wetter as the cohesion value increases. Both demos use a grid resolution of 64 × 64 × 64.

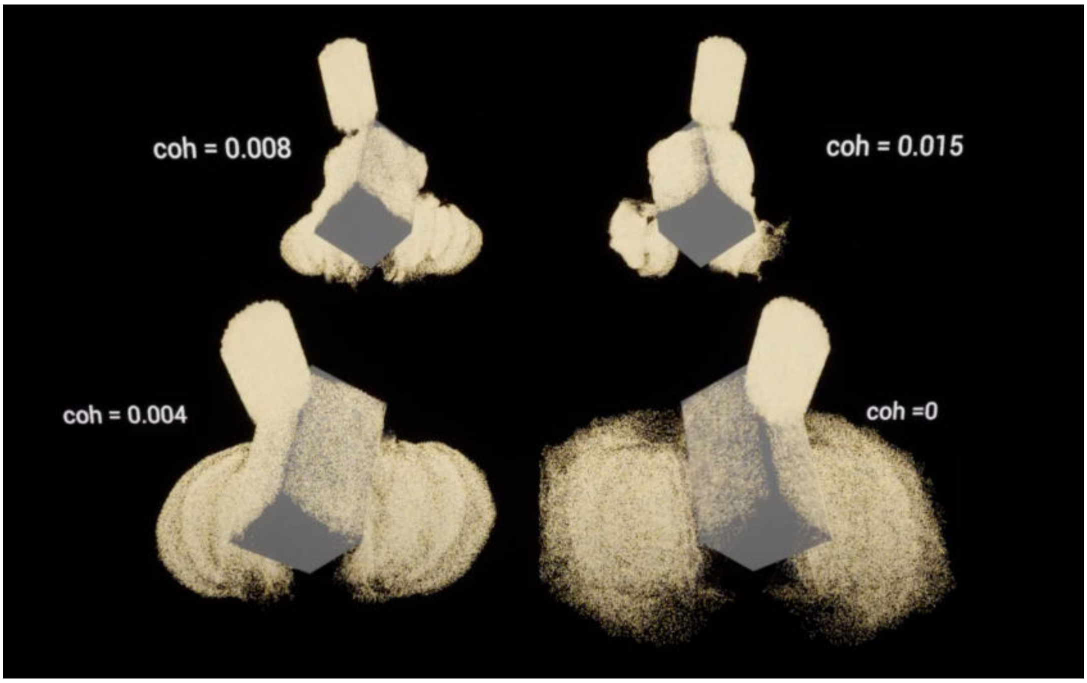


Figure 5.13: Real-time Niagara MPM sand solver with different cohesion value setups.

Real-time Soil. Similar to the sand, we simulate the soil dropping interactively. Here, each soil block contains 46.6K particles that are used for both simulation and rendering. To visualize damage propagation, we use the plastic volumetric strain to color the soil particles. The timing statistics for one-block and four-block soil simulations are listed in Table 5.1.

Real-time Fluid. In Figure 5.14, we simulate fluids dropping on top of some obstacles. Except for fluid block dropping, we further added some user-controllable blocks to create more interactive features. The player can change the value of the friction factor between the collision object and fluids. The fluid particles are colored through the initial particle ID, and the fluid turbulence is thus visible as well. In this test, the grid resolution is 32 × 32 × 32, and each fluid block contains 110K particles for both simulation and rendering.

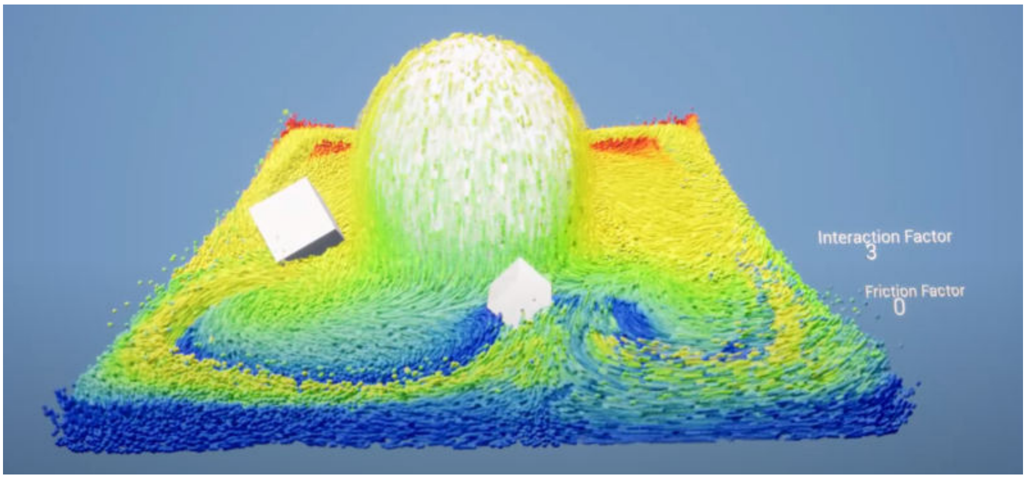


Figure 5.14: Real-time Niagara MPM fluid.

CHAPTER 6

Material Point Method for AI Generation Tasks

6.1 TPA-Gen: A Multi-Modal Data Generative Method for Text and Physics-Based Animation

Powered by an enormous amount of paired data from the vision and language domains, Vision-Language (V&L) Multi-Modality (MM) research has achieved remarkable results in both text-driven generation and understanding. However, constrained by the data, the learned MM knowledge space predominantly represents the alignments between text and appearances or shapes, lacking further understanding of the underlying dynamics. In this section, we aim to expand the Multi-Modality (MM) knowledge space by bridging the gap between text, vision, and real-world physical dynamics from a data-centric perspective, enabling MM models to better estimate these dynamics. We propose an automatic pipeline to generate Text-to-Video/Simulation (T2V/S) data. Each generated scenario comprises a high-resolution 3D physical simulation and a textual description of the physical phenomena. To simulate a diverse set of real-world dynamic phenomena—such as elastic deformations, material fractures, collisions, and turbulence—as faithfully as possible, we take advantage of state-of-the-art physical simulation methods: (i) Incremental Potential Contact (IPC) and (ii) Material Point Method (MPM). Additionally, high-quality, multi-view rendering is integrated into the pipeline. We envision our work as the first step towards fully automatic Text-to-Simulation (T2S), potentially shifting the paradigm towards understanding world dynamics.

6.1.1 Introduction

In the past few years, we have witnessed the blooming of the Vision-Language (V&L) Multi-Modality (MM) community in solving diverse tasks (Lu et al., 2019; Li et al., 2019a; Chen et al., 2020; Ramesh et al., 2021; Radford et al., 2021; Zhang et al., 2021; Alayrac et al., 2022; Gao et al., 2022; Kamath et al., 2021; Ramesh et al., 2022). Particularly, V&L models have achieved remarkable performances on various conventional V&L tasks thanks to the availability of an enormous amount of V&L data (Schuhmann et al., 2022) and the rapidly developing Large-scale Language Model (LLM) (Vaswani et al., 2017; Devlin et al., 2018; Radford et al., 2019; Brown et al., 2020). On the other hand, MM V&L generative tasks are unprecedentedly popular thanks to the advances in Vision-Language (V&L) domain. Text-to-Image (T2I) generation (Ho et al., 2020; Ramesh et al., 2021; Rombach et al., 2022; Saharia et al., 2022; Chang et al., 2023) can already produce commercial quality images from free-form text, and meanwhile, Text-to-Video (T2V) (Singer et al., 2022; Ho et al., 2022; Khachatryan et al., 2023) and Text-to-3D (T2-3D) (Jain et al., 2022; Poole et al., 2022; Jun and Nichol, 2023a) are also gaining more attention.

There are three key factors that jointly contribute to the success of the recent Vision-Language (V&L) research: (i) self-supervised learning techniques and self-attention/cross-attention deep learning architectures are fully explored (Vaswani et al., 2017; Devlin et al., 2018; Radford et al., 2019; Lewis et al., 2019; Brown et al., 2020); (ii) Vision-Language (V&L) generative models such as Denoising Diffusion Models (DDM) and Vector-Quantized (VQ) transformer decoder, are well studied (Goodfellow et al., 2020; Zhu et al., 2017; Ho et al., 2020; Rombach et al., 2022; Chang et al., 2022, 2023); (iii) Most importantly, a large volume of paired V&L data, *e.g.* (Lin et al., 2014; Ordonez et al., 2011; Sharma et al., 2018; Changpinyo et al., 2021; Schuhmann et al., 2022), are available on the Internet, enabling (i) and (ii) to capture the alignments between visual appearance/shape and language tokens' representations.

However, things are not as rosy as they seem. First, as shown in Figure 6.1(a), current MM V&L paradigm only models the alignments between visual characteristics

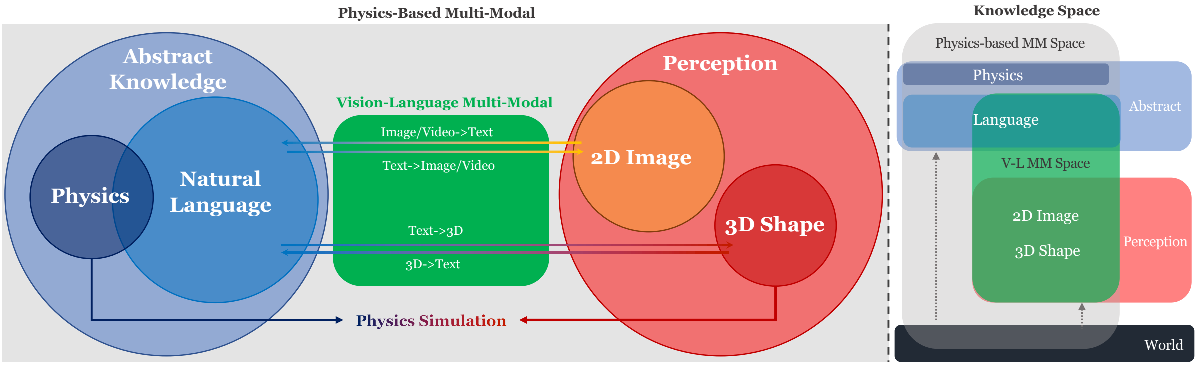


Figure 6.1: (a) Task space of Physics-based multi-modality. (b) Knowledge space of Physics-based multi-modality.

and corresponding text descriptions. Visual information, such as appearances and shapes, is a projection of the world dynamics to the perception space (see Figure 6.1(b)). This projection loses a lot of physical information, and the learned V&L representation may not reflect the real-world dynamics correctly. It will lead to distortions in both generation and understanding. For instance, the generated video may not be physically realistic. Second, from a statistical perspective, although there is a significant amount of V&L data, it is still far from fully covering the entire data distribution. In other words, the learned V&L representation cannot cover wide enough real-world phenomena in the spectrum. It will lead to a lack of generalizability and compositionality in text-driven V&L generation. Third, video-text paired data is more domain-specific compared to image-text paired data. In fact, T2V training oftentimes mixes T2I data. Lastly, the quality of publicly accessible V&L data is varied. A large portion of the data is not usable. Meanwhile, the cost of high-quality labeling is too expensive.

As illustrated in Figure 6.1, our solution is to expand the current V&L MM knowledge space to physics-based MM space where models can directly learn the alignments across text, perception, and physics. In this knowledge space, a Multi-Modality (MM) model can better estimate the world dynamics. In this section, we propose an automatic data generation pipeline to be the first towards this goal. In each run, our proposed pipeline generates a high-resolution, physically realistic animation with descriptive texts. To cover a wide enough range of physical phenomena, we take the advantage of (i) Incremental

Potential Contact (IPC)(Li et al., 2020b), a robust solid simulation framework that can accurately resolve the intricate contact dynamics for both rigid and deformable objects with guaranteed intersection-free results; (ii) Material Point Method (MPM)(Stomakhin et al., 2013; Sulsky et al., 1995), a multi-physics simulation framework that is capable of simulating versatile solids, fluids, granular materials, and multi-physics phenomena. Our pipeline covers various real-world dynamics, such as **deformations**, **fractures**, **collisions**, **turbulence**, *etc.* With commercial-level rendering tools, we also produce high-resolution multi-view videos. To summarize, our automatic data generation pipeline has three-folds of contributions:

- It generates high-quality physically realistic 3D animations along with sentences describing the physical phenomena, including a wide spectrum of commonly seen real-world dynamics.
- With the generated data, we can expand the current vision-language multi-modal knowledge space to the physics-based multi-modal knowledge space. It could help us to better estimate the real-world dynamics behind the scene.
- The generated high-quality data provides more diversity, which could benefit a wide range of research, such as Text-to-Image (T2I), Text-to-Video (T2V), Text-to-3D (T2-3D), Text-to-Simulation (T2S), and Text-to-Animation (T2A).

6.1.2 Related Work

Text-to-Image and Text-to-Video Generation.

Reed et al. (2016) are recognized as the pioneers in Text-to-Image (T2I), which extends the Generative Adversarial Network (GAN) (Goodfellow et al., 2020) to multi-modal generation. Similarly, Zhang et al. (2017) and Xu et al. (2018) apply GAN variants and further enhance the quality of the generated images with improved image-text alignments. Other works, such as DALL-E (Ramesh et al., 2021), formulate the T2I problem as a sequence-to-sequence transfer, and incorporate both Transformer and VQVAE for solutions. Some follow-up studies show that the results could be further improved by replacing DALL-E components with other vision language modules, such as the CLIP latent space in DALLE2 (Ramesh et al., 2022). Moreover, the recent success of Denoising Diffusion Models (DDM) (Ho et al., 2020; Rombach et al., 2022) also improves the generation quality with cascading up-sampling diffusion decoder. In Text-to-Video (T2V), most previous works (Pan et al., 2017; Li et al., 2018) produce relatively low-resolution videos in simplified domains. The latest research (Wu et al., 2021; Hong et al., 2022b; Singer et al., 2022; Ho et al., 2022; Khachatryan et al., 2023) extends the T2I framework to T2V by improving modules in diffusion-based T2I framework, adding additional attention modules, and making use of both image-text and video-text data.

Text-to-3D, Text-to-Animation Generation and 3D-Text Retrieval.

As extensions of T2I, DreamFusion (Poole et al., 2022) and (Michel et al., 2022) synthesize 3D meshes from texts. Moreover, DreamField (Jain et al., 2022) generates radiance field with NeRF. Latest work such as Shap-E (Jun and Nichol, 2023b) predicts latent parameters for 3D texture and radiance field. Chen et al. (2022b) use texts to control lighting conditions in rendering. Besides, several works use CLIP to enable text-to-3D representations. For example, Mohammad Khalid et al. (2022) generate mesh and texture in CLIP space; Wang et al. (2022) incorporate CLIP with NeRF, enabling simple text-editable 3D object manipulation; (Tevet et al., 2022) generate human motion from text. (Hong et al., 2022a) further apply text-to-3D generation to Avatar.

Vision-Language Datasets.

Microsoft COCO(Lin et al., 2014), Google concept caption (Sharma et al., 2018; Changpinyo et al., 2021), WIT (Srinivasan et al., 2021), and VisualGenome (Krishna et al., 2017) *etc.* are the most popular fine-labeled image-based V&L datasets. CLVER (Johnson et al., 2017) is one of the iconic synthetic V&L datasets. Besides, billions of image-text pairs have been collected from the internet, such as SBU and LAION 5B (Ordonez et al., 2011; Schuhmann et al., 2022). These image-text pair datasets significantly contribute to the success of recent T2I generative models. On the hand, there are less video-text data available, especially fine-annotated video-text datasets. Existing work includes HowTo100M (Miech et al., 2019), which mainly focuses on instructional descriptions, and WebViD (Bain et al., 2021), which contains high-quality daily activity video clips. Additionally, MSRVTT (Xu et al., 2016), MSVD (Chen and Dolan, 2011), DiDeMo (Hendricks et al., 2018), and ActivityNet (Caba Heilbron et al., 2015) are commonly used, especially for video-language pre-training. Most of them only contain daily human activity without physical world dynamics.

6.1.3 Automatic TPA Generation

As demonstrated in Figure 6.2, our work employs the attributed stochastic grammar to represent the unified knowledge scenario space that can be instantiated to concrete representations in any modality. Specifically, this tree-structured representation uses *node*s to represent the object-of-interests and environmental and rendering setups, with different collections of *attribute*s attached to each *node* to represent corresponding properties, as explained in §6.1.3.1. The dynamic behaviors of multiple objects are characterized using dynamic models (§6.1.3.2) that constrain object velocity properties, as well as multi-object motion and positional relationships. By utilizing constrained sampling, we can obtain a parse tree that represents the initial states and motion characteristics of a concrete scenario, which can then be translated into a physical simulation, rendered videos, and descriptive captions.

The procedure of parse tree sampling is summarized below and elaborated in §6.1.3.3. First, the parse tree structure is sampled from the stochastic grammar to decide the content of a scenario. This process will determine the number of simulated objects and collision objects, as well as environmental and rendering setups. Following that, node-related *attribute*s at each hierarchical level will be determined. Afterward, a dynamic model is chosen at random based on the number of objects in the scene, and relation and motion constraints are applied to the *attribute*s of selected objects accordingly.

After settling all *attribute*s and the sampling process is finished, we dump this data

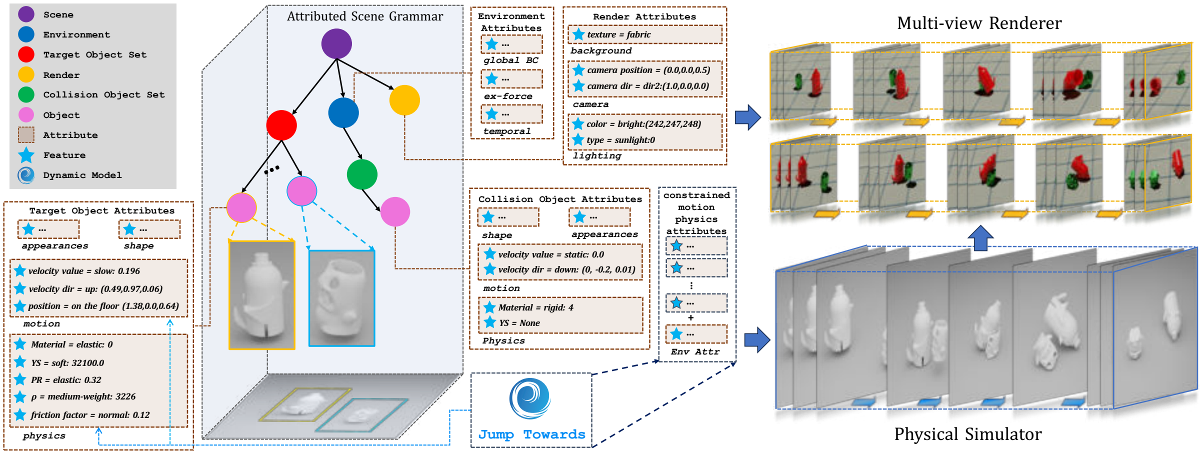


Figure 6.2: Attributed Scene Grammar that defines the unified knowledge scenario space and its instantiation to 3D-physics and 2D-vision domain.

instance to a JSON file and import it to the IPC or MPM simulator based on the user's choice. IPC can be used to simulate elastic or rigid objects with an accurate friction handler, whereas MPM is adept at handling a variety of materials, including elastic, plastic solids, granular materials, and fluids. The simulator will produce 3D scene representations at multiple discretized time steps. With these data, we can further generate photorealistic videos using an automatic rendering algorithm based on the rendering configurations in the parse tree. More details of the process are introduced in §6.1.3.4.

In addition, a stochastic language grammar is constructed from the scenario representation concurrently with the simulation process (§6.1.3.5). A collection of sentences that characterize the animation using randomly selected descriptors is produced as described in §6.1.3.6. These sentences are subsequently rewritten using ChatGPT interfaces. In the subsequent subsections, additional modeling and sampling details are introduced.

6.1.3.1 Attributed Scene Grammar

As previously introduced, we use an attributed stochastic grammar to represent the scenario domain. Specifically, the stochastic grammar is a hierarchical tree composed of the following *node* types: Scene, Target Object Set, Collision Object Set, Environment, Render, and Object. Here, a Scene *node* is the root *node* containing three *nodes*, Target

Object Set, Environment, and Render. Furthermore, Target Object Set and Collision Object Set are non-terminal *set nodes* that can contain an arbitrary number of non-terminal *set nodes* of the same type or a random number of Object *nodes* that are leaf nodes of the grammar. Environment *nodes* are also non-terminal that contain Collision Object Set *nodes*. Each *node*, according to its categorization, has a particular set of attributes.

Object Nodes. Object nodes belonging to the same *set* may have special semantic relationships, while Objects in a specified Collision Object Set may constrain the motion and position of Objects in a given Target Object Set. Moreover, each Object *node* contains multiple categories of *attributes*, including object-render, shape, motion, and physics. The *attributes* are used to specify the corresponding characteristics of the object. Each of them contains several concrete dependent or independent *features* that can be directly mapped to a semantic label and a range of quantitative values. For example, physics *attribute* consists of three independent *features* (material type, friction coefficient, and material density) and two dependent *features* (Young's Modulus and Poisson Ratio). The values of the dependent *features* rely on the sampling results of both the other independent *features* and their own label. In the physics, Young's Modulus, for instance, determines the material's resistance to elastic deformation under loads and is therefore dependent on both material types (whether the object is fluids, granular, soft, or rigid solids) and the sampling results of its own label (whether the object is relatively softer or harder).

Environment Nodes. Environment contains *attributes* to control general scenario configurations such as boundaries, external forces, and temporal discretizations. The boundary *attribute* has a BC *features* for controlling boundary shape, type, and friction settings, a Force *feature* for determining the external force, and a Time *feature* for specifying temporal step size and the total number of frames.

Render Nodes. Expect the object-render *attribute* attached to each Object to depict the object color and reflective properties, there are additional rendering setups that can reflect the human's visual imagination of a given scenario. We use a terminal *node*, `Render`, to specify those configurations, such as background light, textures, and the position of the camera. All of these setups are, as before, supported by *attribute*s with detailed *features*.

6.1.3.2 Dynamic Model

We propose dynamic models to characterize and constrain object motions and relationships in addition to the tree-structure grammar. Each dynamic model can be mapped to a **verb** that semantically describes the velocity feature and interactions between **subjective** and **objective** objects. It may also include directional descriptors such as `from` and `to` to further guide the objects' moving characteristics and initial position properties.

Currently, our data generative model supports the following dynamic models: `JUMP`, `DROP`, `THROW`, `PUSH` and `STRIKE`. The first three models, which are referred to by intransitive verbs, are capable of influencing the behavior of one or two objects. If a single object is sampled, these models will either constrain the initial position of the object to be on the ground or in the air by confining the corresponding position *feature* in the `motion` *attribute*. In addition, they will assist the object in choosing an appropriate velocity scale and movement direction. If two `Object`s are sampled in the corresponding *set nodes*, however, a directional descriptor will be sampled to customize the relationship between these two objects. One of them is selected at random to serve as the subjective object, while the remainder severs as the objective. Their relationship will be constrained by the selected directional descriptor. `From`, for instance, indicates that the initial position of the subjective object is close to the objective object and that it is moving in the opposite direction; whereas `to` means that the subjective object starts from a relatively distant location and moves toward the objective.

The `PUSH` and `STRIKE` models are slightly distinct due to the transitive nature of these verbs. This suggests that they inherently associate an objective object with the subject described. The semantic meaning of the verbs also constrains the initial positions and motion directions of the involved objects. If directional descriptors are also sampled within the transitive dynamic models, an extra object will be introduced with additional constraints. As a case study, we can sample a model that reads "*sub* PUSH *obj_1* to *obj_2*". This can be interpreted as the *sub* moving toward *obj_1* with velocities pointing to *obj_2* and trying to push *obj_1* in the direction of *obj_2*.

In practice, our model can be expanded by integrating additional dynamic models with minimal design and implementation effort. In human languages, all verb semantics are subjectively defined, necessitating the manual design of object relations and feature constraints. As we offer a variety of constraint/relation/feature-related abstract interfaces for defining, validating, and applying each user-defined constraint, it is straightforward to convert the design of a constraint set into a dynamic model class in our codebase.

6.1.3.3 Scenario Sample Process

In order to instantiate a concrete scenario from the stochastic grammar, we need to sample 1) a parse tree structure that defines the content and characteristics of a scene, 2) the concrete qualitative labels and quantitative values of *feature*s in corresponding *attribute*s, 3) a dynamic model with a specified verb and optional directional descriptor that constrains the dynamic behavior of several objects. This section examines these three phases in detail.

Sample Parse Tree Structure. The structure sampling procedure begins at the root *node* and progresses downward until it reaches the terminal *node*s. Different *node*s are sampled according to their individual categories in order to specify which children *node*s are essential and which are optional. *Set node*s, for instance, can sample any number of children *node*s within a permitted children number range. In contrast, an Environment *node* must contain at least one Collision Object Set node. Following this phase, the total number of object-of-interest and collision objects will be determined. This information will further narrow our selection of dynamic models.

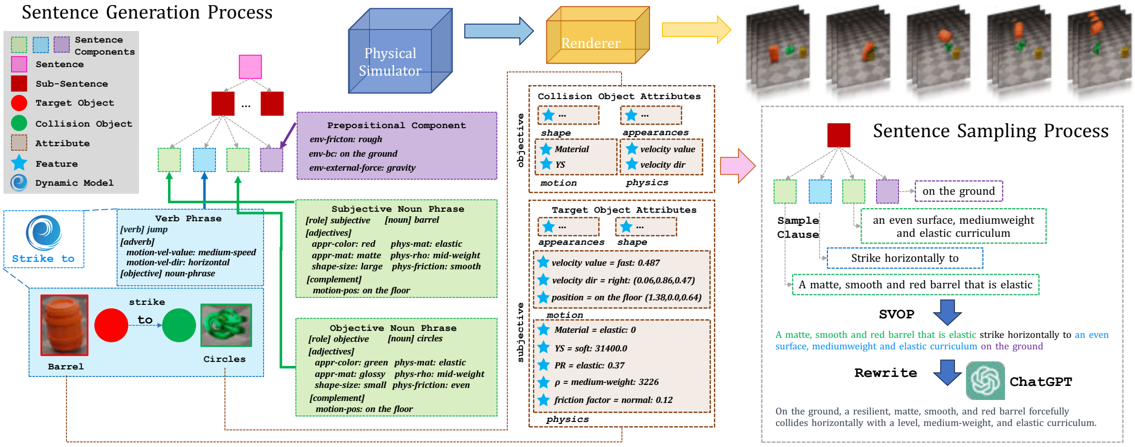

Sentence Generation Process
Sentence Components
Sentence
Sub-Sentence
Target Object
Collision Object
Attribute
Feature
Dynamic Model
...
Physical Simulator
Renderer
Prepositional Component
env-fricton: rough
env-bc: on the ground
env-external-force: gravity
Strike to
Verb Phrase
[verb] jump
[adverb]
motion-vel-value: medium-speed
motion-vel-dir: horizontal
[objective] noun-phrase
Subjective Noun Phrase
[role] subjective [noun] barrel
[adjectives]
appr-color: red phys-mat: elastic
appr-mat: matte phys-rho: mid-weight
shape-size: large phys-friction: smooth
[complement]
motion-pos: on the floor
strike
to
Barrel
Circles
Objective Noun Phrase
[role] objective [noun] circles
[adjectives]
appr-color: green phys-mat: elastic
appr-mat: glossy phys-rho: mid-weight
shape-size: small phys-friction: even
[complement]
motion-pos: on the floor
objective
subjective
Collision Object Attributes
...
...
shape
appearances
Material
YS
velocity value
velocity dir
motion
physics
Target Object Attributes
...
...
appearances
shape
velocity value = fast: 0.487
velocity dir = right: (0.06,0.86,0.47)
position = on the floor (1.38,0.0,0.64)
motion
Material = elastic: 0
YS = soft: 31400.0
PR = elastic: 0.37
ρ = medium-weight: 3226
friction factor = normal: 0.12
physics
Sentence Sampling Process
on the ground
Sample Clause
an even surface, mediumweight and elastic curriculum
Strike horizontally to
A matte, smooth and red barrel that is elastic
SVOP
A matte, smooth and red barrel that is elastic strike horizontally to an even surface, mediumweight and elastic curriculum on the ground
Rewrite
ChatGPT
On the ground, a resilient, matte, smooth, and red barrel forcefully collides horizontally with a level, medium-weight, and elastic curriculum.

Figure 6.3: Sentence generation process.

Sample *Features*. As specified in §6.1.3.1, each *node* in the parse tree comprises a number of *attributes*. Additionally, an *attribute* contains multiple *features* to determine particular semantic properties. In this stage, the goal is to first sample a qualitative label for each *feature*, then sample the corresponding absolute values that define certain physical or visual properties.

To attain this objective, we begin by randomly selecting *feature* labels in each tree hierarchy using in a top-down sampling manner. The independent *features* are sampled from the candidate pool, while the dependent *features* are sampled subsequently to ensure semantic consistency in describing the scenario. Following the selection of all *feature* labels, concrete *feature* values are sampled from the predefined quantitative ranges of each label.

Top-down sampling provides only an initial selection of *features*; the final labels and values are further determined by the choice of dynamic models (next paragraph). With the selected model, all concerned *features* will undergo a bottom-up refinement. We first evaluate whether the values of specified *features* satisfy the dynamic model's constraints. If not, the out-of-range projection of the values will be removed in order to enforce the constraints. Once all constraints are met, we reselect the *feature* labels based on the modified values.

Sample Dynamic Models. As indicated in §6.1.3.2, diverse dynamic models are capable of guiding a variety of object counts. In this step, the choice of the dynamic model is based on the total number of objects (including object-of-interests and collision objects). The total number of objects must be sufficient to accommodate the motion and relationship depicted by the current model. Afterward, we randomly select objects from the parse tree to serve as the subjective or objective(s) of the respective motion. Note that the subject can only refer to the object-of-interest, whereas the objective can be of any types. If additional free objects or collision items remain, they will be regarded as noise unrelated to the current scenario data point and will not be included in the language model (§6.1.3.5).

6.1.3.4 Simulation and Rendering

After determining the scene parse tree structure with appropriate *feature* labels and values, the data point is transferred into a JSON format. This output JSON file is then sent to an Incremental Potential Contact (IPC) (Li et al., 2020b) or an Material Point Method (MPM) (Qiu et al., 2023) simulator based on the sampled object materials or the user's preferences. According to the JSON, physical simulators initially load object shapes (§6.1.4) and assign both object- and environment-related parameters from corresponding *feature* values. Then, the object motion and material behaviors such as deformation and fractures are simulated until the maximum frame number is reached.

A renderer then collects the 3D output results at various time steps to generate high-fidelity rendering results. In addition, rendering configurations such as background texture, object colors, reflective materials, background light, and camera position are also loaded from the sampled JSON file. Blender (Blender, 2018), which is open-sourced and supports fully Python-scriptable rendering operations, is used to accomplish automatic rendering.

In addition to the technologies employed by the proposed pipeline, other publicly available simulators (*e.g.*, NVIDIA's FleX (Macklin et al., 2014)) and rendering engines can also be utilized with corresponding JSON file parser.

6.1.3.5 Language Generation Model

On the basis of the sampled scenario parse tree, a hierarchical tree-structured language model is constructed (Figure 6.3). In this model, the root node, which represents a *sentence* structure, is decomposable into multiple *sub-sentence* nodes. Each *sub-sentence* consists of three nodes representing typical linguistic components: a *prepositional* node, a *noun phrase* node, and a *verb phrase* node. In this case, the *prepositional* node collects feature labels associated with environmental and rendering configurations in the parse tree, thereby describing the global scenario characteristics. If the *noun phrase* node is placed under the root node, it is considered the subjective object in the scene; otherwise, if belonging to a *verb phrase*, it is regarded as the objective object. Detail-wise, a *noun phrase* contains a *noun* and its descriptors which are summarized from the corresponding `Object` *node*. And *verb phrase*, on the other hand, has a *verb* with dynamic descriptors and multiple *noun phrase* children nodes performing the objective roles.

When constructing the language model, we begin by examining the type of dynamic model and mapping it to a verb in the *verb phrase*; the object relationships in the dynamic model determine which `Object` falls to the *subjective phrase* and which refers to the *objectives*. Then, if the *feature*s in the parse tree and dynamic model merit being stated in sentences, they are assigned to language components. Specifically, the shape *feature* of an `Object` is captured by certain *noun phrases*, and the tags of the corresponding object mesh are retrieved and sampled as the noun. The other physical and rendering features are attached as adjective descriptors to the *noun phrase*. Certain specialized *feature*s, such as Young's Modulus, are too specific to be included in common language and are therefore neglected. Additionally, the verb phrase collects the subjective `Object`'s velocity-related features as auxiliaries. And finally, *feature*s associated with the `Environment` and `Render` *node*s are inserted into the *prepositional* node.

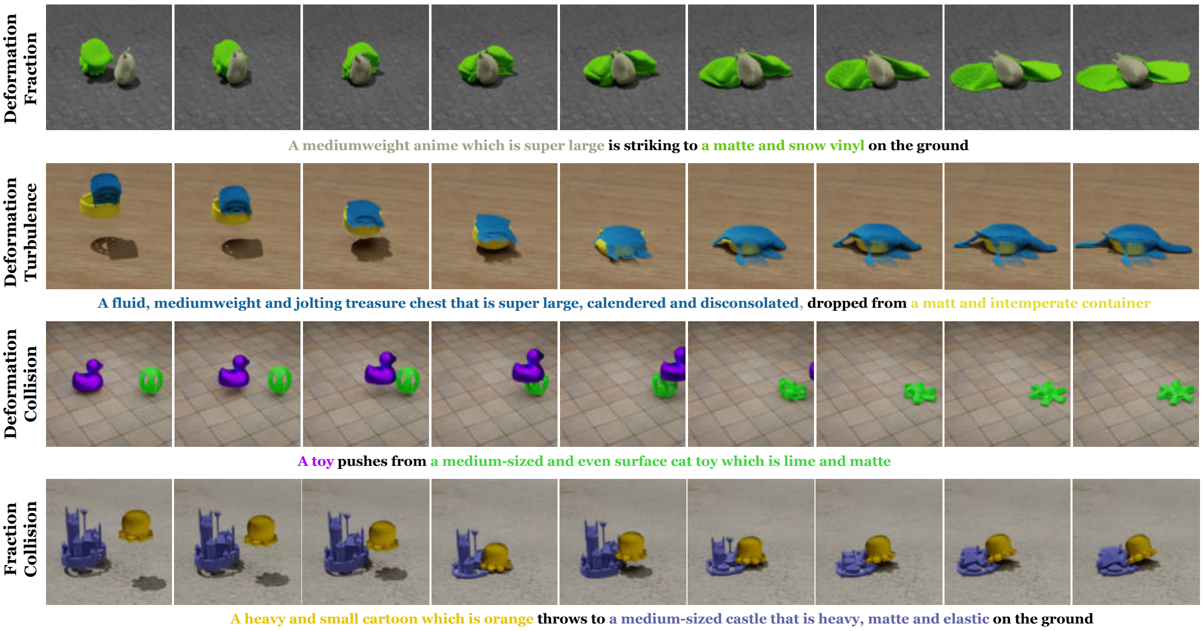
6.1.3.6 Generating Random Sentences

In addition to the aforementioned structured language model, the next stage is to create concrete sentences that describe the scenario. The entire sentence consists of sub-sentences joined by conjunctions such as ",", ";" or "and". Additionally, each sub-sentence is composed of a subjective phrase, a verb phrase, and prepositional components. To obtain each finalized sub-sentence, we break down the problem into multiple steps, which are described in the paragraphs that follow.

Sample Sentence Structure. The components of the sub-sentences can appear in a variety of arrangements to create diverse sentence structures. The objective of this phase is to determine this order. We offer several common sentence structures as candidates. For example, SVOP (Subject-Verb-Object-Preposition) is the most common English sentence structure, whereas OVS is an example of passive voice.

Sample Components. In this step, each sentence component is sampled independently into concrete clauses and concatenated in the order specified by the predefined sentence structure. Using the language model constructed in §6.1.3.5, the corresponding clauses can be formed. As stated previously, each language component node contains all the accumulated feature labels from the parse tree. However, including all potential descriptive terms in our everyday language is unnecessary and cumbersome. Therefore, we arbitrarily select a number of descriptors that appear either before or after the noun/verb in corresponding *phrases*.

Specifically, for *noun-phrases*, descriptors displayed before nouns are adjectives joined by conjunctions, whereas descriptors presented after nouns can be formulated as subclauses introduced by "that" or "which". The descriptor number can also be zero, indicating that the noun contains no description portion. As an instance, we can sample two adjectives and three clause-descriptors from the subjective *noun phrase* to construct a noun phrase clause such as "a blue and matte cube that is small, elastic, and rough". The phrases "blue and matte", "cube", and "small, elastic, and rough" denote the adjective-, noun-,



A mediumweight anime which is super large is striking to a matte and snow vinyl on the ground

A fluid, mediumweight and jolting treasure chest that is super large, calendered and disconsolated, dropped from a matt and intemperate container

A toy pushes from a medium-sized and even surface cat toy which is lime and matte

A heavy and small cartoon which is orange throws to a medium-sized castle that is heavy, matte and elastic on the ground

Figure 6.4: Examples of generated animations.

and clauses-portions, respectively.

Similar strategies are used to manage descriptors in the verb-phrase case. To enhance sentence diversity, we sample the verb tense further when generating the verb phrase clause. The objective object of the verb is sampled as another noun phrase. As for the prepositional portion, a random sample of conjunction-coupled labels is selected.

Sentence Diversity. To maximize the diversity of the sampled sentences, we sample them multiple times so that various types and quantities of object and motion descriptors are chosen. Then, ChatGPT interfaces are called to rewrite each sentence with suitable prompts. In this sense, both sentence structure and word synonyms are interchangeable.

6.1.4 3D Shape Collecting and Processing

Apart from the generation procedure described in the §6.1.3, additional support is required to complete the pipeline. That is, to collect, process, and utilize 3D shapes with noun labels indicating what the shape represents. This section describes three methods for achieving this objective, along with their advantages and disadvantages.

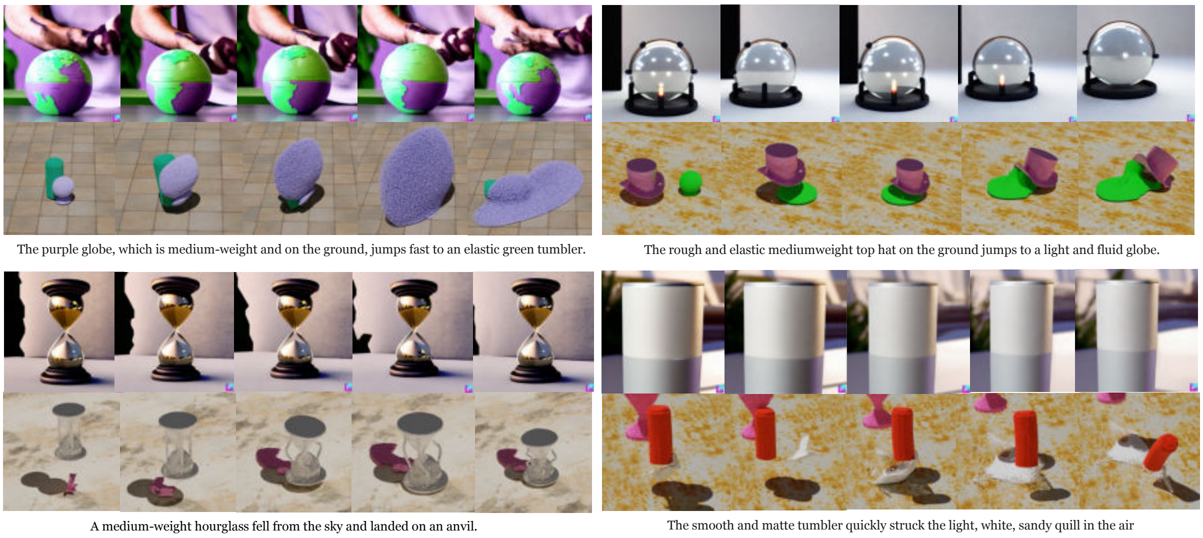
3D Object From Existing Dataset. Existing 3D object datasets, such as Thingi10K(Zhou and Jacobson, 2016), can be used as the prospective shape pool. Preprocessing is required to meet the input format specifications of Incremental Potential Contact (IPC) and Material Point Method (MPM). Particularly, IPC accepts a tetrahedral geometry (4 vertices per face, .ply format) whereas MPM accepts a 3D volumetric signed distance field (.vdb format).

Thanks to the contributions of Zhou and Jacobson (2016), we can readily collect a large number of 3D forms. Nevertheless, noun sampling with Thingi10K is a laborious procedure. Specifically, we use properties like titles and tags affixed to all the shapes as the object noun vocabulary. However, the descriptiveness and quality of these terms are less reliable (containing adjectives like "funny", "movable" and over-broad concepts like "3D", or "art").

Text-Based 3D Shape Generation. Another alternative is to generate 3D shapes from text labels. We can ask ChatGPT to generate a certain number of nouns of a specific type, and then feed these words into a text-to-3D model to generate 3D shapes. In our experiments, we use Shap-E (Jun and Nichol, 2023b), but other similar generators could also be viable alternatives.

This procedure mitigates the disadvantage described in the preceding paragraph. However, new difficulties arise. First, the majority of models for generating 3D shapes require carefully adjusted parameters for desired results, which doesn't fit our requirement for automatic mass production. Therefore, the generated 3D meshes are of unreliable quality. To filter out invalid meshes that are noisy, incomplete, or contain too many holes, we designate all generated shapes with a command-line labeling tool.

Text-Based 3D Shape Retrieval Utilizing an existing 3D shape retrieval model (*e.g.* (Liu et al., 2023a)) to search and extract existing shapes from a pre-sampled noun list is an alternative solution. The quality of the retrieved meshes is trustworthy, and the noun we employed for retrieval can serve as the noun descriptor in our sample sentences. We



The purple globe, which is medium-weight and on the ground, jumps fast to an elastic green tumbler.

The rough and elastic mediumweight top hat on the ground jumps to a light and fluid globe.

A medium-weight hourglass fell from the sky and landed on an anvil.

The smooth and matte tumbler quickly struck the light, white, sandy quill in the air

Figure 6.5: Side-by-side comparison of generated animation and zero-shot Text-to-Video (T2V).

are eager to incorporate this work into our pipeline once their code has been published.

6.1.5 Qualitative Comparison of T2V Generation

We list some representative rendering results with descriptive captions in Figure 6.4. More demos can be found in the Appendix C. Due to the absence of benchmark methods, we set up an qualitative comparison between generated animation and zero-shot text-to-video generation results. Although it may not be a fair comparison, it still conveys our the main idea of the proposed method. As shown in Figure 6.5, the zero-shot Text-to-Video (T2V) (Khachatryan et al., 2023) generated results shows very limited dynamic interactions between the involved objects and the world. Oftentimes the video shows no dynamics but simply slight viewpoint shifts. However, our generated animations show vivid physical dynamic interaction across the scene. As mentioned in §6.1.1, such a difference is caused by the absence of modeling physical knowledge in the Multi-Modality (MM) knowledge space.

6.1.6 Conclusion

In this study, we have introduced an innovative approach for automatically generating physics-based animations with textual descriptions. Our method has been extensively analyzed, and we have presented both qualitative results of our data generation method and comprehensive experiments that highlight the importance of such physically realistic datasets to the multi-modal generation research community. We believe that the addition of these resources could substantially contribute to the expansion of the current vision-language multi-modal knowledge space, facilitating improved understandings and estimations of real-world dynamics.

6.2 PhysGaussian: Physics-Integrated 3D Gaussians for Generative Dynamics

In this section, we introduce PhysGaussian, a new method that seamlessly integrates physically grounded Newtonian dynamics within 3D Gaussians to achieve high-quality novel motion synthesis. Employing a custom Material Point Method (MPM), our approach enriches 3D Gaussian kernels with physically meaningful kinematic deformation and mechanical stress attributes, all evolved in line with continuum mechanics principles. A defining characteristic of our method is the seamless integration between physical simulation and visual rendering: both components utilize the same 3D Gaussian kernels as their discrete representations. This negates the necessity for triangle/tetrahedron meshing, marching cubes, "cage meshes," or any other geometry embedding, highlighting the principle of "what you see is what you simulate (WS^2)." Our method demonstrates exceptional versatility across a wide variety of materials–including elastic entities, plastic metals, non-Newtonian fluids, and granular materials–showcasing its strong capabilities in creating diverse visual content with novel viewpoints and movements. Certain results are better presented in video format and can be found in the supplementary document.

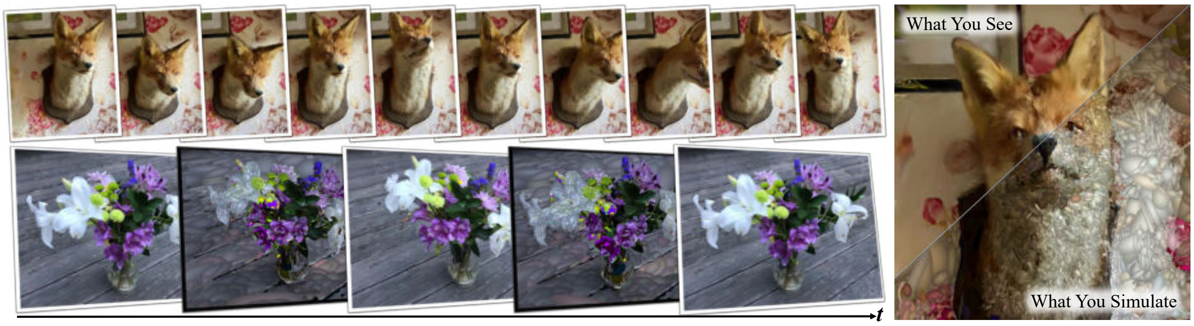


Figure 6.6: **PhysGaussian** is a unified simulation-rendering pipeline based on 3D Gaussians and continuum mechanics.

6.2.1 Introduction

Recent strides in Neural Radiance Fields (NeRFs) have showcased significant advancements in 3D graphics and vision (Mildenhall et al., 2021). Such gains have been further augmented by the cutting-edge 3D Gaussian Splatting (GS) framework (Kerbl et al., 2023). Despite many achievements, a noticeable gap remains in the application towards generating novel dynamics. While there exist endeavors that generate new poses for NeRFs, they typically cater to quasi-static geometry shape editing tasks and often require meshing or embedding visual geometry in coarse proxy meshes such as tetrahedra (Yuan et al., 2022; Xu and Harada, 2022; Peng et al., 2022; Jambon et al., 2023).

Meanwhile, the traditional physics-based visual content generation pipeline has been a tedious multi-stage process: constructing the geometry, making it simulation-ready (often through techniques like tetrahedralization), simulating it with physics, and finally rendering the scene. This sequence, while effective, introduces intermediary stages that can lead to discrepancies between simulation and final visualization. Even within the NeRF paradigm, a similar trend is observed, as the rendering geometry is embedded into a simulation geometry. This division, in essence, contrasts with the natural world, where the physical behavior and visual appearance of materials are intrinsically intertwined. Our overarching philosophy seeks to align these two facets by advocating for a unified representation of a material substance, employed for both simulation and rendering. In essence, our approach champions the principle of *"what you see is what you simulate*

(WS^2)", aiming for a more genuine and coherent integration of simulation, capturing, and rendering.

Building towards this goal, we introduce PhysGaussian: physics-integrated 3D Gaussians for generative dynamics. This novel approach empowers 3D Gaussians to encapsulate physically sound Newtonian dynamics, including realistic behaviors and inertia effects inherent in solid materials. More specifically, we impart physics to 3D Gaussian kernels, endowing them with kinematic attributes such as velocity and strain, along with mechanical properties like elastic energy, stress, and plasticity. Notably, through continuum mechanics principles and a custom Material Point Method (MPM), PhysGaussian ensures that both physical simulation and visual rendering are driven by 3D Gaussians. This eradicates the necessity for any embedding mechanisms, thus eliminating any disparity or resolution mismatch between the simulated and the rendered.

We present PhysGaussian's versatile adeptness in synthesizing generative dynamics across various materials, such as elastic objects, metals, non-Newtonian viscoplastic substances (e.g. foam or gel), and granular mediums (e.g. sand or soil). To summarize, our contributions include

- **Continuum Mechanics for 3D Gaussian Kinematics:** We introduce a continuum mechanics-based strategy tailored for evolving 3D Gaussian kernels and their associated spherical harmonics in physical Partial Differential Equation (PDE)-driven displacement fields.

- **Unified Simulation-Rendering Pipeline:** We present an efficient simulation and rendering pipeline with a unified 3D Gaussian representation. Eliminating the extra effort for explicit object meshing, the motion generation process is significantly simplified.

- **Versatile Benchmarking and Experiments:** We conduct a comprehensive suite of benchmarks and experiments across various materials. Enhanced by real-time GS rendering and efficient MPM simulations, we achieve *real-time* performance for scenes with simple dynamics.

6.2.2 Related Work

Radiance Fields Rendering for View Synthesis.

Radiance field methods have gained considerable interest in recent years due to their extraordinary ability to generate novel-view scenes and their great potential in 3D reconstruction. The adoption of deep learning techniques has led to the prominence of neural rendering and point-based rendering methods, both of which have inspired a multitude of subsequent works. On the one hand, the NeRF framework employs a fully-connected network to model one scene (Mildenhall et al., 2021). The network takes spatial position and viewing direction as inputs and produces the volume density and radiance color. These outputs are subsequently utilized in image generation through volume rendering techniques. Building upon the achievements of NeRF, further studies have focused on enhancing rendering quality and improving training speeds (Fridovich-Keil et al., 2022; Müller et al., 2022; Sun et al., 2022; Barron et al., 2022; Xu et al., 2022). On the other hand, researchers have also investigated differentiable point-based methods for real-time rendering of unbounded scenes. Among the current investigations, the state-of-the-art results are achieved by the recently published 3D Gaussian Splatting framework (Kerbl et al., 2023). Contrary to prior implicit neural representations, GS employs an explicit and unstructured representation of one scene, offering the advantage of straightforward extension to post-manipulation. Moreover, its fast visibility-aware rendering algorithm also enables real-world dynamics generations.

Dynamic Neural Radiance Field.

An inherent evolution of the NeRF framework entails the integration of a temporal dimension to facilitate the representation of dynamic scenes. For example, both Pumarola et al. (2021) and Park et al. (2021) decompose time-dependent neural fields into an inverse displacement field and canonical time-invariant neural fields. In this context, the trajectory of query rays is altered by the inverse displacement field and then positioned within the canonical space. Subsequent studies have adhered to the aforementioned design when exploring applications related to NeRF deformations, such as static scene editing

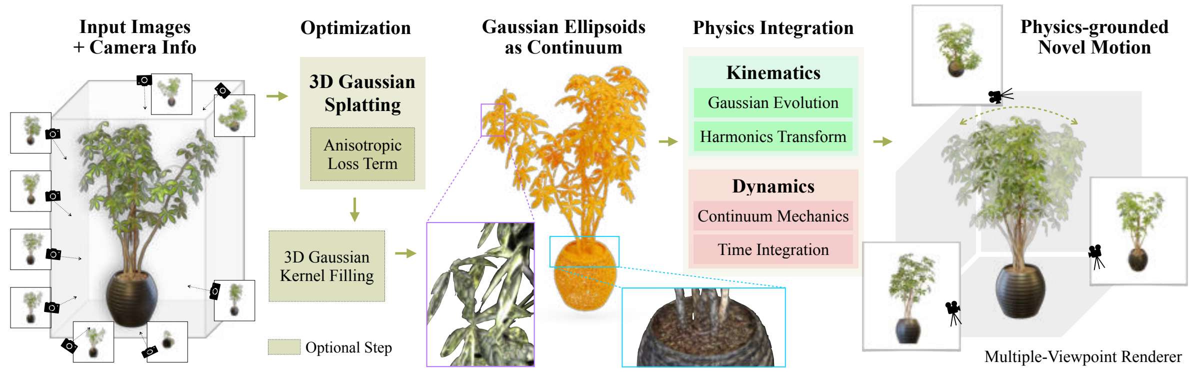


Figure 6.7: **Method Overview.** PhysGaussianis a unified simulation-rendering pipeline that incorporates 3D Gaussian splatting representation and continuum mechanics to generate physics-based dynamics and photo-realistic renderings simultaneously and seamlessly.

and dynamic scene reconstruction (Peng et al., 2022; Yuan et al., 2022; Chen et al., 2022a; Qiao et al., 2022, 2023; Liu et al., 2023b). Additionally, Yuan et al. (2022); Qiao et al. (2022); Liu et al. (2023b) have contributed to the incorporation of physics-based deformations into the NeRF framework. However, the effectiveness of these methodologies relies on the usage of exported meshes derived from NeRFs. To circumvent this restriction, explicit geometric representations have been explored for forward displacement modeling (Xu et al., 2022; Kerbl et al., 2023). In particular, Chen et al. (2023); Luiten et al. (2023); Yang et al. (2023a); Wu et al. (2023); Yang et al. (2023b) directly manipulate NeRF fields. Li et al. (2023) extend this approach by including physical simulators to achieve more dynamic behaviors. In this study, we leverage the explicit 3D Gaussian Splatting ellipsoids as a unified representation for both physics and graphics. In contrast to previous dynamic GS frameworks, which either maintain the shapes of Gaussian kernels or learn to modify them, our approach uniquely leverages the first-order information from the displacement map (deformation gradient) to assist dynamic simulations. In this way, we are able to deform the Gaussian kernels and seamlessly integrate the simulation within the GS framework.

Material Point Method.

The Material Point Method (MPM) is a widely used simulation framework for a broad range of multi-physics phenomena (Hu et al., 2018). The inherent capability of the MPM system allows for topology changes and frictional interactions, making it suitable for simulating various materials, including but not limited to elastic objects, fluids, sand, and snow (Stomakhin et al., 2013; Jiang et al., 2015; Klár et al., 2016). MPM can also be expanded to simulate objects that possess codimensional characteristics (Jiang et al., 2017). In addition, the efficacy of utilizing GPU(s) to accelerate MPM implementations has also been demonstrated by Gao et al. (2018b); Hu et al. (2019a); Wang et al. (2020); Qiu et al. (2023). Owing to its well-documented advantages, we employ the MPM to support the latent physical dynamics. This choice allows us to efficiently import dynamics into various scenarios with a shared particle representation alongside the Gaussian Splatting framework.

6.2.3 Method Overview

We propose PhysGaussian (Fig. 6.7), a unified simulation-rendering framework for generative dynamics based on continuum mechanics and 3D GS. Adopted from (Kerbl et al., 2023), we first reconstruct a GS representation of a static scene, with an optional anisotropic loss term to regularize over-skinny kernels. These Gaussians are viewed as the discretization of the scene to be simulated. Under our novel kinematics, we directly splat the deformed Gaussians for photo-realistic renderings. For better physics compliance, we also optionally fill the internal regions of objects. We detail these in this subsection.

6.2.3.1 3D Gaussian Splatting

3D Gaussian Splatting method (Kerbl et al., 2023) reparameterizes NeRF (Mildenhall et al., 2021) using a set of unstructured 3D Gaussian kernels $\{\boldsymbol{x}_p, \sigma_p, \boldsymbol{A}_p, \mathcal{C}_p\}_{p\in\mathcal{P}}$, where $\boldsymbol{x}_p$, σ_p, $\boldsymbol{A}_p$, and $\mathcal{C}_p$ represent the centers, opacities, covariance matrices, and spherical harmonic coefficients of the Gaussians, respectively. To render a view, GS projects these 3D Gaussians onto the image plane as 2D Gaussians, differing from traditional NeRF techniques that emit rays from the camera. The final color of each pixel is computed as

$$\boldsymbol{C} = \sum_{k \in \mathcal{P}} \alpha_k \mathrm{SH}(\boldsymbol{d}_k; \mathcal{C}_k) \prod_{j=1}^{k-1} (1 - \alpha_j). \tag{6.1}$$

Here α_k represents the z-depth ordered effective opacities, *i.e.*, products of the 2D Gaussian weights and their overall opacities σ_k; $\boldsymbol{d}_k$ stands for the view direction from the camera to $\boldsymbol{x}_k$. Per-view optimizations are performed using L_1 loss and SSIM loss. This explicit representation of the scene not only significantly accelerates training and rendering speeds, but also enables direct manipulation of the NeRF scene. The data-driven dynamics are supported by making $\boldsymbol{x}_p, A_p$ time-dependent (Wu et al., 2023) and minimizing rendering losses over videos. In §6.2.3.1, we show that this time-dependent evolution can be given by the continuum deformation map.

6.2.3.2 Continuum Mechanics

Continuum mechanics describes motions by a time-dependent continuous deformation map $\boldsymbol{x} = \boldsymbol{\phi}(\boldsymbol{X}, t)$ between the undeformed material space Ω^0 and the deformed world space Ω^t at time t. The deformation gradient $F(\boldsymbol{X}, t) = \nabla_{\boldsymbol{X}} \boldsymbol{\phi}(\boldsymbol{X}, t)$ encodes local transformations including stretch, rotation, and shear (Bonet and Wood, 1997). The evolution of the deformation $\boldsymbol{\phi}$ is governed by the conservation of mass and momentum. Conservation of mass ensures that the mass within any infinitesimal region $B_\epsilon^0 \in \Omega^0$ remains constant over time:

$$\int_{B_\epsilon^t} \rho(\boldsymbol{x}, t) \equiv \int_{B_\epsilon^0} \rho(\boldsymbol{\phi}^{-1}(\boldsymbol{x}, t), 0), \tag{6.2}$$

where $B_\epsilon^t = \boldsymbol{\phi}(B_\epsilon^0, t)$ and $\rho(\boldsymbol{x}, t)$ is the density field characterizing material distribution. Denoting the velocity field with $\boldsymbol{v}(\boldsymbol{x}, t)$, the conservation of momentum is given by

$$\rho(\boldsymbol{x}, t) \dot{\boldsymbol{v}}(\boldsymbol{x}, t) = \nabla \cdot \boldsymbol{\sigma}(\boldsymbol{x}, t) + \boldsymbol{f}^{\mathrm{ext}}, \tag{6.3}$$

where $\boldsymbol{\sigma} = \frac{1}{\det(\boldsymbol{F})} \frac{\partial \Psi}{\partial \boldsymbol{F}}(\boldsymbol{F}^E) \boldsymbol{F}^{E^T}$ is the Cauchy stress tensor associated with a hyperelastic energy density $\Psi(\boldsymbol{F})$, and $\boldsymbol{f}^{\mathrm{ext}}$ is the external force per unit volume (Bonet and Wood, 1997; Jiang et al., 2016). Here the total deformation gradient can be decomposed into an elastic part and a plastic part $\boldsymbol{F} = \boldsymbol{F}^E\boldsymbol{F}^P$ to support permanent rest shape changes caused by plasticity. The evolution of $\boldsymbol{F}^E$ follows some specific plastic flow such that it is always constrained within a predefined elastic region (Bonet and Wood, 1997).

6.2.3.3 Material Point Method

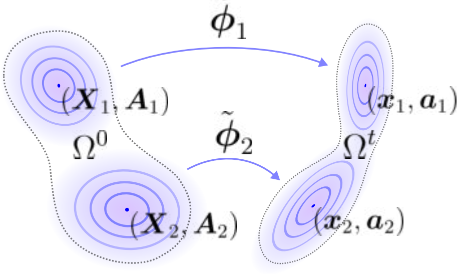
Material Point Method (MPM) solves the above governing equations by combining the strengths of both Lagrangian particles and Eulerian grids (Stomakhin et al., 2013; Jiang et al., 2016). The continuum is discretized by a collection of particles, each representing a small material region. These particles track several time-varying Lagrangian quantities such as position $\boldsymbol{x}_p$, velocity $\boldsymbol{v}_p$, and deformation gradient $\boldsymbol{F}_p$. The mass conservation in Lagrangian particles ensures the constancy of total mass during movement. Conversely, momentum conservation is more natural in Eulerian representation, which avoids mesh construction. We follow Stomakhin et al. (2013) to integrate these representations using C^1 continuous B-spline kernels for two-way transfer. From time step t^n to t^{n+1}, the momentum conservation, discretized by the forward Euler scheme, is represented as

$$\frac{m_i}{\Delta t}(\boldsymbol{v}_i^{n+1} - \boldsymbol{v}_i^n) = -\sum_p V_p^0 \frac{\partial \Psi}{\partial \boldsymbol{F}}(\boldsymbol{F}_p^{E,n})\boldsymbol{F}_p^{E,n^T} \nabla w_{ip}^n + \boldsymbol{f}_i^{ext}. \tag{6.4}$$

Here i and p represent the fields on the Eulerian grid and the Lagrangian particles respectively; w_{ip}^n is the B-spline kernel defined on i-th grid evaluated at $\boldsymbol{x}_p^n$; V_p^0 is the initial representing volume, and Δt is the time step size. The updated grid velocity field $\boldsymbol{v}_i^{n+1}$ is transferred back onto particle to v_p^{n+1}, updating the particles' positions to $\boldsymbol{x}_p^{n+1} = \boldsymbol{x}_p^n + \Delta t \boldsymbol{v}_p^{n+1}$. We track $\boldsymbol{F}^E$ rather than both $\boldsymbol{F}$ and $\boldsymbol{F}^P$ (Simo and Hughes, 2006), which is updated by $\boldsymbol{F}_p^{E,n+1} = (\boldsymbol{I} + \Delta t \nabla \boldsymbol{v}_p)\boldsymbol{F}_p^{E,n} = (\boldsymbol{I} + \Delta t \sum_i \boldsymbol{v}_i^{n+1} \nabla w_{ip}^{n\,T})\boldsymbol{F}_p^{E,n}$ and regularized by an additional return mapping to support plasticity evolution: $\boldsymbol{F}_p^{E,n+1} \leftarrow \mathcal{Z}(\boldsymbol{F}_p^{E,n+1})$. Different plasticity models define different return mappings. We refer to the supplemental document for details of the simulation algorithm and different return mappings.

6.2.3.4 Physics-Integrated 3D Gaussians

We treat Gaussian kernels as discrete particle clouds to spatially discretize the simulated continuum. As the continuum deforms, we let the Gaussian kernels deform as well. However, for a Gaussian kernel defined at $\boldsymbol{X}_p$ in the material space, $G_p(\boldsymbol{X}) = e^{-\frac{1}{2}(\boldsymbol{X}-\boldsymbol{X}_p)^T \boldsymbol{A}_p^{-1}(\boldsymbol{X}-\boldsymbol{X}_p)}$, the deformed kernel under the deformation map $\boldsymbol{\phi}(\boldsymbol{X}, t)$,


$\tilde{\phi}_1$
$(\boldsymbol{X}_1, \boldsymbol{A}_1)$
$(\boldsymbol{x}_1, \boldsymbol{a}_1)$
Ω^0
$\tilde{\phi}_2$
Ω^t
$(\boldsymbol{X}_2, \boldsymbol{A}_2)$
$(\boldsymbol{x}_2, \boldsymbol{a}_2)$

$$G_p(\boldsymbol{x}, t) = e^{-\frac{1}{2}(\boldsymbol{\phi}^{-1}(\boldsymbol{x},t)-\boldsymbol{X}_p)^T \boldsymbol{A}_p^{-1}(\boldsymbol{\phi}^{-1}(\boldsymbol{x},t)-\boldsymbol{X}_p)} \tag{6.5}$$

is not necessarily Gaussian in the world space, which violates the requirements of the splatting process. Fortunately, if we assume particles undergo local affine transformations characterized by the first-order approximation

$$\tilde{\phi}_p(\boldsymbol{X}, t) = \boldsymbol{x}_p + \boldsymbol{F}_p(\boldsymbol{X} - \boldsymbol{X}_p), \tag{6.6}$$

the deformed kernel becomes Gaussian as desired:

$$G_p(\boldsymbol{x}, t) = e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{x}_p)^T (\boldsymbol{F}_p \boldsymbol{A}_p \boldsymbol{F}_p^T)^{-1}(\boldsymbol{x}-\boldsymbol{x}_p)}. \tag{6.7}$$

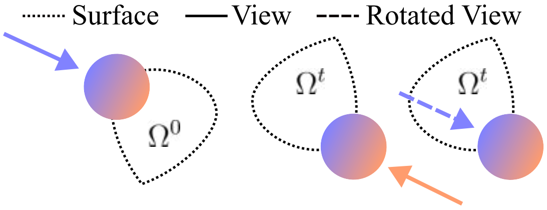
This transformation naturally provides a time-dependent version of $\boldsymbol{x}_p$ and $\boldsymbol{A}_p$ for the 3D GS framework:

$$\begin{aligned} \boldsymbol{x}_p(t) &= \boldsymbol{\phi}(\boldsymbol{X}_p, t), \\ \boldsymbol{a}_p(t) &= \boldsymbol{F}_p(t) \boldsymbol{A}_p \boldsymbol{F}_p(t)^T. \end{aligned} \tag{6.8}$$

In summary, given the 3D GS of a static scene $\{\boldsymbol{X}_p, \boldsymbol{A}_p, \sigma_p, \mathcal{C}_p\}$, we use simulation to dynamize the scene by evolving these Gaussians to produce dynamic Gaussians $\{\boldsymbol{x}_p(t), \boldsymbol{a}_p(t), \sigma_p, \mathcal{C}_p\}$. Here we assume that the opacity and the coefficients of spherical harmonics are invariant over time, but the harmonics will be rotated as discussed in the next subsection. We also initialize other physical quantities in Eq. (6.4): the representing volume of each particle V_p^0 is initialized as background cell volume divided by the number of contained particles; the mass m_p is then inferred from user-specified density ρ_p as $m_p = \rho_p V_p^0$. To render these deformed Gaussian kernels, we use the splatting from the original GS framework (Kerbl et al., 2023). It should be highlighted that the integration of physics into 3D Gaussians is **seamless**: on the one hand, the Gaussians themselves are viewed as the discretization of the continuum, which can be simulated directly; on the other hand, the deformed Gaussians can be directly rendered by the splatting procedure, avoiding the need for commercial rendering software in traditional animation pipelines. Most importantly, we can directly simulate scenes reconstructed from real data, achieving WS^2.

6.2.3.5 Evolving Orientations of Spherical Harmonics

Rendering the world-space 3D Gaussians can already obtain high-quality results. However, when the object undergoes rotations, the spherical harmonic bases are still represented in the material space, resulting in varying appearances even if the view direction is relatively fixed to the object. The solution is simple: when an ellipsoid is rotated over time, we rotate the orientations of its spherical harmonics as well. However, the bases are hard-coded inside the GS framework. We equivalently achieve this evolution by applying inverse rotation to view directions. This effect is illustrated in the inset figure. We remark that the rotation of view directions is not considered by Wu et al. (2023). Chen et al. (2023) tackle this issue in the Point-NeRF framework, but requires tracking of surface orientation. In our framework, the local rotation is readily obtained in the deformation gradient $\boldsymbol{F}_p$. Denote $f^0(\boldsymbol{d})$ as a spherical harmonic basis in material space, with $\boldsymbol{d}$ being a point on the unit sphere (indicating view direction). The polar decomposition, $\boldsymbol{F}_p = \boldsymbol{R}_p\boldsymbol{S}_p$, leads us to the rotated harmonic basis:



$$f^t(\boldsymbol{d}) = f^0(\boldsymbol{R}^T\boldsymbol{d}). \tag{6.9}$$

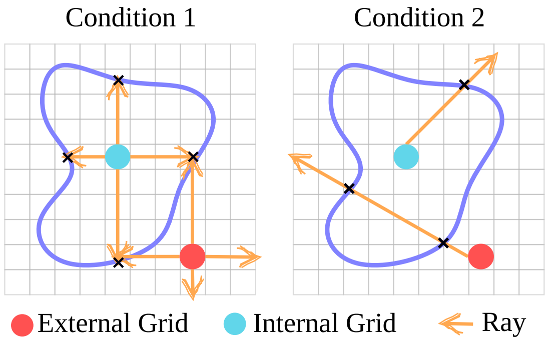
6.2.3.6 Incremental Evolution of Gaussians

We also propose an alternative way for Gaussian kinematics that better fits the updated Lagrangian framework, which avoids the dependency on the total deformation gradient $\boldsymbol{F}$. This approach also paves the way for physical material models that do not rely on employing $\boldsymbol{F}$ as the strain measure. Following conventions from computational fluid dynamics (McKIVER and Dritschel, 2003; Chandrasekhar, 1967), the update rule for the world-space covariance matrix $\boldsymbol{a}$ can also be derived by discretizing the rate form of kinematics $\dot{\boldsymbol{a}} = (\nabla \boldsymbol{v})\boldsymbol{a} + \boldsymbol{a}(\nabla \boldsymbol{v})^T$:

$$\boldsymbol{a}_p^{n+1} = \boldsymbol{a}_i^n + \Delta t(\nabla \boldsymbol{v}_p \boldsymbol{a}_p^n + \boldsymbol{a}_p^n \nabla \boldsymbol{v}_p^T). \tag{6.10}$$

This formulation facilitates the incremental update of the Gaussian kernel shapes from time step t^n to t^{n+1} without the need to obtain $\boldsymbol{F}_p$. The rotation matrix $\boldsymbol{R}_p$ of each spherical harmonics basis can be incrementally updated in a similar manner. Starting from $\boldsymbol{R}_p^0 = \boldsymbol{I}$, we extract the rotation matrix $\boldsymbol{R}_p^{n+1}$ from $(\boldsymbol{I} + \Delta t \boldsymbol{v}_p)\boldsymbol{R}_p^n$ using the polar decomposition.

6.2.3.7 Internal Filling



The internal structure is occluded by the object's surface, as the reconstructed Gaussians tend to distribute near the surface, resulting in inaccurate behaviors of volumetric objects. To fill particles into the void internal region, inspired by Tang et al. (2023), we borrow the 3D opacity field from 3D Gaussians

$$d(\boldsymbol{x}) = \sum_p \sigma_p \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_p)^T \boldsymbol{A}_p^{-1}(\boldsymbol{x} - \boldsymbol{x}_p)\right). \tag{6.11}$$

This continuous field is discretized onto a 3D grid. To achieve robust internal filling, we first define the concept of “intersubsection” within the opacity field, guided by a user-defined threshold σ_{th}. Specifically, we consider it an intersubsection when a ray passes from a lower-opacity grid ($\sigma_i < \sigma_{th}$) to a higher-opacity one ($\sigma_j > \sigma_{th}$). Based on this definition, we identify candidate grids by casting rays along 6 axes and checking inter-subsections (condition 1). Rays originating from internal cells will always intersect with the surface. To further refine our selection of candidate grids, we employ an additional ray to assess the intersubsection number (condition 2), thus ensuring greater accuracy.

Visualization of these internal particles is also crucial as they may get exposed due to large deformation. Those filled particles inherit $\sigma_p, \mathcal{C}_p$ from their closet Gaussian kernels. Each particle's covariance matrix is initialized as $\text{diag}(r_p^2, r_p^2, r_p^2)$, where r is the particle radius calculated from its volume: $r_p = (3V_p^0/4\pi)^{\frac{1}{3}}$.

6.2.3.8 Anisotropy Regularizer

The anisotropy of Gaussian kernels increases the efficiency of 3D representation, while over-skinny kernels may point outward from the object surface under large deformations, leading to unexpected plush artifacts. We propose the following training loss during 3D Gaussian reconstruction:

$$\mathcal{L}_{aniso} = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \max\{\max(\boldsymbol{S}_p)/\min(\boldsymbol{S}_p), r\} - r, \tag{6.12}$$

where $\boldsymbol{S}_p$ are the scalings of 3D Gaussians (Kerbl et al., 2023). This loss essentially constrains that the ratio between the major axis length and minor axis length does not exceed r. If desired, this term can be added to the training loss.

6.2.3.9 Elasticity and Plasticity Models

We adopt the constitutive models used by Zong et al. (2023). We list the models used for each scene in Table 6.1.

In all plasticity models used in our work, the deformation gradient is multiplicatively

Scene	Figure	Constitutive Model
Vasedeck	Fig. 6.6 & Fig. 6.9	Fixed corotated
Ficus	Fig. 6.7	Fixed corotated
Fox	Fig. 6.6 & Fig. 6.8	Fixed corotated
Plane	Fig. 6.8	von Mises
Toast	Fig. 6.8	Fixed corotated
Ruins	Fig. 6.8	Drucker-Prager
Jam	Fig. 6.8	Herschel-Bulkley
Sofa Suite	Fig. 6.8	Fixed corotated
Materials	Fig. 6.12	Fixed corotated
Microphone	Fig. 6.13	Neo-Hookean
Bread	Fig. 6.9	Fixed corotated
Cake	Fig. 6.9	Herschel-Bulkley
Can	Fig. 6.9	von Mises
Wolf	Fig. 6.9	Drucker-Prager

Table 6.1: **PhysGaussian Demo Model Setups.**

decomposed into $\boldsymbol{F} = \boldsymbol{F}^E \boldsymbol{F}^P$ following some yield stress condition. A hyperelastic constitutive model is applied to $\boldsymbol{F}^E$ to compute the Kirchhoff stress $\boldsymbol{\tau}$. For a pure elastic continuum, we simply take $\boldsymbol{F}^E = \boldsymbol{F}$.

6.2.3.10 Experiments

In this subsection, we show the versatility of our approach across a wide range of materials. We also evaluate the effectiveness of our method across a comprehensive suite of benchmarks.

6.2.3.11 Evaluation of Generative Dynamics

Datasets. We evaluate our method for generating diverse dynamics using several sources of input. In addition to the synthetic data (*sofa suite*) generated by BlenderNeRF (Raafat, 2023), we utilize *fox*, *plane*, and *ruins* from the datasets of Instant-NGP (Müller et al., 2022), Nerfstudio (Tancik et al., 2023) and the DroneDeploy NeRF (Pilkington, 2022), respectively. Furthermore, we collect two real-world datasets (referred to as *toast* and *jam*) with an iPhone. Each scene contains 150 photos. The initial point clouds and

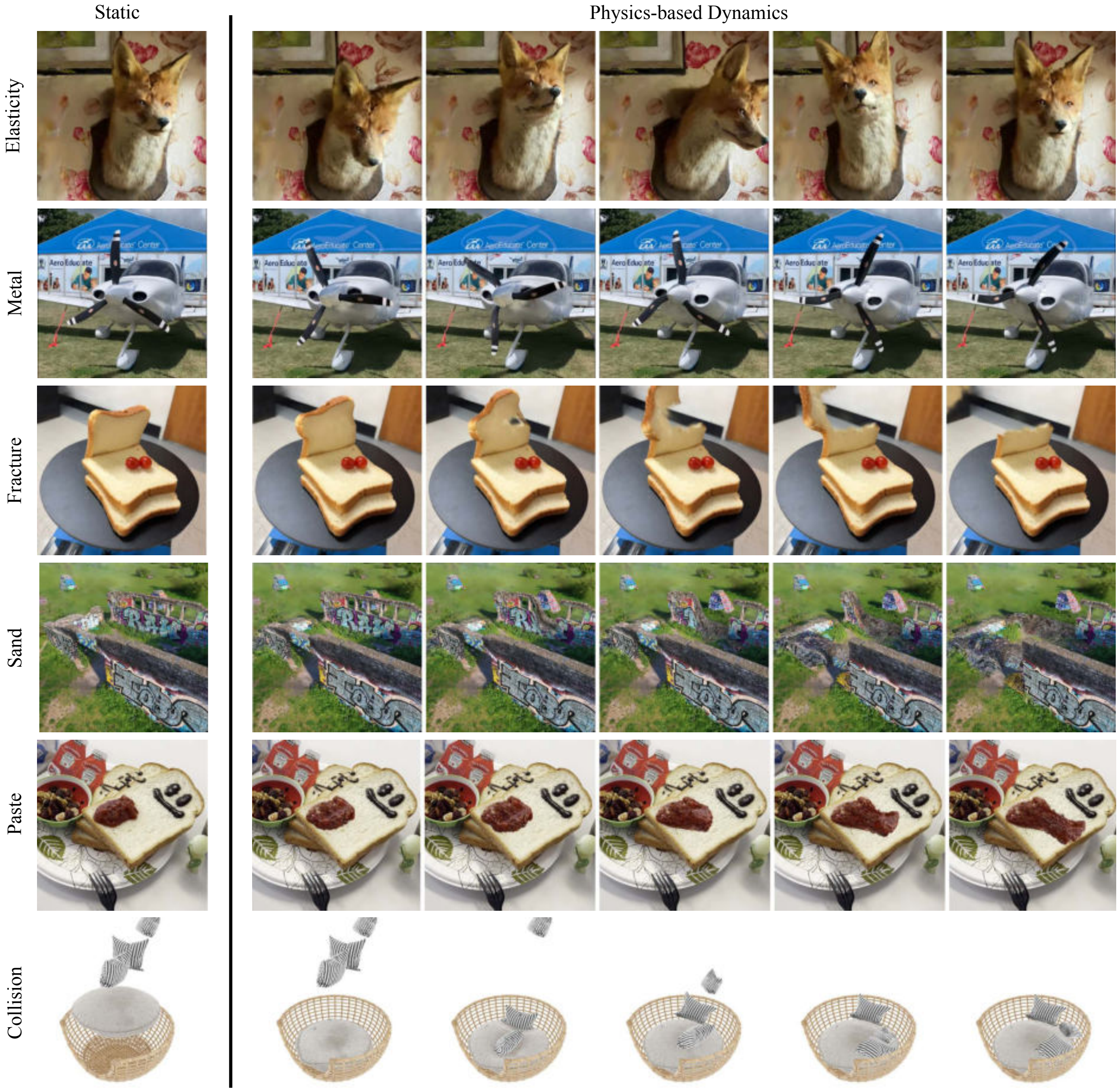


Figure 6.8: **Material Versatility.** We demonstrate the exceptional versatility of our approach across a wide variety of examples: *fox* (elastic entity), *plane* (plastic metal), *toast* (fracture), *ruins* (granular material), *jam* (viscoplastic material), and *sofa suite* (collision).

camera parameters are obtained using COLMAP (Schönberger et al., 2016; Schönberger and Frahm, 2016).

Simulation Setups. We build upon the MPM by Zong et al. (2023). To generate novel physics-based dynamics of a 3D Gaussian scene, we manually select a simulation region and normalize it to a cube with edge length 2. The internal particle filling can be performed before simulation. The cuboid simulation domain is discretized by a 3D dense grid. We selectively modify the velocities of specific particles to induce controlled movement. The remaining particles follow natural motion patterns governed by the established physical laws. All our experiments are performed on a 24-core 3.50GHz Intel i9-10920X machine with a Nvidia RTX 3090 GPU.

Results. We simulate a wide range of physics-based dynamics. For each type of dynamics, we visualize one example with its initial scene and deformation sequence, as shown in Fig. 6.8 and Fig. 6.9. The dynamics include: **Elasticity** refers to the property where the rest shape of the object remains invariant during deformation, representing the simplest form of daily-life dynamics. **Metal** can undergo permanent rest shape changes, which follows von-Mises plasticity model. **Fracture** is naturally supported by MPM simulation, where large deformations can cause particles to separate into multiple groups. **Sand** follows Druker-Prager plasticity model (Klár et al., 2016), which can capture granular-level frictional effects among particles. **Paste** is modeled as viscoplastic non-Newtonian fluid, adhering to Herschel-Bulkley plasticity model (Yue et al., 2015). **Collision** is another key feature of MPM simulation, which is automatically handled by grid time integration. Explicit MPM can be highly optimized to run on GPUs. We highlight that some of the cases can achieve real-time based on the 1/24-s frame duration: *plane* (30 FPS), *toast* (25 FPS) and *jam* (36 FPS).

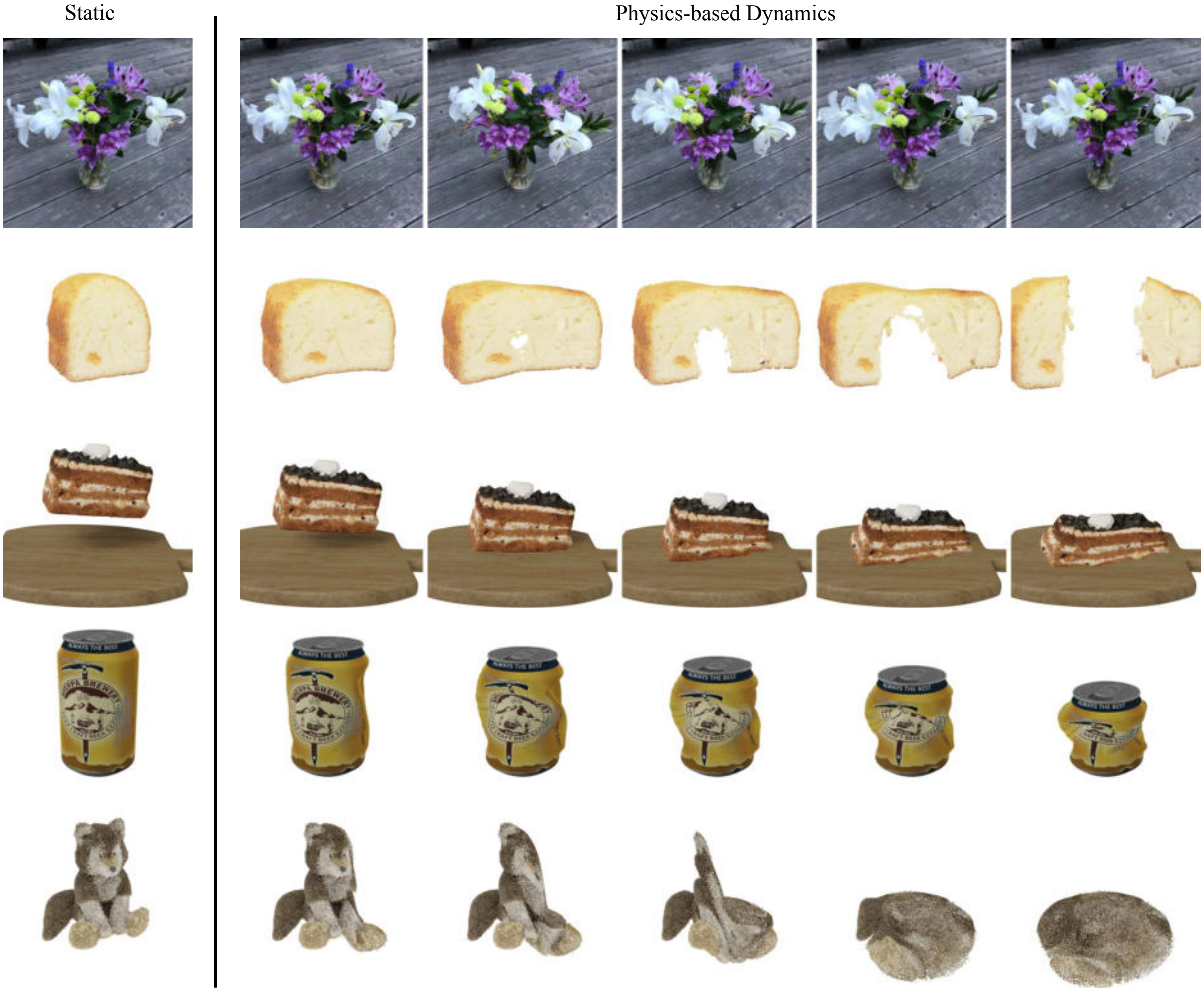


Figure 6.9: **Additional Evaluation.** Examples from top to bottom are: *vasedeck* (elastic entity), *bread* (fracture), *cake* (viscoplastic material), *can* (plastic metal) and *wolf* (granular material).

6.2.3.12 Lattice Deformation Benchmarks

Dataset. Due to the absence of ground truth for post-deformation, we utilize BlenderNeRF (Raafat, 2023) to synthesize several scenes, applying bending and twisting with the lattice deformation tool. For each scene, we create 100 multi-view renderings of the undeformed state for training, and 100 multi-view renderings of each deformed state to serve as ground truth for the deformed NeRFs. The lattice deformations are set as input to all methods for fair comparisons.

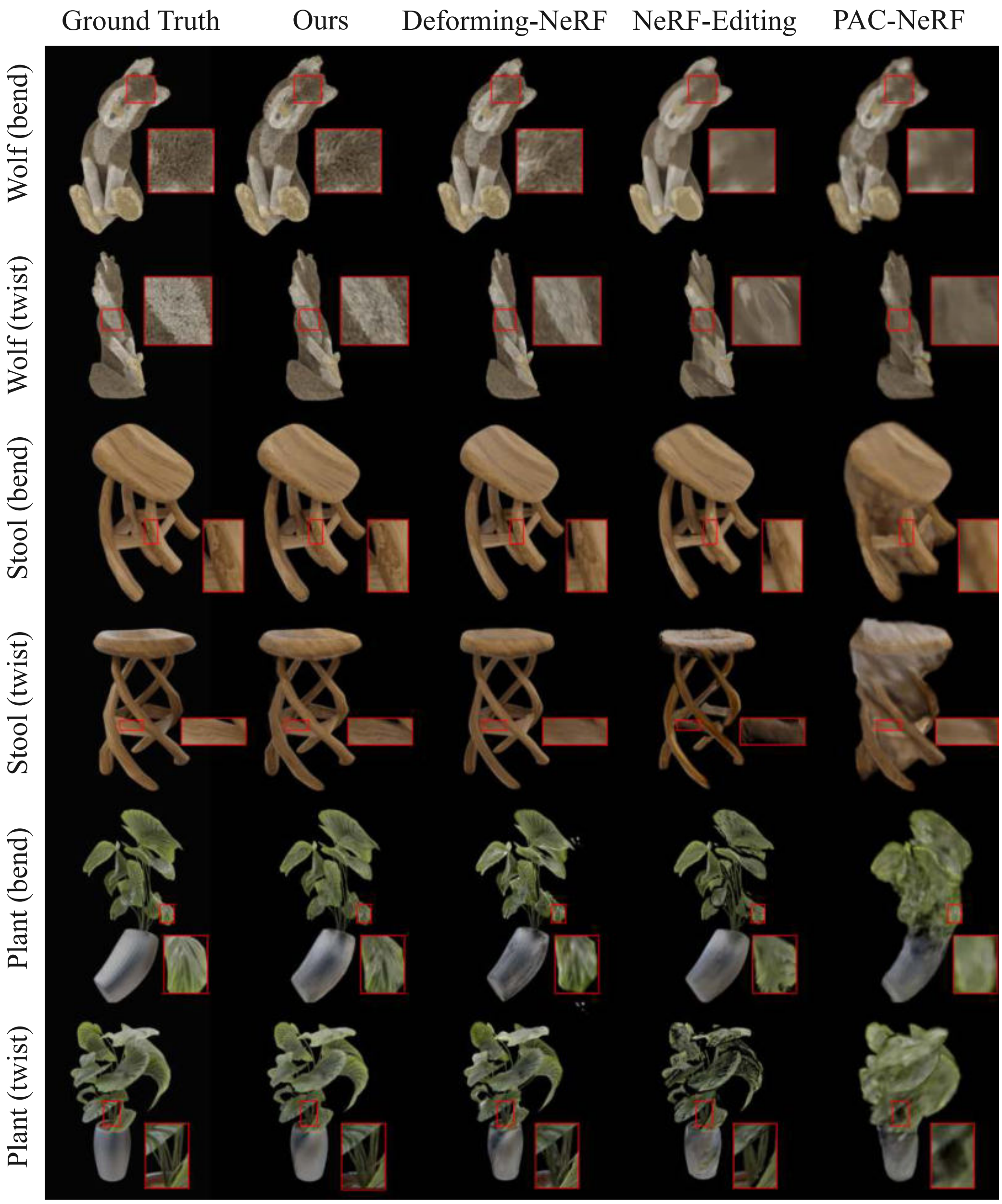


Figure 6.10: **Comparisons.** For each benchmark case, we select one test viewpoint and visualize all comparisons. We zoom in on some regions to highlight the ability of our method to maintain high-fidelity rendering quality after deformations. We use a black background to avoid interference from the background.

Comparisons. We compare our method with several state-of-the-art NeRF frameworks that support manual deformations: **1) NeRF-Editing (Yuan et al., 2022)** deforms NeRF using an extracted surface mesh, **2) Deforming-NeRF (Xu and Harada, 2022)** utilizes a cage mesh for deformation, and **3) PAC-NeRF (Li et al., 2023)** manipulates individual initial particles.

We show qualitative results in Fig. 6.10 and quantitative results in Table 6.2. NeRF-Editing uses NeuS (Wang et al., 2021) as the scene representation, which is more suited for surface reconstructions than high-fidelity renderings. Consequently, its rendering quality is inherently lower than that of Gaussian splitting. Furthermore, the deformation highly depends on the precision of the extracted surface mesh and the dilated cage mesh – an overly tight mesh might not encompass the entire radiance field, while an excessively large one could result in a void border, as observed in the twisting stool and plant examples. Deforming-NeRF, on the other hand, provides clear renderings, but its internal deformation, interpolated from the bounding cage vertices, does not perfectly match lattice deformations, limiting its ability to manipulate individual parts. PAC-NeRF is designed for simpler objects and textures in system identification tasks. While offering flexibility through its particle representation, it does not achieve high rendering fidelity. Our method utilizes both zero-order information (the deformation map) and first-order information (the deformation gradient) from each lattice cell. It outperforms the other methods across all cases, as high rendering qualities are well preserved after deformations. Although not primarily designed for editing tasks, this comparison showcases our method's significant potential for realistic editing of static NeRF scenes.

Ablation Studies We further conduct several ablation studies on these benchmark scenes to validate the necessity of the kinematics of Gaussian kernels and spherical harmonics: **1) Fixed Covariance** only translates the Gaussian kernels. **2) Rigid Covariance** only applys rigid transformations on the Gaussians, as assumed by Luiten et al. (2023). **3) Fixed Harmonics** does not rotate the orientations of spherical harmonics, as assumed by Wu et al. (2023).

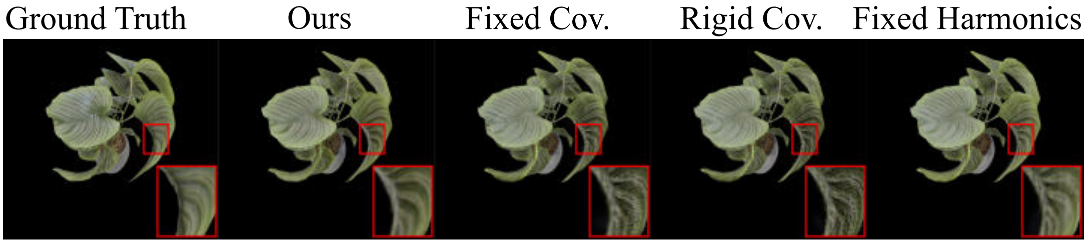


Figure 6.11: **Ablation Studies.** Non-extensible Gaussians can lead to severe visual artifacts during deformations. Although direct rendering of the deformed Gaussian kernels can already obtain good results, additional rotations on spherical harmonics can improve the rendering quality.

Test Case	**Wolf**		**Stool**		**Plant**	
Deformation Operator	Bend	Twist	Bend	Twist	Bend	Twist
NeRF-Editing (Yuan et al., 2022)	26.74	24.37	25.00	21.10	19.85	19.08
Deforming-NeRF (Xu and Harada, 2022)	21.65	21.72	22.32	21.16	17.90	18.63
PAC-NeRF (Li et al., 2023)	26.91	25.27	21.83	21.26	18.50	17.78
Fixed Covariance	26.77	26.02	29.94	25.31	23.95	23.09
Rigid Covariance	26.84	26.16	30.28	25.70	24.09	23.53
Fixed Harmonics	26.83	26.02	30.87	25.75	25.09	23.69
Ours	**26.96**	**26.46**	**31.15**	**26.15**	**25.81**	**23.87**

Table 6.2: We synthesize a lattice deformation benchmark dataset to compare with baselines and conduct ablation studies to validate our design choices. PSNR scores are reported (higher is better). Our method outperforms the others across all cases.

Here we visualize one example in Fig. 6.11. We can observe that Gaussian will not properly cover the surface after deformation if they are non-extensible, leading to severe visual artifacts. Enabling the rotation of spherical harmonics can slightly improve the consistency with the ground truth. We include quantitative results on all test cases in Table 6.2, which shows that all these enhancements are needed to achieve the best performance of our method.

6.2.3.13 Additional Qualitative Studies

Internal Filling. Typically, the 3D Gaussian splatting framework focuses on the surface appearance of objects and often fails to capture their internal structure. Consequently, the interior of the modeled object remains hollow, resembling a mere shell. This is usually not

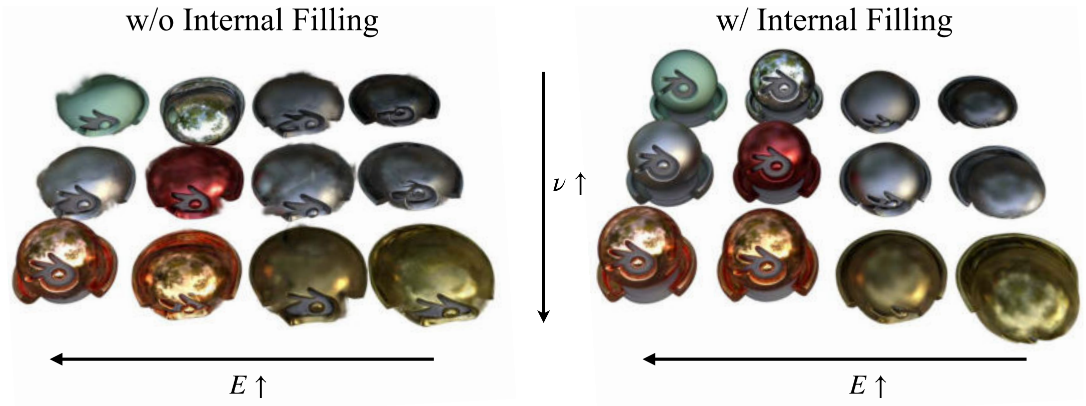


Figure 6.12: **Internal filling** enables more realistic simulation results. Our method also supports flexible control of dynamics via material parameters. A larger Young's modulus E indicates higher stiffness while a larger poission ratio ν leads to better volume preservation.

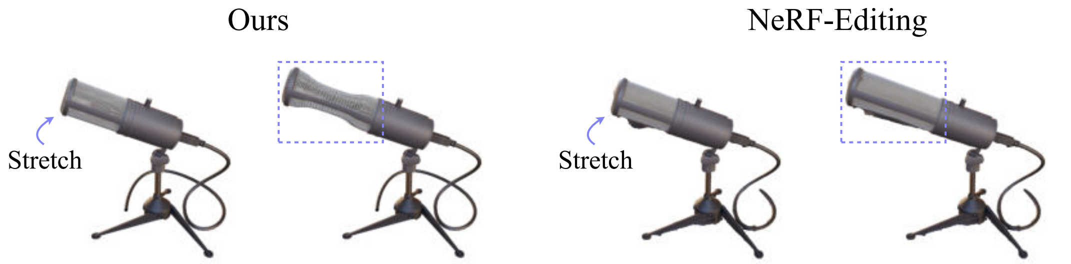


Figure 6.13: **Volume Conservation.** Compared to the geometry-based editing method (Yuan et al., 2022), our physics-based method is able to capture volumetric behaviors, leading to more realistic dynamics.

true in the real world, leading to unrealistic simulation results. To address this challenge, we introduce an internal filling method leveraging a reconstructed density field, which is derived from the opacity of Gaussian kernels. Fig. 6.12 showcases our simulation results with varying physical parameters. Objects devoid of internal particles tend to collapse when subjected to gravity forces, irrespective of the material parameters used. In contrast, our approach, assisted by internal filling methods, allows for nuanced control over object dynamics, effectively adjusting to different material characteristics.

Volume Conservation. Existing approaches to NeRF manipulation focus primarily on geometric adjustments without incorporating physical properties. A key attribute of real-world objects is their inherent ability to conserve volume during deformation. In

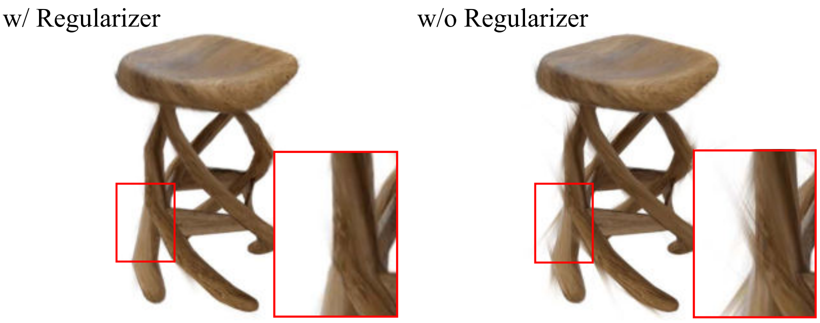


Figure 6.14: **Anisotropy Regularizer.** We introduce an anisotropy constraint for Gaussian kernels, effectively enhancing the fidelity of the Gaussian-based representation under dynamic conditions.

Fig. 6.13, we conduct a comparison study between our method and NeRF-Editing (Yuan et al., 2022), which utilizes surface As-Rigid-As-Possible (ARAP) deformation (Sorkine and Alexa, 2007). Unlike NeRF-Editing, our approach accurately captures and maintains the volume of the deformed objects.

Anisotropy Regularizer. 3D Gaussian models inherently represent anisotropic ellipsoids. However, excessively slender Gaussian kernels can lead to burr-like visual artifacts, especially pronounced during large deformations To tackle this issue, we introduce an additional regularization loss Eq. (6.12) to constrain anisotropy. As demonstrated in Fig. 6.14, this additional loss function effectively mitigates the artifacts induced by extreme anisotropy.

6.2.4 Discussion

Conclusion. This subsection introduces PhysGaussian, a unified simulation-rendering pipeline that generates physics-based dynamics and photo-realistic renderings simultaneously and seamlessly.

Limitations. The evolution of shadows is not considered in our framework. Additionally, we use a one-point quadrature for integrating volume integrals, which may not adequately represent the size of individual Gaussian ellipsoids. MPM with high-order quadratures may be adopted (Dritschel et al., 2004). Furthermore, while PDE-based dynamics offer a useful approximation, incorporating neural networks learned from real data or data-driven dynamics could provide more realistic modeling (Li et al., 2022a; Ma et al., 2023). Future work will also explore handling more versatile materials like liquids and integrating more intuitive user controls, possibly leveraging advancements in the Large Language Models (LLMs).

CHAPTER 7

Conclusions

In the realm of Computer Graphics and Scientific Computing, the accurate simulation of the dynamics of solids and fluids has lead to significant advancements in both theoretical understanding and practical applications. In this thesis, we explored the extensive capabilities and applications of the Material Point Method (MPM), a computational technique that stands at the forefront of physics-based simulation, aiming to enhance its efficiency, scalability, and range of application. Our key contributions are summarized below:

Enhanced Multi-GPU MPM. We have extended the capabilities of MPM on single and multiple GPU systems. The introduction of an optimized Grid-to-Particle-to-Grid MPM pipeline and the Array-of-Structure-of-Array particle data structure has dramatically improved the computational performance, achieving 2× to 3× speed enhancements on the single-GPU platform. The adaptation to multi-GPU environments further breaks the barriers of scale, enabling simulations with millions of particles in around one minute per frame.

Distributed Gigantic Resolution MPM. By developing a device-portable, distributed simulation system, we have addressed the memory and computational limitations of single machines. This system not only optimizes resource utilization, but also showcases flexibility across different simulation methods, proving invaluable for large-scale simulations.

Real-Time Simulation Applications. The advancements in modern device architectures have paved the way for employing physics-based simulations and the MPM in real-time applications such as interactive environments for robot training and gaming. These applications demonstrate the practicality and adaptability of the MPM in scenarios subject to limited computing resources.

The MPM in AI-Driven Generation Tasks. The final frontier explored in this thesis is the integration of the MPM with AI, particularly in generative tasks. The development of a unified representation for simulations, image rendering, and natural language is a pioneering step toward comprehensive model training and testing. Our explorations have not only enhanced our understanding of physical simulations, but they have also opened up new avenues for AI-driven creative processes.

APPENDIX A

Multi-GPU MPM System Implementation

A.1 Compile-Time Settings

To maximize the performance, we use compile-time constants for both controls- and material-related settings. Below, we provide additional details on compile-time settings for reproduction purposes.

We set the maximum number of particle-per-cell to be 64 for all the experiments. Note that this setting is more than sufficient for most MPM simulations since the typical particle-per-cell is 8 when initializing scenes. We have not observed any violations in any examples we used in this chapter, and one can easily modify this setting in our code. However, the particles will be discarded if the particle number inside one cell exceeds the compile-time setting, leading to incorrect results.

We also present the maximum number of particle blocks and grid blocks, as well as the maximum particle number for the experiments. These settings are adopted to enable the pre-allocation of all spatial data structures. Still, we periodically check the current demand for memory and dynamically resize it to fulfill the need. Different scenes require different settings, and the program works as long as the whole memory allocated does not exceed the device's memory limit.

Another assumption is that the Courant-Friedrichs-Lewy (CFL) condition always holds during run-time, indicating that the particles would move at most one-cell distances in each time step. We use Courant number 0.6 to compute the CFL-bounded default stepping time in all the experiments. The material stiffness is also considered when computing the default stepping time for stability requirements. During run-time, we compute the maximum of the grid velocity and calculate a stepping time to ensure particles do not move more than one cell distance. The final stepping time is chosen as the minimum of the computed time among all devices and the default stepping time. The CFL condition is crucial for the correctness of the results, and the *G2P2G* kernel will crash if it is violated, leading to failures as would happen in traditional CPU and GPU solvers.

A.2 Hierarchical Data Structure Composition

Since the efficacy of the data structure is usually hardware- and algorithm-dependent, it often requires non-trivial engineering efforts to explore different choices. Therefore, the ability to quickly design and benchmark new data structures for a specific task can significantly reduce code complexity.

A.2.1 Data-Oriented Design Philosophy

Due to the increased overhead of memory operations, the data-oriented design philosophy (A., 2014) has been widely adopted in HPC. Following this design principle, Hu et al. (2019a) introduce a high-performance programming language, *Taichi*, wherein dedicated data structures can be developed by assembling components of different properties in static hierarchies. *Taichi* provides a powerful and easy-to-use toolchain for developing a wide range of high-performance applications. It implements an abstraction to define multi-level spatial data structures and kernel functions through a user-friendly Python front-end and a robust LLVM back-end that automatically handles memory, manages executions, and deploys to CPU or GPU.

Still, there are two major issues in *Taichi* that prevent us from directly adopting it when developing multi-GPU-tailored MPM algorithms: 1) no access to low-level operations, including CUDA warp intrinsics, and 2) lack of multi-GPU support. Therefore, in our implementation, we refer to the data structure description described in *Taichi* as the mini-language and build up the infrastructure within our C++ codebase with the following improvements.

C++ Oriented Programming. Unlike developing a new compiler as in the *Taichi* programming language, we intend to develop a tool that can be directly used in both native C++ and CUDA C++. The latest standard supported by CUDA is C++14; thus it becomes our final choice. Function definitions are decorated with 'constexpr' keyword whenever applicable on both the host- and the device side.

Structural Composition. The C++ template meta-programming is adopted to implement the infrastructure. Most setups, including hierarchy, layout, the relationship of elements, *etc.*, are known beforehand and can be statically specified as template parameters. Hence, the access interface and the internal composition of the customized data structure are specified.

Memory Management. The representations of memory handles vary across APIs for GPU computing. For CUDA C++, the memory handle of the device memory is simply a pointer on the host; the cost of copying is trivial. Thus, the memory handle can be value-copied to CUDA kernel functions from the host device. The specific type of memory (*e.g.*, unified virtual memory or device memory) that the variable is allocated with is determined by the allocator given at the run-time. The instance does not own the handle of the allocator so its lifetime could be managed by programmers explicitly.

In our C++ codebase, we follow the same principle emphasized by the data-oriented design principle: the internal data structure should be highly compositional and shielded under a set of high-level access interfaces. Specifically, *Structural Nodes* can be associated with child nodes recursively for multi-level hierarchy composition, and the accompanying *Decorator* specifies the property of the node itself. For high performance, most specifications of the structure are performed at compile-time. We provide these utilities through C++ variadic templates in the following form:

```
domain<Tn, Ns...>; // Tn: Index Type, Ns: Multi-dimensional coordinates of type Tn
enum attrib_layout{aos,soa};
enum structural_type{entity, hash, dense, dynamic};
decorator<structural_allocation_policy, structural_padding_policy, attrib_layout>;
```

```
structural<structural_type, domain, decorator, structurals...>;
```

A.2.2 C++ Implementation

The entire infrastructure consists of the four major components: *Domain*, *Decorator*, *Structural Node*, and *Structural Instance*. For more details, please refer to the opensourced code.

Domain *Domain* describes the range for the index of a data structure. It maps from multi-dimensional coordinates to a 1D memory span.

```
template<typename Tn, Tn Ns...>
struct domain {
    template<typename... Indices>
    static constexpr Tn offset(Indices&&... indices);
};
```

Decorator *Decorator* describes the auxiliary and detailed properties regarding the data structure it decorates.

```
enum class structural_allocation_policy : std::size_t {
  full_allocation = 0,
  on_demand = 1,
  ...
};
enum class structural_padding_policy : std::size_t {
  compact = 0,
  align = 1,
  ...
};
enum class attrib_layout : std::size_t {
  aos = 0,
  soa = 1,
  ...
};
template <structural_allocation_policy alloc_policy_,
          structural_padding_policy padding_policy_,
          attrib_layout layout_>
struct decorator {
  static constexpr auto alloc_policy = alloc_policy_;
  static constexpr auto padding_policy = padding_policy_;
  static constexpr auto layout = layout_;
};
```

Structural Node *Structural Nodes* with particular properties are formed in a hierarchy to compose a multi-level data structure. Currently, we support three types of structural nodes (*i.e.*, *hash*, *dense*, and *dynamic*), the same as in (Hu et al., 2019a).

```
enum class structural_type : std::size_t {
  /// leaf
  sentinel = 0,
  entity = 1,
  /// trunk
  hash = 2,
  dense = 3,
  dynamic = 4,
  ...
};
```

No matter what the internal relationship of elements is within a structural node (either contiguous- or node-based), we assume there is at least one contiguous chunk of physical memory to store the data; the size is a multiple of the extent of the *Domain* and the total size of all the attributes of an element.

```
/// attribute index of a structural node
using attrib_index = placeholder::placeholder_type;

/// traits of structural nodes
template <structural_type NodeType, typename Domain, typename Decoration, typename... Structurals>
struct structural_traits {
  using attribs = type_seq<Structurals...>;
  using self =
      structural<NodeType, Domain, Decoration, Structurals...>;
  template <attrib_index I>
  using value_type = ...;
  static constexpr auto attrib_count = sizeof...(Structurals);
  static constexpr std::size_t element_size = ...;
  static constexpr std::size_t element_storage_size = ...;
  /// for allocation
  static constexpr std::size_t size = domain::extent * element_storage_size;

  template <attrib_index AttribNo> struct accessor {
    static constexpr uintptr_t element_stride_in_bytes = ...;
    static constexpr uintptr_t attrib_base_offset = ...;
    template <typename... Indices>
    static constexpr uintptr_t coord_offset(Indices &&... is) {
      return attrib_base_offset + Domain::offset(std::forward<Indices>(is)...) * element_stride_in_bytes;
    }
    template <typename Index>
    static constexpr uintptr_t linear_offset(Index &&i) {
      return attrib_base_offset + std::forward<Index>(i) *
      element_stride_in_bytes;
    }
  };

  // manage memory
  template <typename Allocator> void allocate_handle(Allocator
    allocator) {
    if (self::size != 0)
      _handle.ptr = allocator.allocate(self::size);
    else
      _handle.ptr = nullptr;
  }
  template <typename Allocator> void deallocate(Allocator allocator) {
    allocator.deallocate(_handle.ptr, self::size);
    _handle.ptr = nullptr;
  }
  // access value
  template <attrib_index ChAttribNo, typename Type = value_type<
    ChAttribNo>, typename... Indices>
  constexpr auto &val(std::integral_constant<attrib_index, ChAttribNo>,
     Indices &&... indices) {
    return *reinterpret_cast<Type *>(_handle.ptrval + accessor<
    ChAttribNo>::coord_offset(std::forward<Indices>(indices)...));
  }
  template <attrib_index ChAttribNo, typename Type = value_type<
    ChAttribNo>, typename Index>
  constexpr auto &val_1d(std::integral_constant<attrib_index,
    ChAttribNo>,
                        Index &&index) {
    return *reinterpret_cast<Type *>(
        _handle.ptrval +
        accessor<ChAttribNo>::linear_offset(std::forward<Index>(index))
    );
  }
  /// data member
  MemResource _handle;
};
/// specializations of different types of structural nodes
template <typename Domain, typename Decoration, typename... Structurals
    >
struct structural<structural_type::hash, Domain, Decoration,
    Structurals...> : structural_traits<structural_type::hash, Domain,
    Decoration, Structurals...> {...};
...
```

We define two types of *Structural Nodes*, the root node and the leaf node, to form the hierarchy.

```
/// special structural node
template <typename Structural> struct root_instance;
template <typename T> struct structural_entity;
```

Structural Instance A variable defined by a *Structural Node* is an *Structural Instance* spawned given an allocator at the run-time. The instance is customizable (*e.g.*, accessing the parent node requires additional data) as it is assembled from data components.

```
enum class structural_component_index : std::size_t {
  default_handle = 0,
  parent_scope_handle = 1,
  ...
};
template <typename ParentInstance, attrib_index,
    structural_component_index>
struct structural_instance_component;
/// specializations for each data component
template <typename ParentInstance, attrib_index>
struct structural_instance_component<ParentInstance, attrib_index,
    structural_component_index::parent_scope_handle> {...};
...
```

Besides the data components, the *Structural Instance* also inherits from the *Structural Node* that specifies the properties of itself.

```
/// traits of structural instance, inherit from structural node
template <typename parent_instance, attrib_index AttribNo>
struct structural_instance_traits
    : parent_instance::attribs::template type<(std::size_t)AttribNo> {
  using self = typename parent_instance::attribs::type<(std::size_t)
    AttribNo>;
  using parent_indexer = typename parent_instance::domain::index;
  using self_indexer = typename self::domain::index;
};

/// structural instance, inherit from all data components and its
    traits (which is derived from structural node)
template <typename ParentInstance, attrib_index AttribNo, typename
    Components>
struct structural_instance;
template <typename ParentInstance, attrib_index AttribNo, std::size_t
    ... Cs>
struct structural_instance<ParentInstance, AttribNo,
                           std::integer_sequence<std::size_t, Cs...>>
    : structural_instance_traits<ParentInstance, AttribNo>,
      structural_instance_component<ParentInstance, AttribNo,
    static_cast<structural_component_index>(Cs)>... {
  using traits = structural_instance_traits<ParentInstance, AttribNo>;
  using component_seq = std::integer_sequence<std::size_t, Cs...>;
  using self_instance =
      structural_instance<ParentInstance, AttribNo, component_seq>;
  template <attrib_index ChAttribNo>
  using accessor = typename traits::template accessor<ChAttribNo>;

  // hierarchy traverse
  template <attrib_index ChAttribNo, typename... Indices>
  constexpr auto chfull(std::integral_constant<attrib_index, ChAttribNo
   >,
                        Indices &&... indices) const {
    ...
  }
  template <attrib_index ChAttribNo, typename... Indices>
  constexpr auto ch(std::integral_constant<attrib_index, ChAttribNo>,
                    Indices &&... indices) const {
    ...
  }
  template <attrib_index ChAttribNo, typename... Indices>
  constexpr auto chptr(std::integral_constant<attrib_index, ChAttribNo
   >,
                       Indices &&... indices) const {
    ...
  }
};
```

A.2.3 Examples

Here, we showcase usages of *Structural* in C++ by providing a set of examples that describes a GPU SPGrid.

Common Useful Definitions:

```
/// leaf node
using empty_ = structural_entity<void>;
using i32_ = structural_entity<int32_t>;
using f32_ = structural_entity<float>;

/// attribute index
namespace placeholder {
using placeholder_type = unsigned;
constexpr auto _0 = std::integral_constant<placeholder_type, 0>{};
...
}

/// default data components for constructing instances
using orphan_signature = std::integer_sequence<std::size_t,
   static_cast<std::size_t>(structural_component_index::default_handle)
   >;
```

Definition of GPU SPGrid:

```
// domain
using BlockDomain = domain<char, 4, 4, 4>;
using GridBufferDomain = domain<int, g_max_active_block>;
// decorator
using DefaultDecorator = decorator<structural_allocation_policy::
   full_allocation, structural_padding_policy::compact, attrib_layout::
   soa>;
```

```cpp
// structural node
using grid_block_ = structural<structural_type::dense, DefaultDecorator, BlockDomain, f32_, f32_, f32_, f32_>;
using grid_buffer_ = structural<structural_type::dynamic, DefaultDecorator, GridBufferDomain, grid_block_>;
```

Create Instances:

```cpp
template <typename Structural, typename Signature = orphan_signature>
using Instance = structural_instance<root_instance<Structural>, (attrib_index)0, Signature>;
template <typename Structural, typename Componenets, typename Allocator>
constexpr auto spawn(Allocator allocator) {
  auto ret = Instance<Structural, Componenets>{};
  ret.allocate_handle(allocator);
  return ret;
}
auto allocator = ...;
auto grid = spawn<grid_buffer_, orphan_signature>(allocator);
```

Access GPU SPGrid in a Function:

```cpp
/// acquire blockno-th grid block
auto grid_block = grid.ch(_0, blockno);
/// access cidib-th cell within this block
grid_block.val_1d(_0, cidib);   // access 0-th channel (mass)
/// access cell within by coordinates
grid_block.val(_1, cx, cy, cz);   // access 1-th channel (velocity x)
```

Memory Layout:

Two types of *Structural Nodes* with different *Decorators* are illustrated in Fig. A.1 to explain the underlying memory layout.

```
using Attr0 =
    structural_entity<float>;
using Attr1 =
    structural_entity<double>;
using DecoratorA = decorator<
    structural_allocation_policy::
    full_allocation,
    structural_padding_policy::align,
     attrib_layout:aos>;
using DecoratorB = decorator<
    structural_allocation_policy::
    full_allocation,
    structural_padding_policy::align,
     attrib_layout:soa>;
```

```
using StructuralA = structural<structural_type::dense, DecoratorA,
    domain<int, 4, 4>, Attr0, Attr1>;
using StructuralB = Structural<structural_type::dense, DecoratorB,
    domain<int, 4, 4>, Attr0, Attr1>;
```

Figure A.1: **Spatial structure specification.** Two *structurals* are specified with different *decorators*. The arrows connecting all elements indicate the ascending order in a contiguous chunk of memory. The *structural* can be used as a child of another *structural* to form a multi-level hierarchy. Elements displayed in the grid view are accessed by a child *structural* index (marked with different colors) and a coordinate within its domain. Note that the memory size of each *structural* is padded to the next power of 2 due to the alignment decoration.

APPENDIX B

Distributed MPM System Implementation

This appendix chapter shows some implementation details of the distributed MPM system we introduced in §4.

B.1 Dynamic Load Balancing Implementation

Here we present algorithm details and the unit tests of the proposed dynamic load balancing method. Also, a 2D example is shown in Figure B.2 to visualize the dynamic optimization process. We first compute the workload matrix locally on each rank and then use `MPI_ALLReduce` to get the global workload matrix. Then we compute the prefix summation for constant-time workload computation in any given rectangle region. After that, we perform iterations of 1D rectangle partition optimization. We first fix the position of the previous partition boundary p_{i-1}, then we move current partition boundary p_i to the right until finding the optimal partition position $\underset{p_i}{argmin} \sum_{jk} |w_{jk}^{p_{i-1}:p_i} - w_{jk}^{ave}|$.

B.2 Load Balancing Algorithm Validation

To validate the partition optimization algorithm, we design a grid/particle distribution that leads to a unique ground truth of the partition boundary set. Figure B.1 illustrates a 2D case of this unit test process. We first generate a ground truth partition by random sampling. Then we sample the same number of valid *grid cells* or particles inside each partitioned sub-domain. To ensure the uniqueness of this partition, two anchors *grid cell*s or particles are included and placed at the top-left and bottom-right corners. After that,

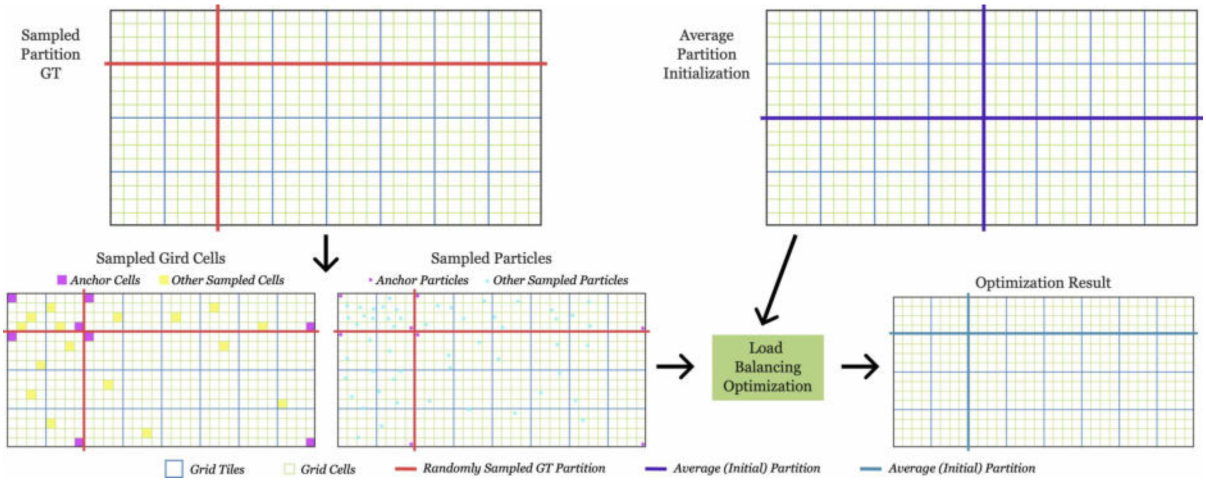


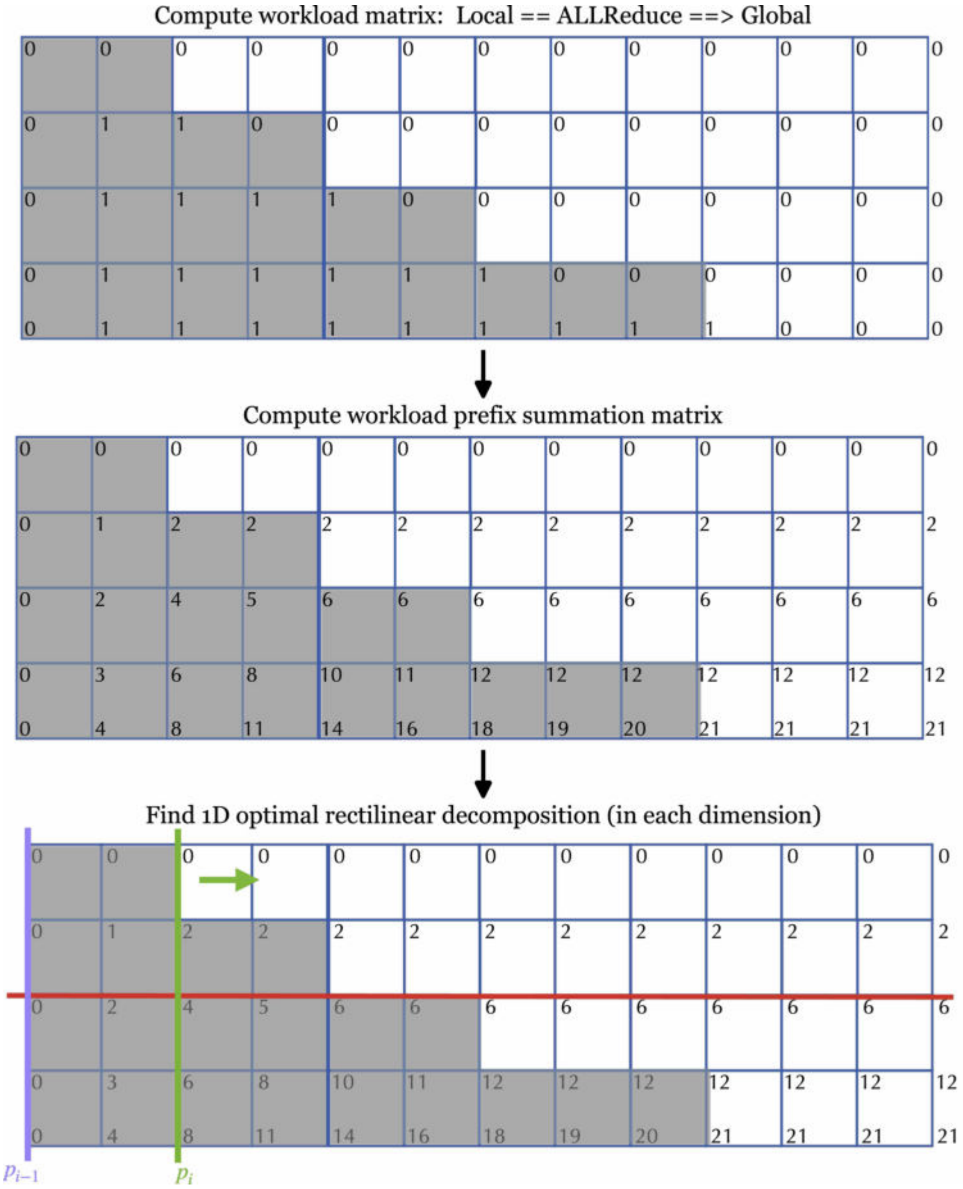
Figure B.1: **Load Balancing Unit Test.** In this 2D example, we sample six *grid cells* in each rank for *dynamic grid partition* test, and we sample 15 particles for *dynamic particle partition*. Each case will contain two fixed anchor *grid cells* or particles to ensure the uniqueness of the partition results.

we use the average partition for initialization and perform the dynamic load balancing algorithm. The testing results show that our partition optimization algorithm can converge to the ground truth within several (usually 1-4) optimization iterations.

B.3 Distributed MPM System Application Implementation

Here we show an example of setting up a scenario with collision objects in the *application level*.

```
// define the latent programming model, using CUDA as an example
using EXECSPACE = Kokkos::Cuda;
using MEMSPACE = Kokkos::CudaSpace;
using DEVICE = Kokkos::Device<Kokkos::Cuda, Kokkos::CudaSpace>;
// data type
using T = float;
// define particles
using particle_members =
    Cabana::MemberTypes<T, T[3], T[3], T[3][3], T[3][3], T>;
using particle_list = Cabana::AoSoA<particle_members, MEMSPACE>;
using particle_type = typename particle_list::tuple_type;
struct particle_index
{
    enum Names
    {
        mass = 0,
        pos = 1,
```



Compute workload matrix: Local == ALLReduce ==> Global

0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1	1	0	0	0

Compute workload prefix summation matrix

0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	2	2	2	2	2	2	2	2	2	2
0	2	4	5	6	6	6	6	6	6	6	6	6
0	3	6	8	10	11	12	12	12	12	12	12	12
0	4	8	11	14	16	18	19	20	21	21	21	21

Find 1D optimal rectilinear decomposition (in each dimension)

0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	2	2	2	2	2	2	2	2	2	2
0	2	4	5	6	6	6	6	6	6	6	6	6
0	3	6	8	10	11	12	12	12	12	12	12	12
0	4	8	11	14	16	18	19	20	21	21	21	21

Figure B.2: **Illustration of the dynamic load balancing algorithm.** In this 2D example, we visualize how the dynamic partition is performed on the entire simulation domain. Here we use the grid as the workload unit for illustration, where gray grids refer to activated grid nodes, and the numbers represent the element values of the matrix.

```
        vel = 2,
        F = 3,
        affine = 4,
        logJp = 5,
        total = 6,
    };
};
// basic parameter settings
namespace Settings
{
// domain corners
static constexpr T low_x = 0.0;
static constexpr T low_y = 0.0;
static constexpr T low_z = 0.0;
static constexpr T high_x = 200.0;
static constexpr T high_y = 200.0;
static constexpr T high_z = 200.0;
// spatial resolutions
static constexpr int res_x = 512;
static constexpr int res_y = 512;
static constexpr int res_z = 512;
static constexpr T dx = high_x / res_x;
// halo and partition control
static constexpr int halo_size = 4;
static constexpr int num_step_rebalance = 200;
static constexpr bool partition_op_on = true;
// boundary type
static constexpr MultiSim::BCTypes boundary_type = MultiSim::BCTypes::STICKY;
// temporal settings
static constexpr int frame_num = 170;
static constexpr T cfl = 0.3;
static constexpr T fps = 4;
// material related settings
static constexpr MultiSim::MaterialTypes material_type =
    MultiSim::MaterialTypes::FIX_COROTATED;
static constexpr T par_density = 1000.;
static constexpr T E = 9e6;
static constexpr T PR = 0.4;
// physical parameters
static constexpr T gravity = -9.8;
// particle info
static constexpr int PPC = 8;
static constexpr T par_volume =
    ( high_x / res_x ) * ( high_y / res_y ) * ( high_z / res_z ) / PPC;
static constexpr T par_mass = par_density * par_volume;
static constexpr T init_y_vel = 100;
// sourcing related info
static constexpr T source_init_vel = 50;
// static constexpr T source_init_vel = 13;
// static constexpr int source_step = 800;
static constexpr int source_step_0 = 1;
// static constexpr int source_step_1 = 297;
static constexpr int source_step_1 = 98;
static constexpr int source_step_2 = 199;
}; // end namespace Settings

// customized particle initialization
template <typename Scalar, class ExecSpace>
struct InitParticleFunc
{
    using execution_space = ExecSpace;
    using memory_spcae = MEMSPACE;

    template <class ParticleList>
    int operator()( ParticleList& particles, const Kokkos::Array<Scalar, g_dim>& local_low_corner, const Kokkos::Array<Scalar, g_dim>& local_high_corner, const Scalar cell_size, const int ppc )
    {
        // sample from Analytical Level Set
         MultiSim::Analytic_Shape::AnalyticLevelSet<MultiSim::Analytic_Shape::cuboid, Scalar, 3>
            cube( half_size, mid_pt, { 0, 0, 0 } );
        std::vector<std::array<T, 3>> points;
        MultiSim::Particle_Sample::Sampler<T, 3> sampler;
        int particle_num = sampler.sample_particle_pos( points, cube, ppc, cell_size );

        // sampled particles to Kokkos:View
        Kokkos::View<T* [3], memory_spcae> poses( "particles", particle_num );
        auto host_view = Kokkos::create_mirror_view( Kokkos::HostSpace(), poses );
        for ( int i = 0; i < particle_num; ++i )
            for ( int d = 0; d < 3; ++d )
                host_view( i, d ) = points[i][d];
        Kokkos::deep_copy( poses, host_view );

        // Initialize Particles
        using particle_type = typename ParticleList::tuple_type;
        using P = particle_index;
        particles.resize( particle_num );
        Kokkos::parallel_for( Kokkos::RangePolicy<execution_space>( 0, particle_num ),
            KOKKOS_LAMBDA( const int idx ) {
                particle_type p;
                // mass
                Cabana::get<P::mass>( p ) = Settings::par_mass;
                // pos
                Cabana::get<P::pos>( p, 0 ) = poses( idx, 0 );
                Cabana::get<P::pos>( p, 1 ) = poses( idx, 1 );
                Cabana::get<P::pos>( p, 2 ) = poses( idx, 2 );
                // vel
                for ( int d = 0; d < g_dim; ++d )
                    Cabana::get<P::vel>( p, d ) = _v[d];
                // F
                for ( int d0 = 0; d0 < g_dim; ++d0 )
                    for ( int d1 = 0; d1 < g_dim; ++d1 )
```

```
                        Cabana::get<P::F>( p, d0, d1 ) =
                            d0 == d1 ? (Scalar)1 : (Scalar)0;
                // C
                for ( int d0 = 0; d0 < g_dim; ++d0 )
                    for ( int d1 = 0; d1 < g_dim; ++d1 )
                        Cabana::get<P::affine>( p, d0, d1 ) = (Scalar)0;
                // logJp
                Cabana::get<P::logJp>( p ) = 0;
                // init particle
                particles.setTuple( idx, p );
            } );
        Kokkos::fence();
        return particle_num;
    }
};
// customized scene initialization
template <typename Scalar>
struct InitSceneFunc
{
    using data_type = Scalar;
    // set all
    template <class BoundaryCondition, class ProblemManager, class MeshType>
    void operator()( BoundaryCondition& bc, ProblemManager& pm_ptr, MeshType& mesh_ptr )
    {
        set_bc( bc, mesh_ptr );
        set_mat( pm_ptr );
        pm_ptr->set_gravity( Settings::gravity );
    }

  private:
    // set boundary condition - if each side has different settings
    template <class BoundaryCondition, class MeshType>
    std::enable_if_t<BoundaryCondition::bc_type == MultiSim::BCTypes::MIX, void>
    set_bc( BoundaryCondition& bc, MeshType& mesh_ptr )
    {
        using BCT = MultiSim::BCTypes;
        auto& gm = mesh_ptr->globalMeshPtr();
        std::array<BCT, g_dim * 2> bc_types;
        bc_types[0] = BCT::STICKY;
        bc_types[1] = BCT::STICKY;
        bc_types[2] = BCT::STICKY;
        bc_types[3] = BCT::STICKY;
        bc_types[4] = BCT::STICKY;
        bc_types[5] = BCT::NONE;
        bc.set_bc( bc_types[0], bc_types[1], bc_types[2], bc_types[3],
                   bc_types[4], bc_types[5], 0, 0, 0,
                   ( gm->highCorner( 0 ) - gm->lowCorner( 0 ) ) /
                       mesh_ptr->cell_size(),
                   ( gm->highCorner( 1 ) - gm->lowCorner( 1 ) ) /
                       mesh_ptr->cell_size(),
                   ( gm->highCorner( 2 ) - gm->lowCorner( 2 ) ) /
                       mesh_ptr->cell_size() );
    }
    // set boundary conditon - if all sides share the same setting
    template <class BoundaryCondition, class MeshType>
    std::enable_if_t<BoundaryCondition::bc_type != MultiSim::BCTypes::
   MIX, void>
    set_bc( BoundaryCondition& bc, MeshType& mesh_ptr )
    {
        auto& gm = mesh_ptr->globalMeshPtr();
        bc.set_bc( 0, 0, 0,
                   ( gm->highCorner( 0 ) - gm->lowCorner( 0 ) ) /
                       mesh_ptr->cell_size(),
                   ( gm->highCorner( 1 ) - gm->lowCorner( 1 ) ) /
                       mesh_ptr->cell_size(),
                   ( gm->highCorner( 2 ) - gm->lowCorner( 2 ) ) /
                       mesh_ptr->cell_size() );
    }
    // set material
    template <class ProblemManager>
    void set_mat( ProblemManager& pm_ptr )
    {
        auto& mat = pm_ptr->materialFunc();
        // material parameters
        mat.density = Settings::par_density;
        mat.ys = Settings::E;
        mat.pr = Settings::PR;
        mat.lambda = Settings::E * Settings::PR /
                     ( ( 1 + Settings::PR ) * ( 1 - 2 * Settings::PR )
   );
        mat.mu = Settings::E / ( 2 * ( 1 + Settings::PR ) );
        mat.volume = Settings::par_volume;
    }
};

void test_example()
{
    Kokkos::Array<T, g_dim * 2> global_bounding_box(
        { Settings::low_x, Settings::low_y, Settings::low_z, Settings::
   high_x,
          Settings::high_y, Settings::high_z } );
    std::array<int, g_dim> global_num_cell(
        { Settings::res_x, Settings::res_y, Settings::res_z } );
    // initializer
    InitParticleFunc<T, EXECSPACE> parpos_init_functor;
    InitSceneFunc<T> scene_init_functor;
    MultiSim::Init_Partitioner::InitUniformPartitionerFunc
        partition_init_functor;
    // solver
    auto solver = MultiSim::createMPMSolver<DEVICE, Settings::
   boundary_type,
                                            Settings::material_type,
                                            particle_members,
   particle_index>(
```

```cpp
        global_bounding_box, global_num_cell, Settings::halo_size,
        parpos_init_functor, scene_init_functor, partition_init_functor,
        Settings::PPC, Settings::res_x * Settings::res_y,
        Settings::num_step_rebalance, Settings::partition_op_on, Settings::cfl);
    // collision object
    MultiSim::VDB_Shape::VdbLevelSet<T, 3> vdb_ls(
        INPUT_DATA_PATH, "/collision_object.vdb", { 0.0, 0.0, 0.0 } );
    MultiSim::CollisionObject<MultiSim::CollisionTypes::STICKY, T, g_dim, DEVICE> collision_obj( global_num_cell, Settings::dx, vdb_ls );
    solver->solve( Settings::frame_num, Settings::fps, std::string( OUTPUT_DATA_PATH ) + "out_rank", Settings::init_y_vel, collision_obj, LOGGER_PATH );
}

int main( int argc, char* argv[] )
{
    using T = typename Examples::T;
    MPI_Init( &argc, &argv );
    Kokkos::initialize( argc, argv );

    test_example();

    Kokkos::finalize();
    MPI_Finalize();
    return 0;
}
```

APPENDIX C

TPA-Gen: Implementation and Other Details

C.1 Scene Grammar Productions

We use an attributed stochastic grammar as a hierarchical and structured representation that determines the scenario's content with initial physical parameters and appearance settings. The grammar is decomposed into multiple levels of components which are sampled according to the production rules defined in Table C.1. The tree structure itself describes the scenario's content, while the related `attributes`, which contain numerous `features`, specialize the content's characteristics. Table C.2 presents a list of the `attributes` and `features` designed for each *node*.

C.2 Dynamic Model and Constraints

Constraints.

In practice, we use the following eight constraints to reveal object relationships and the constraints are checked and applied on selected object `features`. Every constraint consists of a list of operands "$[o_0, o_1, ..., o_N]$" and an ID number list "$[n_0, .., n_M]$" which represents the unalterable criteria operand(s). Here, operand o_i refers to either a constant (value or vector that is always unalterable), or a *node*-`attribute`-`feature` pair, in which case the value of the corresponding feature is fetched for computation. Additionally, the criteria operands must by default satisfy the given constraints to avoid ambiguities.

- *less_eq*$([o_0, ..., o_N], [n_0, ..., n_M])$:
  $o_0 \leqslant ... \leqslant o_N$, with $o_{n_0}, ...o_{n_M}$ stay unchanged during resampling.

- *less*$([o_0, ..., o_N], [n_0, ..., n_M])$:

  $o_0 < ... < o_N$, with $o_{n_0}, ...o_{n_M}$ stay unchanged during resampling.

- *larger_eq*$([o_0, ..., o_N], [n_0, ..., n_M])$:

  $o_0 \geqslant ... \geqslant o_N$, with $o_{n_0}, ...o_{n_M}$ stay unchanged during resampling.

- *larger*$([o_0, ..., o_N], [n_0, ..., n_M])$:

  $o_0 > ... > o_N$, with $o_{n_0}, ...o_{n_M}$ stay unchanged during resampling.

- *eq*$([o_0, ..., o_N], [n_0, ..., n_M])$:

  $o_0 = ... = o_N$, with $o_{n_0}, ...o_{n_M}$ stays unchanged during resampling.

- *same_dir*$([o_0, ..., o_N], [n_0])$:

  o_i must be vectors, and for $\forall i \in [0, ..., N]$ the angle between o_i and o_{n_0} is zero. Note that only one criterion operand is allowed to be present in this constraint.

- *oppo_dir*$([o_0, ..., o_N], [n_0])$:

  o_i must be vectors, and for $\forall i \in [0, ..., N]$ the angle between o_i and o_{n_0} is $180°$. Note that only one criterion operand is allowed to be present in this constraint.

- *similar_dir*$([o_0, ..., o_N], [n_0, ..., n_M], \theta)$:

  o_i must be vectors, and for $\forall i \in [0, ..., N]$, $\forall j \in [0, ..., M]$ the angle between o_i and o_j is less or equal to θ. Here, $o_{n_0}, ...o_{n_M}$ stays unchanged during resampling.

During the sampling process, we validate the defined constraints after random sampling all `features`. The non-criteria operands that violate the constraints will be resampled to guarantee the correctness of the relation. If the criteria operands themselves violate the constraint, the resampling process will be terminated and errors will be reported.

Dynamic Model.

As introduced before, we have the following dynamic models: `JUMP`, `DROP`, `THROW`, `PUSH` and `STRIKE`. We summarize the basic constraints required for each intransitive dynamic model in Table C.3 and transitive dynamical verbs in Table C.4. In addition

Label	**Production Rules**
Scene	Scene → TarObjSet ⊕ Environment ⊕ Render
Component-0	TarObjSet → TargetObj$^+$ ⊙ TarObjSet*
	Environment → ColObjSet
Component-*	ColObjSet → CollisionObj* ⊙ ColObjSet*
	TarObjSet* → TargetObj* ⊙ TarObjSet*

Table C.1: **Production rule of the scenario stochastic grammar.** Here, TarObjSet is short for Target Object Set which includes a set of simulated object (TargetObj) with potential relationships; ColObjSet represents a set of non-movable collision objects (CollisionObj) serving as boundary conditions. Moreover, ⊕ represents *and* relation, making the child elements mandatory; while ⊙ refers to *or* relation to connect optional child nodes; $^+$ means one or more and * means zero or more.

to these dynamic models, one can easily define and generate other dynamic models into our codebase with predefined interfaces for constraints. Some example definitions can be found in Table C.5.

In all the above-mentioned tables, ”DM” refers to the Dynamic Model, and the ”sub” is used to denote the subjective object on which the verb in the dynamic model focuses. As shown in the last two columns of the table, objective objects (represented by ”obj”) are introduced with `from` and `to` directional relations.

C.3 Datasheet for Dataset

Motivation. We propose a method to generate Text and Physics-based Animation (TPA) for multi-modal model training. The goal is to expand the current problem domain of multi-modal learning from image-text understanding to vision-world dynamics understanding. We believe that this is one of the beginning steps to enable the multi-modal model's capability to understand our world from a model fundamental perspective.

Composition Details.

- **What do the instances that comprise the dataset represent?**

  Each generated instance consists of the following elements: a video of the rendered

Node	**Attribute**	**Feature**	**Label Candidates**
Env	Boundary Condition	Boundary	Box, Floor
		Type	Sticky, Slip
		Friction Factor	Smooth, Even Surface, Rough, Extremely rough
	External Force	Force type	Gravity, Wind
		Force value	*Dependent on* Force type
	Temporal	Total Frame	Short, Medium, Long
Object	Appearance	Color	White, Red, Blue, Green, Lime, Orange, Yellow, Pink, Purple ...
		Material	Glossy, Matte
	Shape	Shape	Cube, Sphere, Cylinder, Mesh
		Size	Small, Medium-sized, Large, Super large
	Motion	Velocity value	Slow, Medium-speed, Fast
		Velocity direction	Up, Down, Right, Left, Forward, Backward, Horizontal, Vertical
		Initial position	On the ground, In the sky
	Physics	Material	Elastic, Rigid, Fluid, Snow, Mud, Sand, Granular
		Young's Modulus	*Dependent on* Material*;* Soft, Moderate-hardness, Hard, Rigid
		Poisson Ratio	*Dependent on* Material*;* Elastic, Rigid
		density	Light, Medium-weight, Heavy
		Friction factor	Smooth, Even Surface, Rough, Extremely rough
Render	Background	Light	Bright, Dark
		Texture	Preset texture list or "random"
	Camera	Camera position	Preset camera position
		Viewpoint	*Dependent on* Camera position

Table C.2: **Attributes with features associated for each scene *node*.** In the table, independent features are highlighted with the small caps font style, whereas dependent features are labeled with *italic*. The final column lists examples of candidate labels for each feature. Each label is mapped to a specific value or range of values based on its semantics. Additional labels can be easily appended by providing a mapping between the label name and corresponding value ranges.

physics-based animation, a set of text descriptions of the animation, and a 3D material points (position) of the object ID at frame ID.

- **How many instances are there in total?**

As mentioned before, we propose a method to generate TPA data. In theory, one

DM	Type	Constraint
JUMP	Basic	$\textit{similar\_dir}([[0,1,0],(\text{sub, Motion, }\textsc{Velocity direction})],[0],\theta_0)$
		$\textit{less\_eq}([v_{min},(\text{sub, Motion, }\textsc{Velocity value})],[0])$ (v_{min} defined by user)
	`from`	$\textit{eq}([\mathbf{p}^{\text{gt}},(\text{sub, Motion, }\textsc{Initial position})],[0])$, with $\mathbf{p}^{\text{gt}}=[p_0^{\text{gt}},p_1^{\text{gt}},p_2^{\text{gt}}]$ Here, $p_i^{\text{gt}}=p_i^{\text{obj}}\pm(s_i^{\text{sub}}+s_i^{\text{obj}}+C)*0.5$ for $i\in[0,2]$
	`to`	$\textit{similar\_dir}([\mathbf{d}^{\text{gt}},(\text{sub, Motion, }\textsc{Velocity direction})],[0],\theta_1)$ Here, $\mathbf{d}^{\text{gt}}=(\mathbf{p}^{\text{obj}}-\mathbf{p}^{\text{sub}})+\alpha\cdot[0,1,0]$ (α defined by user)
DROP	Basic	$\textit{similar\_dir}([[0,-1,0],(\text{sub, Motion, }\textsc{Velocity direction})],[0],\theta_0)$
		$\textit{larger\_eq}([v_{\text{small}},(\text{sub, Motion, }\textsc{Velocity value})],[0])$ (v_{small} defined by user)
		$\textit{less\_eq}([\mathbf{p}^{\text{gt}},(\text{sub, Motion, }\textsc{Initial position})],[0])$, $\mathbf{p}^{\text{gt}}=[p_0^{\text{gt}},p_1^{\text{gt}},p_2^{\text{gt}}]$; p_1^{gt} is user-defined threshold; p_0^{gt} and p_2^{gt} are the global minimum position
	`from`	$\textit{eq}([\mathbf{p}^{\text{gt}},(\text{obj, Motion, }\textsc{Initial position})],[0])$ Here, $p_i^{\text{gt}}=p_i^{\text{sub}}\pm(s_i^{\text{sub}}+s_i^{\text{obj}}+C)*0.5$ for $i\in[0,2]$
	`to`	$\textit{eq}([\mathbf{p}^{\text{gt}},(\text{obj, Motion, }\textsc{Initial position})],[0])$ Here, $p_i^{\text{gt}}=p_i^{\text{sub}}\pm(s_i^{\text{sub}}+s_i^{\text{obj}}+C)*0.5$ for $i=0,2$; $p_1^{\text{gt}}=s_1^{\text{obj}}+C$
THROW	Basic (UP)	$\textit{similar\_dir}([\mathbf{v}_{dir}^{\text{gt}},(\text{sub, Motion, }\textsc{Velocity direction})],[0],\theta_0)$, $\mathbf{v}_{dir}^{\text{gt}}=[C_0,1,C_1]$
	Basic (DOWN)	$\textit{similar\_dir}([\mathbf{v}_{dir}^{\text{gt}},(\text{sub, Motion, }\textsc{Velocity direction})],[0],\theta_0)$, $\mathbf{v}_{dir}^{\text{gt}}=[C_0,-1,C_1]$
		$\textit{eq}([\mathbf{p}^{\text{gt}},(\text{obj, Motion, }\textsc{Initial position})],[0])$. Here, $p_i^{\text{gt}}=p_i^{\text{sub}}\pm(s_i^{\text{sub}}+s_i^{\text{obj}}+C)*0.5$
	`from`	$\textit{eq}([\mathbf{p}^{\text{gt}},(\text{obj, Motion, }\textsc{Initial position})],[0])$ Here, $p_i^{\text{gt}}=p_i^{\text{sub}}\pm(s_i^{\text{sub}}+s_i^{\text{obj}}+C)*0.5$ for $i\in[0,2]$
	`to`	$\textit{eq}([\mathbf{p}^{\text{gt}},(\text{obj, Motion, }\textsc{Initial position})],[0])$ Here, $\mathbf{p}^{\text{gt}}=\mathbf{p}^{\text{sub}}+\mathbf{s}^{\text{sub}}+v_{value}^{\text{sub}}\cdot\mathbf{v}_{dir}^{\text{sub}}\cdot C$

Table C.3: **Constraints in each single-object dynamic model.** In the table, $\mathbf{s}$, $\mathbf{p}$, v_{value} and $\mathbf{v}_{dir}$ refers to object size, initial position, velocity value and velocity direction, separately; $C \geqslant 0, C_0 \in [-1,1], C_1 \in [-1,1], C_{small} \in [0, 0.1 * s_{max}^{sub}]$ represents random noise, and ± means +/- are chosen randomly in practice. All the dynamic model has the basic constraints applied to the objects, with randomly sampled `from`, `to`, or NONE relation. Note that `from` and `to` relation will be chosen only when there are enough objects in the scenario. Specially, for `THROW` model, we will first sample to decide if it is ”Throw UP” or “Throw DOWN” before defining the basic constraints.

can generate as much as possible of TPA data with sufficient objects, types of materials, and pre-defined motion types. To better illustrate the details of the data, we provide a sample set of TPA data which contains 500 instances.

- **Are there recommended data splits?**

  We don’t specify any splits. One can configure the split accordingly.

DM	Type	Constraint
PUSH	Basic	$\textit{similar\_dir}([\mathbf{v}^{\text{gt}}_{dir}, (\text{sub, Motion, }\textsc{Velocity direction})], [0], \theta_0)$ Here $\mathbf{v}^{\text{gt}}_{dir} = \mathbf{p}^{obj} - \mathbf{p}^{\text{gt-sub}}$
		$\textit{less\_eq}([v_{\text{large}}, (\text{sub, Motion, }\textsc{Velocity value})], [0])$ (v_{large} defined by user)
		$\textit{larger\_eq}([v_{\text{small}}, (\text{obj, Motion, }\textsc{Velocity value})], [0])$ (v_{small} defined by user)
	Basic (Not `from`)	$eq([\mathbf{p}^{\text{gt-sub}}, (\text{sub, Motion, }\textsc{Initial position})], [0])$ Here, $p_i^{\text{gt-sub}} = p_i^{\text{obj}} \pm (s_i^{\text{sub}} + s_i^{\text{obj}} + C) * 0.5$ for $i \in [0, 2]$
	`from`	$eq([\mathbf{p}^{\text{gt-sub}}, (\text{sub, Motion, }\textsc{Initial position})], [0])$ Here, $p_i^{\text{gt-sub}} = p_i^{\text{obj-extra}} \pm (s_i^{\text{sub}} + s_i^{\text{obj-extra}} + C) * 0.5$ for $i \in [0, 2]$
		$eq([\mathbf{p}^{\text{gt-obj}}, (\text{obj, Motion, }\textsc{Initial position})], [0])$ Here, $\mathbf{p}^{\text{gt-obj}} = \mathbf{p}_i^{\text{gt-sub}} + K * \frac{\mathbf{p}^{\text{gt-sub}} - \mathbf{p}^{\text{obj-extra}}}{\Vert\mathbf{p}^{\text{gt-sub}} - \mathbf{p}^{\text{obj-extra}}\Vert}$ $K = 0.5 \cdot \max_{i=0,1,2}(s_i^{\text{sub}} + s_i^{\text{obj}}) + C$
	`to`	$eq([\mathbf{p}^{\text{gt-obj-extra}}, (\text{obj-extra, Motion, }\textsc{Initial position})], [0])$ Here, $\mathbf{p}^{\text{gt-obj-extra}} = \mathbf{p}^{\text{obj}} + K * \mathbf{v}^{\text{sub}}_{dir}$ $K = 0.5 \cdot \max_{i=0,1,2}(s_i^{\text{obj}} + s_i^{\text{obj-extra}}) + C$
STRIKE	`to`	$\mathbf{p}^{to} = \mathbf{p}^{\text{obj-extra}}$
	Not `to`	Random sample $\mathbf{p}^{to}$
	Basic	$\textit{similar\_dir}([\mathbf{p}^{\text{to}} - \mathbf{p}^{\text{sub}}, ($ $\text{sub, Motion, }\textsc{Velocity direction})], [0], \theta_0)$
		$\textit{similar\_dir}([\mathbf{p}^{\text{to}} - \mathbf{p}^{\text{obj}}, ($ $\text{obj, Motion, }\textsc{Velocity direction})], [0], \theta_0)$
		$\textit{less\_eq}([v_{\text{large}}, (\text{sub, Motion, }\textsc{Velocity value})], [0])$ (v_{large} defined by user)
		$\textit{less\_eq}([v_{\text{large}}, (\text{obj, Motion, }\textsc{Velocity value})], [0])$ (v_{large} defined by user)
	`from`	$eq([\mathbf{p}^{\text{gt}}, (\text{obj, Motion, }\textsc{Initial position})], [0])$ Here, $p_i^{\text{gt}} = p_i^{\text{sub}} \pm (s_i^{\text{sub}} + s_i^{\text{obj}} + C) * 0.5$ for $i \in [0, 2]$

Table C.4: **Constraints in each multiple-object dynamic model.** These dynamic models are applied to at least two objects, one representing the subjective and the other representing the objective ("sub" and "obj" in the table). `from` and `to` relation will include another objective object, named "obj-extra" in the table. $C \geqslant 0$ represents random noise, and ± means +/- are chosen randomly in practice.

Collection Process. As we mentioned before, we propose a method to generate text and physics-based animation data. Since the data is synthetic, we don't require any human annotators to be involved. Besides, to improve the quality of the generated texts, we take advantage of the most popular large language model, ChatGPT, to rewrite the generated description of the animation. Besides, we also use ChatGPT to help propose label names of the 3D objects in a TPA instance. The 3D objects are generated with a SoTA text-3D

DM	Type	Constraint
FLY	Basic	*similar_dir*($[\mathbf{v}^{\text{gt}}_{dir}$, (sub, Motion, Velocity direction)], [0], θ_0), $\mathbf{v}^{\text{gt}}_{dir} = [C_0, 0, C_1]$
		less_eq([v_{large}, (sub, Motion, Velocity value)], [0]) (v_{large} defined by user)
		less_eq([$\mathbf{p}^{\text{gt}}$, (sub, Motion, Initial position)], [0]), $\mathbf{p}^{\text{gt}} = [p^{\text{gt}}_0, p^{\text{gt}}_1, p^{\text{gt}}_2]$, p^{gt}_1 is user-defined threshold p^{gt}_0 and p^{gt}_2 are the global minimum position
	`from`	*eq*([$\mathbf{p}^{\text{gt}}$, (obj, Motion, Initial position)], [0]) Here, $p^{\text{gt}}_i = p^{\text{sub}}_i \pm (s^{\text{sub}}_i + s^{\text{obj}}_i + C) * 0.5$ for $i \in [0, 2]$
	`to`	*eq*([$\mathbf{p}^{\text{gt}}$, (obj, Motion, Initial position)], [0]) Here, $\mathbf{p}^{\text{gt}} = \mathbf{p}^{\text{sub}} + \mathbf{s}^{\text{sub}} + v^{\text{sub}}_{value} \cdot \mathbf{v}^{\text{sub}}_{dir} \cdot C$
SLIDE	Basic	*similar_dir*($[\mathbf{v}^{\text{gt}}_{dir}$, (sub, Motion, Velocity direction)], [0], θ_0), $\mathbf{v}^{\text{gt}}_{dir} = [C_0, 0, C_1]$
	Basic (Not `from`)	*less_eq*([$\mathbf{p}^{\text{gt-min}}$, (sub, Motion, Initial position), $\mathbf{p}^{\text{gt-max}}$], [0, 2]) Here $\mathbf{p}^{\text{gt-*}} = [p^{\text{gt-*}}_0, p^{\text{gt-*}}_1, p^{\text{gt-*}}_2]$, $p^{\text{gt-min}}_1 = s^{\text{sub}}_1$, $p^{\text{gt-max}}_1 = s^{\text{sub}}_1 + C_{\text{small}}$ $p^{\text{gt-*}}_i$ refers to global * position for $i = 0, 2$ and * refers to min/max
	`from`	*eq*([$\mathbf{p}^{\text{gt}}$, (sub, Motion, Initial position)], [0]), with $\mathbf{p}^{\text{gt}} = [p^{\text{gt}}_0, p^{\text{gt}}_1, p^{\text{gt}}_2]$ Here, $p^{\text{gt}}_1 = p^{\text{obj}}_1 + (s^{\text{sub}}_1 + s^{\text{obj}}_1 + C_{small}) * 0.5$ Random sample $p^{\text{gt}}_i \in [p^{\text{obj}}_i - s^{\text{obj}}_i * 0.5, p^{\text{obj}}_i + s^{\text{obj}}_i * 0.5]$ for $i = 0, 2$
	`to`	*eq*([$\mathbf{p}^{\text{gt}}$, (obj, Motion, Initial position)], [0]) Here, $\mathbf{p}^{\text{gt}} = \mathbf{p}^{\text{sub}} + \mathbf{s}^{\text{sub}} + v^{\text{sub}}_{value} \cdot \mathbf{v}^{\text{sub}}_{dir} \cdot C$

Table C.5: **Example of other possible dynamic models.** One can easily define and implement additional dynamic models within our codebase.

generation model. Specifically, we use the checkpoint of GPT-3.5-turbo-0301 as the backbone model.

Pre-processing/Cleaning/Labeling.

- **3D object representation transformation**:

  The 3D objects are spatially discretized to material points at each time step. The positions of the material points representing each object (including object-of-interests and collision object)d are stored in separate PLY files.

- **Description rewriting**:

  We use ChatGPT to help clean the generated text description by rewriting the sentences without changing the meaning of it.

Distribution.

- **How will the dataset be distributed?**
  The implementation of the proposed TPA data generation method contains three parts: 1, a scene sampling process code; 2, a binary execution of a physics-based simulation engine compiled from a set of source code. The source code of this simulation engine is based on an open-source version of Material Point Method (MPM); 3, data cleaning codes. The above implementations will be made public on GitHub.

- **When will the dataset be distributed?**
  The sample dataset and the implementation code will be distributed after the NeurIPS dataset track review process.

Impact and Challenges. We expect our proposed method and the data generated by this method can make a broad impact on both the multi-modal and computer graphics communities.

- In terms of multi-modal understanding, we aim to help this community expand the problem domain from shallow vision-language alignment to deep comprehension of the knowledge space of vision-language-world dynamics. It could broadly impact specific research domains such as Text-to-Video/Simulation (T2V/S), robotics, and intuitive physics.

- Our method could also make a large impact in the conventional computer graphics domain by reformulating the 3D animation creation process. Our generation process provides a rule-based generation of 3D animation scenes. On the other hand, once we have a reliable model that can generate 3D animation from human language, it could save a huge amount of effort to create 3D animation from scratch. The traditional pipeline usually requires very experienced artists, computer graphics researchers, and engineers to collaborate even for a simple scene. Our method initializes the very first step towards the fully automatic generation of 3D physics-based animations from the text description.

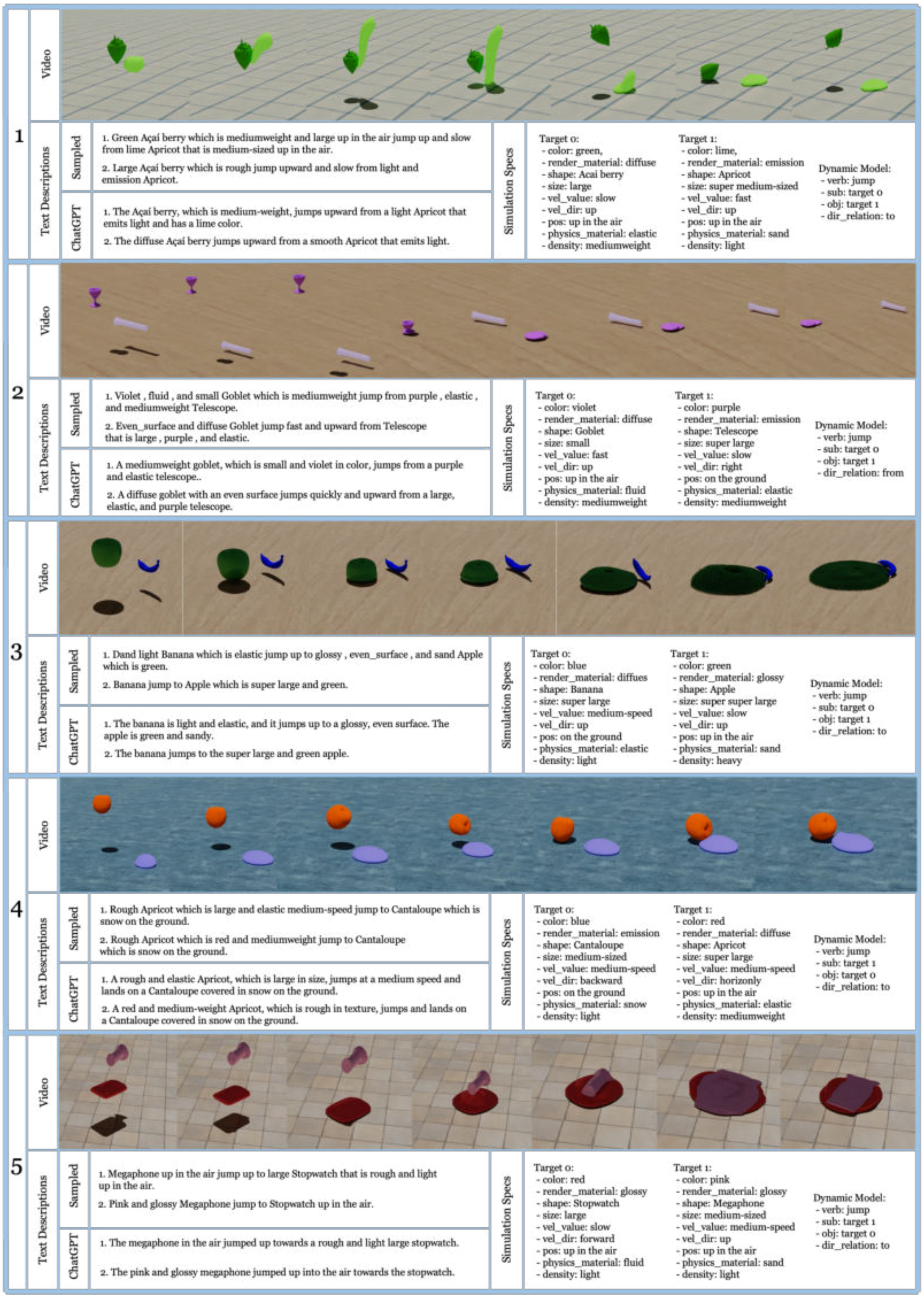
Licensing and Access. We would like to specify that we intend to utilize the **MIT license** for the method proposed in this section to generate TPA data. This open-source license grants users the freedom to use, modify, and distribute the dataset while providing clear attribution to the original creators. By choosing the MIT license, I aim to foster collaboration, encourage innovation, and ensure that the generated data and the code of the proposed method to generate remain accessible to the wider community for further exploration and development.

C.4 Data Samples

Figure C.1, Figure C.2, Figure C.3, and Figure C.4 demonstrate more sample examples of the proposed generative algorithm. For more demos, please check out this Google drive: https://drive.google.com/drive/folders/1IbPJBmPLlzB4DPmXVx1eQhSr42WYjLU_?usp=sharing. One can check out the demos in the following file hierarchy (DM refers to concrete dynamic model names):

- DM_scene:
  - label_out.json (labels of all nodes, attributes, and features)
  - value_out.json (quantitative values of corresponding setups)
  - sentence_original.json (sentence sampled from the proposed language model)
  - sentence_rewrite.json (sentence rewritten by ChatGPT)
  - render (a folder contains rendered results)
    - FID.png (FID refers to frame ID)
    - out.mp4 (video of rendering results)
- DM_3d: (3D object data)

- _0_0_0_target_OID_FID.ply (OID refers to object ID; 3D material points (position) of object OID at frame FID)
- _0_0_0_collision_FID.ply (point positions of collision objects)



1

Video

Text Descriptions

Sampled

1. Green Açaí berry which is mediumweight and large up in the air jump up and slow from lime Apricot that is medium-sized up in the air.

2. Large Açaí berry which is rough jump upward and slow from light and emission Apricot.

ChatGPT

1. The Açaí berry, which is medium-weight, jumps upward from a light Apricot that emits light and has a lime color.

2. The diffuse Açaí berry jumps upward from a smooth Apricot that emits light.

Simulation Specs

Target 0:
- color: green,
- render_material: diffuse
- shape: Acai berry
- size: large
- vel_value: slow
- vel_dir: up
- pos: up in the air
- physics_material: elastic
- density: mediumweight

Target 1:
- color: lime,
- render_material: emission
- shape: Apricot
- size: super medium-sized
- vel_value: fast
- vel_dir: up
- pos: up in the air
- physics_material: sand
- density: light

Dynamic Model:
- verb: jump
- sub: target 0
- obj: target 1
- dir_relation: to

2

Video

Text Descriptions

Sampled

1. Violet , fluid , and small Goblet which is mediumweight jump from purple , elastic , and mediumweight Telescope.

2. Even_surface and diffuse Goblet jump fast and upward from Telescope that is large , purple , and elastic.

ChatGPT

1. A mediumweight goblet, which is small and violet in color, jumps from a purple and elastic telescope..

2. A diffuse goblet with an even surface jumps quickly and upward from a large, elastic, and purple telescope.

Simulation Specs

Target 0:
- color: violet
- render_material: diffuse
- shape: Goblet
- size: small
- vel_value: fast
- vel_dir: up
- pos: up in the air
- physics_material: fluid
- density: mediumweight

Target 1:
- color: purple
- render_material: emission
- shape: Telescope
- size: super large
- vel_value: slow
- vel_dir: right
- pos: on the ground
- physics_material: elastic
- density: mediumweight

Dynamic Model:
- verb: jump
- sub: target 0
- obj: target 1
- dir_relation: from

3

Video

Text Descriptions

Sampled

1. Dand light Banana which is elastic jump up to glossy , even_surface , and sand Apple which is green.

2. Banana jump to Apple which is super large and green.

ChatGPT

1. The banana is light and elastic, and it jumps up to a glossy, even surface. The apple is green and sandy.

2. The banana jumps to the super large and green apple.

Simulation Specs

Target 0:
- color: blue
- render_material: diffues
- shape: Banana
- size: super large
- vel_value: medium-speed
- vel_dir: up
- pos: on the ground
- physics_material: elastic
- density: light

Target 1:
- color: green
- render_material: glossy
- shape: Apple
- size: super super large
- vel_value: slow
- vel_dir: up
- pos: up in the air
- physics_material: sand
- density: heavy

Dynamic Model:
- verb: jump
- sub: target 0
- obj: target 1
- dir_relation: to

4

Video

Text Descriptions

Sampled

1. Rough Apricot which is large and elastic medium-speed jump to Cantaloupe which is snow on the ground.

2. Rough Apricot which is red and mediumweight jump to Cantaloupe which is snow on the ground.

ChatGPT

1. A rough and elastic Apricot, which is large in size, jumps at a medium speed and lands on a Cantaloupe covered in snow on the ground.

2. A red and medium-weight Apricot, which is rough in texture, jumps and lands on a Cantaloupe covered in snow on the ground.

Simulation Specs

Target 0:
- color: blue
- render_material: emission
- shape: Cantaloupe
- size: medium-sized
- vel_value: medium-speed
- vel_dir: backward
- pos: on the ground
- physics_material: snow
- density: light

Target 1:
- color: red
- render_material: diffuse
- shape: Apricot
- size: super large
- vel_value: medium-speed
- vel_dir: horizonly
- pos: up in the air
- physics_material: elastic
- density: mediumweight

Dynamic Model:
- verb: jump
- sub: target 1
- obj: target 0
- dir_relation: to

5

Video

Text Descriptions

Sampled

1. Megaphone up in the air jump up to large Stopwatch that is rough and light up in the air.

2. Pink and glossy Megaphone jump to Stopwatch up in the air.

ChatGPT

1. The megaphone in the air jumped up towards a rough and light large stopwatch.

2. The pink and glossy megaphone jumped up into the air towards the stopwatch.

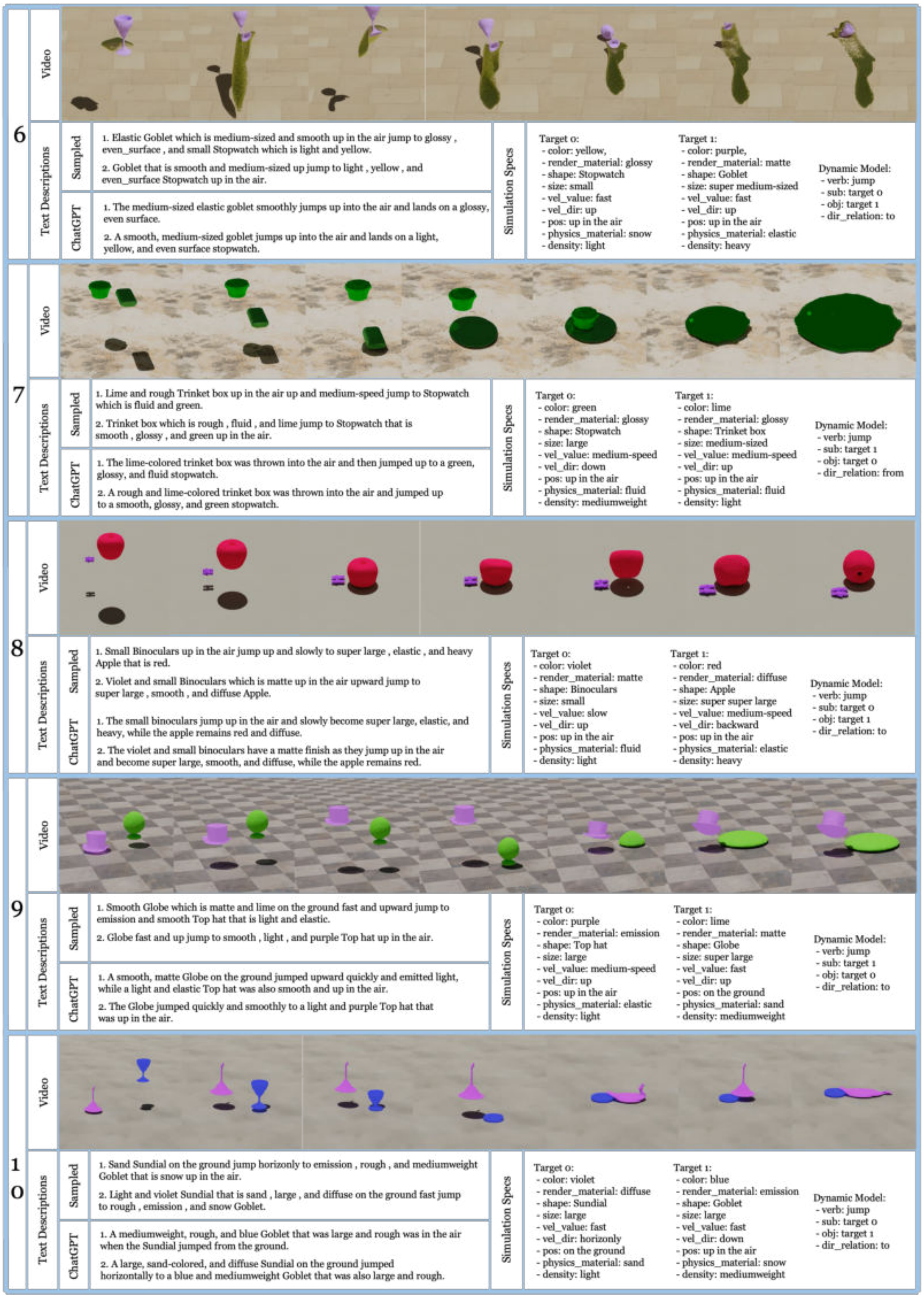
Simulation Specs

Target 0:
- color: red
- render_material: glossy
- shape: Stopwatch
- size: large
- vel_value: slow
- vel_dir: forward
- pos: up in the air
- physics_material: fluid
- density: light

Target 1:
- color: pink
- render_material: glossy
- shape: Megaphone
- size: medium-sized
- vel_value: medium-speed
- vel_dir: up
- pos: up in the air
- physics_material: sand
- density: light

Dynamic Model:
- verb: jump
- sub: target 1
- obj: target 0
- dir_relation: to

Figure C.1: Data samples of **DROP** dynamics.



1. Elastic Goblet which is medium-sized and smooth up in the air jump to glossy , even_surface , and small Stopwatch which is light and yellow.

2. Goblet that is smooth and medium-sized up jump to light , yellow , and even_surface Stopwatch up in the air.

1. The medium-sized elastic goblet smoothly jumps up into the air and lands on a glossy, even surface.

2. A smooth, medium-sized goblet jumps up into the air and lands on a light, yellow, and even surface stopwatch.

Target 0:
- color: yellow,
- render_material: glossy
- shape: Stopwatch
- size: small
- vel_value: fast
- vel_dir: up
- pos: up in the air
- physics_material: snow
- density: light

Target 1:
- color: purple,
- render_material: matte
- shape: Goblet
- size: super medium-sized
- vel_value: fast
- vel_dir: up
- pos: up in the air
- physics_material: elastic
- density: heavy

Dynamic Model:
- verb: jump
- sub: target 0
- obj: target 1
- dir_relation: to

1. Lime and rough Trinket box up in the air up and medium-speed jump to Stopwatch which is fluid and green.

2. Trinket box which is rough , fluid , and lime jump to Stopwatch that is smooth , glossy , and green up in the air.

1. The lime-colored trinket box was thrown into the air and then jumped up to a green, glossy, and fluid stopwatch.

2. A rough and lime-colored trinket box was thrown into the air and jumped up to a smooth, glossy, and green stopwatch.

Target 0:
- color: green
- render_material: glossy
- shape: Stopwatch
- size: large
- vel_value: medium-speed
- vel_dir: down
- pos: up in the air
- physics_material: fluid
- density: mediumweight

Target 1:
- color: lime
- render_material: glossy
- shape: Trinket box
- size: medium-sized
- vel_value: medium-speed
- vel_dir: up
- pos: up in the air
- physics_material: fluid
- density: light

Dynamic Model:
- verb: jump
- sub: target 1
- obj: target 0
- dir_relation: from

1. Small Binoculars up in the air jump up and slowly to super large , elastic , and heavy Apple that is red.

2. Violet and small Binoculars which is matte up in the air upward jump to super large , smooth , and diffuse Apple.

1. The small binoculars jump up in the air and slowly become super large, elastic, and heavy, while the apple remains red and diffuse.

2. The violet and small binoculars have a matte finish as they jump up in the air and become super large, smooth, and diffuse, while the apple remains red.

Target 0:
- color: violet
- render_material: matte
- shape: Binoculars
- size: small
- vel_value: slow
- vel_dir: up
- pos: up in the air
- physics_material: fluid
- density: light

Target 1:
- color: red
- render_material: diffuse
- shape: Apple
- size: super super large
- vel_value: medium-speed
- vel_dir: backward
- pos: up in the air
- physics_material: elastic
- density: heavy

Dynamic Model:
- verb: jump
- sub: target 0
- obj: target 1
- dir_relation: to

1. Smooth Globe which is matte and lime on the ground fast and upward jump to emission and smooth Top hat that is light and elastic.

2. Globe fast and up jump to smooth , light , and purple Top hat up in the air.

1. A smooth, matte Globe on the ground jumped upward quickly and emitted light, while a light and elastic Top hat was also smooth and up in the air.

2. The Globe jumped quickly and smoothly to a light and purple Top hat that was up in the air.

Target 0:
- color: purple
- render_material: emission
- shape: Top hat
- size: large
- vel_value: medium-speed
- vel_dir: up
- pos: up in the air
- physics_material: elastic
- density: light

Target 1:
- color: lime
- render_material: matte
- shape: Globe
- size: super large
- vel_value: fast
- vel_dir: up
- pos: on the ground
- physics_material: sand
- density: mediumweight

Dynamic Model:
- verb: jump
- sub: target 1
- obj: target 0
- dir_relation: to

1. Sand Sundial on the ground jump horizonly to emission , rough , and mediumweight Goblet that is snow up in the air.

2. Light and violet Sundial that is sand , large , and diffuse on the ground fast jump to rough , emission , and snow Goblet.

1. A mediumweight, rough, and blue Goblet that was large and rough was in the air when the Sundial jumped from the ground.

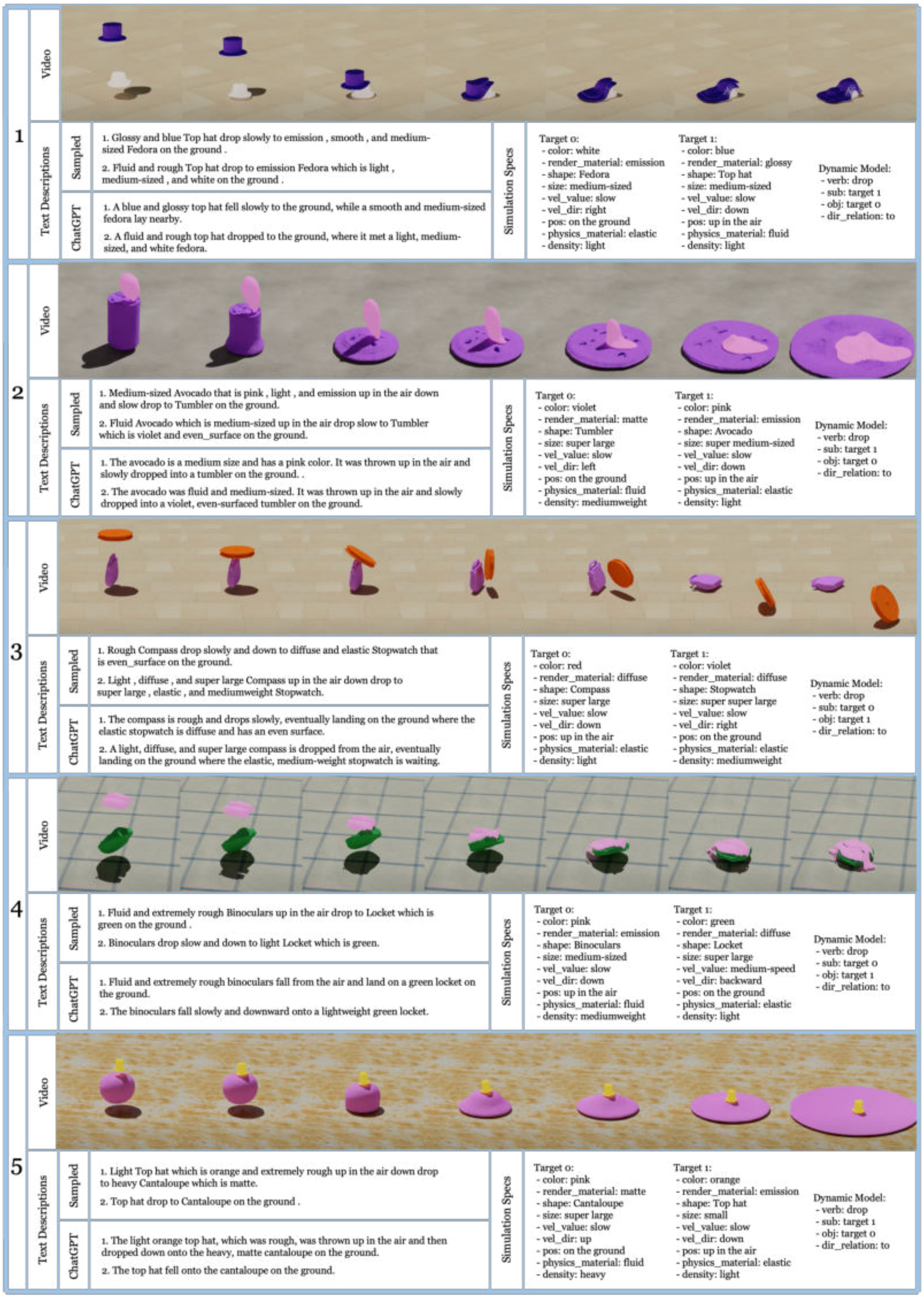
2. A large, sand-colored, and diffuse Sundial on the ground jumped horizontally to a blue and mediumweight Goblet that was also large and rough.

Target 0:
- color: violet
- render_material: diffuse
- shape: Sundial
- size: large
- vel_value: fast
- vel_dir: horizonly
- pos: on the ground
- physics_material: sand
- density: light

Target 1:
- color: blue
- render_material: emission
- shape: Goblet
- size: large
- vel_value: fast
- vel_dir: down
- pos: up in the air
- physics_material: snow
- density: mediumweight

Dynamic Model:
- verb: jump
- sub: target 0
- obj: target 1
- dir_relation: to

Figure C.2: Data samples of **DROP** dynamics.



1. Glossy and blue Top hat drop slowly to emission , smooth , and medium-sized Fedora on the ground .

2. Fluid and rough Top hat drop to emission Fedora which is light , medium-sized , and white on the ground .

1. A blue and glossy top hat fell slowly to the ground, while a smooth and medium-sized fedora lay nearby.

2. A fluid and rough top hat dropped to the ground, where it met a light, medium-sized, and white fedora.

Target 0:
- color: white
- render_material: emission
- shape: Fedora
- size: medium-sized
- vel_value: slow
- vel_dir: right
- pos: on the ground
- physics_material: elastic
- density: light

Target 1:
- color: blue
- render_material: glossy
- shape: Top hat
- size: medium-sized
- vel_value: slow
- vel_dir: down
- pos: up in the air
- physics_material: fluid
- density: light

Dynamic Model:
- verb: drop
- sub: target 1
- obj: target 0
- dir_relation: to

1. Medium-sized Avocado that is pink , light , and emission up in the air down and slow drop to Tumbler on the ground.

2. Fluid Avocado which is medium-sized up in the air drop slow to Tumbler which is violet and even_surface on the ground.

1. The avocado is a medium size and has a pink color. It was thrown up in the air and slowly dropped into a tumbler on the ground. .

2. The avocado was fluid and medium-sized. It was thrown up in the air and slowly dropped into a violet, even-surfaced tumbler on the ground.

Target 0:
- color: violet
- render_material: matte
- shape: Tumbler
- size: super large
- vel_value: slow
- vel_dir: left
- pos: on the ground
- physics_material: fluid
- density: mediumweight

Target 1:
- color: pink
- render_material: emission
- shape: Avocado
- size: super medium-sized
- vel_value: slow
- vel_dir: down
- pos: up in the air
- physics_material: elastic
- density: light

Dynamic Model:
- verb: drop
- sub: target 1
- obj: target 0
- dir_relation: to

1. Rough Compass drop slowly and down to diffuse and elastic Stopwatch that is even_surface on the ground.

2. Light , diffuse , and super large Compass up in the air down drop to super large , elastic , and mediumweight Stopwatch.

1. The compass is rough and drops slowly, eventually landing on the ground where the elastic stopwatch is diffuse and has an even surface.

2. A light, diffuse, and super large compass is dropped from the air, eventually landing on the ground where the elastic, medium-weight stopwatch is waiting.

Target 0:
- color: red
- render_material: diffuse
- shape: Compass
- size: super large
- vel_value: slow
- vel_dir: down
- pos: up in the air
- physics_material: elastic
- density: light

Target 1:
- color: violet
- render_material: diffuse
- shape: Stopwatch
- size: super super large
- vel_value: slow
- vel_dir: right
- pos: on the ground
- physics_material: elastic
- density: mediumweight

Dynamic Model:
- verb: drop
- sub: target 0
- obj: target 1
- dir_relation: to

1. Fluid and extremely rough Binoculars up in the air drop to Locket which is green on the ground .

2. Binoculars drop slow and down to light Locket which is green.

1. Fluid and extremely rough binoculars fall from the air and land on a green locket on the ground.

2. The binoculars fall slowly and downward onto a lightweight green locket.

Target 0:
- color: pink
- render_material: emission
- shape: Binoculars
- size: medium-sized
- vel_value: slow
- vel_dir: down
- pos: up in the air
- physics_material: fluid
- density: mediumweight

Target 1:
- color: green
- render_material: diffuse
- shape: Locket
- size: super large
- vel_value: medium-speed
- vel_dir: backward
- pos: on the ground
- physics_material: elastic
- density: light

Dynamic Model:
- verb: drop
- sub: target 0
- obj: target 1
- dir_relation: to

1. Light Top hat which is orange and extremely rough up in the air down drop to heavy Cantaloupe which is matte.

2. Top hat drop to Cantaloupe on the ground .

1. The light orange top hat, which was rough, was thrown up in the air and then dropped down onto the heavy, matte cantaloupe on the ground.

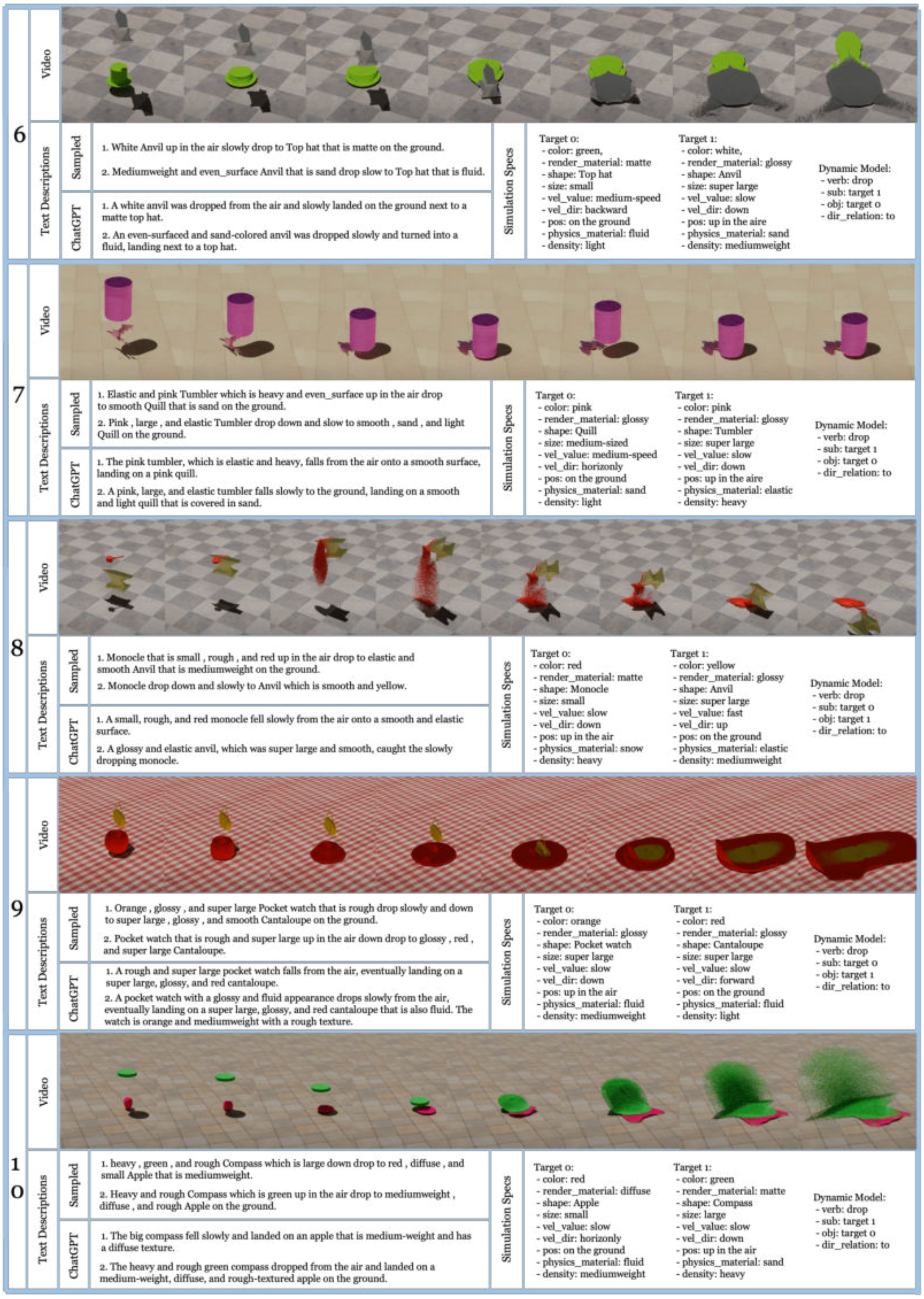
2. The top hat fell onto the cantaloupe on the ground.

Target 0:
- color: pink
- render_material: matte
- shape: Cantaloupe
- size: super large
- vel_value: slow
- vel_dir: up
- pos: on the ground
- physics_material: fluid
- density: heavy

Target 1:
- color: orange
- render_material: emission
- shape: Top hat
- size: small
- vel_value: slow
- vel_dir: down
- pos: up in the air
- physics_material: elastic
- density: light

Dynamic Model:
- verb: drop
- sub: target 1
- obj: target 0
- dir_relation: to

Figure C.3: Data samples of **JUMP** dynamics.



6

Video

Text Descriptions

Sampled

1. White Anvil up in the air slowly drop to Top hat that is matte on the ground.

2. Mediumweight and even_surface Anvil that is sand drop slow to Top hat that is fluid.

ChatGPT

1. A white anvil was dropped from the air and slowly landed on the ground next to a matte top hat.

2. An even-surfaced and sand-colored anvil was dropped slowly and turned into a fluid, landing next to a top hat.

Simulation Specs

Target 0:
- color: green,
- render_material: matte
- shape: Top hat
- size: small
- vel_value: medium-speed
- vel_dir: backward
- pos: on the ground
- physics_material: fluid
- density: light

Target 1:
- color: white,
- render_material: glossy
- shape: Anvil
- size: super large
- vel_value: slow
- vel_dir: down
- pos: up in the aire
- physics_material: sand
- density: mediumweight

Dynamic Model:
- verb: drop
- sub: target 1
- obj: target 0
- dir_relation: to

7

Video

Text Descriptions

Sampled

1. Elastic and pink Tumbler which is heavy and even_surface up in the air drop to smooth Quill that is sand on the ground.

2. Pink , large , and elastic Tumbler drop down and slow to smooth , sand , and light Quill on the ground.

ChatGPT

1. The pink tumbler, which is elastic and heavy, falls from the air onto a smooth surface, landing on a pink quill.

2. A pink, large, and elastic tumbler falls slowly to the ground, landing on a smooth and light quill that is covered in sand.

Simulation Specs

Target 0:
- color: pink
- render_material: glossy
- shape: Quill
- size: medium-sized
- vel_value: medium-speed
- vel_dir: horizonly
- pos: on the ground
- physics_material: sand
- density: light

Target 1:
- color: pink
- render_material: glossy
- shape: Tumbler
- size: super large
- vel_value: slow
- vel_dir: down
- pos: up in the aire
- physics_material: elastic
- density: heavy

Dynamic Model:
- verb: drop
- sub: target 1
- obj: target 0
- dir_relation: to

8

Video

Text Descriptions

Sampled

1. Monocle that is small , rough , and red up in the air drop to elastic and smooth Anvil that is mediumweight on the ground.

2. Monocle drop down and slowly to Anvil which is smooth and yellow.

ChatGPT

1. A small, rough, and red monocle fell slowly from the air onto a smooth and elastic surface.

2. A glossy and elastic anvil, which was super large and smooth, caught the slowly dropping monocle.

Simulation Specs

Target 0:
- color: red
- render_material: matte
- shape: Monocle
- size: small
- vel_value: slow
- vel_dir: down
- pos: up in the air
- physics_material: snow
- density: heavy

Target 1:
- color: yellow
- render_material: glossy
- shape: Anvil
- size: super large
- vel_value: fast
- vel_dir: up
- pos: on the ground
- physics_material: elastic
- density: mediumweight

Dynamic Model:
- verb: drop
- sub: target 0
- obj: target 1
- dir_relation: to

9

Video

Text Descriptions

Sampled

1. Orange , glossy , and super large Pocket watch that is rough drop slowly and down to super large , glossy , and smooth Cantaloupe on the ground.

2. Pocket watch that is rough and super large up in the air down drop to glossy , red , and super large Cantaloupe.

ChatGPT

1. A rough and super large pocket watch falls from the air, eventually landing on a super large, glossy, and red cantaloupe.

2. A pocket watch with a glossy and fluid appearance drops slowly from the air, eventually landing on a super large, glossy, and red cantaloupe that is also fluid. The watch is orange and mediumweight with a rough texture.

Simulation Specs

Target 0:
- color: orange
- render_material: glossy
- shape: Pocket watch
- size: super large
- vel_value: slow
- vel_dir: down
- pos: up in the air
- physics_material: fluid
- density: mediumweight

Target 1:
- color: red
- render_material: glossy
- shape: Cantaloupe
- size: super large
- vel_value: slow
- vel_dir: forward
- pos: on the ground
- physics_material: fluid
- density: light

Dynamic Model:
- verb: drop
- sub: target 0
- obj: target 1
- dir_relation: to

1
0

Video

Text Descriptions

Sampled

1. heavy , green , and rough Compass which is large down drop to red , diffuse , and small Apple that is mediumweight.

2. Heavy and rough Compass which is green up in the air drop to mediumweight , diffuse , and rough Apple on the ground.

ChatGPT

1. The big compass fell slowly and landed on an apple that is medium-weight and has a diffuse texture.

2. The heavy and rough green compass dropped from the air and landed on a medium-weight, diffuse, and rough-textured apple on the ground.

Simulation Specs

Target 0:
- color: red
- render_material: diffuse
- shape: Apple
- size: small
- vel_value: slow
- vel_dir: horizonly
- pos: on the ground
- physics_material: fluid
- density: mediumweight

Target 1:
- color: green
- render_material: matte
- shape: Compass
- size: large
- vel_value: slow
- vel_dir: down
- pos: up in the air
- physics_material: sand
- density: heavy

Dynamic Model:
- verb: drop
- sub: target 1
- obj: target 0
- dir_relation: to

Figure C.4: Data samples of **JUMP** dynamics.

A., M. (2014). Cppcon 2014: Mike acton “data-oriented design and c++”. https://www.youtube.com/watch?v=rX0ItVEVjHc. 168

Aanjaneya, M., Gao, M., Liu, H., Batty, C., and Sifakis, E. (2017). Power diagrams and sparse paged grids for high resolution adaptive liquids. *ACM TOG*, 36(4):140. 18

Adinets, A. (2014). Cuda pro tip: Optimized filtering with warp-aggregated atomics. https://devblogs.nvidia.com/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/. 34

Alayrac, J.-B., Donahue, J., Luc, P., Miech, A., Barr, I., Hasson, Y., Lenc, K., Mensch, A., Millican, K., Reynolds, M., et al. (2022). Flamingo: a visual language model for few-shot learning. *arXiv preprint arXiv:2204.14198*. 127

Amada, T., Imura, M., Yasumuro, Y., Manabe, Y., and Chihara, K. (2004). Particle-based fluid simulation on gpu. In *ACM workshop on general-purpose computing on graphics processors*, volume 41, page 42. 17, 60

Ament, M., Knittel, G., Weiskopf, D., and Strasser, W. (2010). A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-gpu platform. In *Parallel, Distributed and Network-based Processing*, pages 583–592. IEEE. 17

Argall, B. D., Chernova, S., Veloso, M., and Browning, B. (2009). A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483. 103

Bailey, D., Masters, I., Warner, M., and Biddle, H. (2013). Simulating fluids using a coupled voxel-particle data model. In *SIGGRAPH 2013 Talks*, page 15. ACM. 19, 31

Bain, M., Nagrani, A., Varol, G., and Zisserman, A. (2021). Frozen in time: A joint video and image encoder for end-to-end retrieval. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1728–1738. 131

Barron, J. T., Mildenhall, B., Verbin, D., Srinivasan, P. P., and Hedman, P. (2022). Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5470–5479. 146

Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. (2012). Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE. 60

Berger, M. J. and Bokhari, S. H. (1987). A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, 36(05):570–580. 61

Bernstein, G. L., Shah, C., Lemire, C., Devito, Z., Fisher, M., Levis, P., and Hanrahan, P. (2016). Ebb: A dsl for physical simulation on cpus and gpus. *ACM TOG*, 35(2):21. 17

Blender, O. (2018). Blender—a 3d modelling and rendering package. *Retrieved. represents the sequence of Constructs1 to*, 4. 137

Bojsen-Hansen, M., Nielsen, M. B., Stamatelos, K., and Bridson, R. (2021). Spatially adaptive volume tools in bifrost. In *ACM SIGGRAPH 2021 Talks*, SIGGRAPH '21, New York, NY, USA. Association for Computing Machinery. 61

Bonet, J. and Wood, R. D. (1997). *Nonlinear continuum mechanics for finite element analysis*. Cambridge university press. 149, 150

Brackbill, J. U. and Ruppel, H. M. (1986). Flip: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. *Journal of Computational Physics*, 65(2):314–343. 19

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. 103

Brown, N. and Sandholm, T. (2018). Superhuman ai for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359(6374):418–424. 103

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901. 127

Caba Heilbron, F., Escorcia, V., Ghanem, B., and Carlos Niebles, J. (2015). Activitynet: A large-scale video benchmark for human activity understanding. In *Proceedings of the ieee conference on computer vision and pattern recognition*, pages 961–970. 131

Calli, B., Walsman, A., Singh, A., Srinivasa, S., Abbeel, P., and Dollar, A. M. (2015). Benchmarking in manipulation research. *IEEE Robotics & Automation Magazine*, 1070(9932/15):36. 107

Catalyurek, U. V., Boman, E. G., Devine, K. D., Bozdag, D., Heaphy, R., and Riesen, L. A. (2007). Hypergraph-based dynamic load balancing for adaptive scientific computations. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–11. IEEE. 61

Chandrasekhar, S. (1967). Ellipsoidal figures of equilibrium—an historical account. *Communications on Pure and Applied Mathematics*, 20(2):251–265. 153

Chang, A., Dai, A., Funkhouser, T., Halber, M., Niebner, M., Savva, M., Song, S., Zeng, A., and Zhang, Y. (2017). Matterport3d: Learning from rgb-d data in indoor environments. In *International Conference on 3D Vision (3DV)*. 107

Chang, A. X., Funkhouser, T., Guibas, L., Hanrahan, P., Huang, Q., Li, Z., Savarese, S., Savva, M., Song, S., Su, H., et al. (2015). Shapenet: An information-rich 3d model repository. 107

Chang, H., Zhang, H., Barber, J., Maschinot, A., Lezama, J., Jiang, L., Yang, M.-H., Murphy, K., Freeman, W. T., Rubinstein, M., et al. (2023). Muse: Text-to-image generation via masked generative transformers. *arXiv preprint arXiv:2301.00704*. 127

Chang, H., Zhang, H., Jiang, L., Liu, C., and Freeman, W. T. (2022). Maskgit: Masked generative image transformer. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11315–11325. 127

Changpinyo, S., Sharma, P., Ding, N., and Soricut, R. (2021). Conceptual 12m: Pushing web-scale image-text pre-training to recognize long-tail visual concepts. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3558–3568. 127, 130

Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C., and Smith, L. (2010). Introducing openshmem: Shmem for the pgas community. In *Proceedings of PGAS Programming Model*, pages 1–3. 40

Chen, D. and Dolan, W. B. (2011). Collecting highly parallel data for paraphrase evaluation. In *Proceedings of the 49th annual meeting of the association for computational linguistics: human language technologies*, pages 190–200. 131

Chen, H.-y., Tretschk, E., Stuyck, T., Kadlecek, P., Kavan, L., Vouga, E., and Lassner, C. (2022a). Virtual elastic objects. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15827–15837. 147

Chen, J.-K., Lyu, J., and Wang, Y.-X. (2023). Neuraleditor: Editing neural radiance fields via manipulating point clouds. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12439–12448. 147, 152

Chen, Y.-C., Li, L., Yu, L., El Kholy, A., Ahmed, F., Gan, Z., Cheng, Y., and Liu, J. (2020). Uniter: Universal image-text representation learning. In *European conference on computer vision*, pages 104–120. Springer. 127

Chen, Z., Wang, G., and Liu, Z. (2022b). Text2light: Zero-shot text-driven hdr panorama generation. *ACM Transactions on Graphics (TOG)*, 41(6):1–16. 130

Chentanez, N. and Müller, M. (2011). Real-time eulerian water simulation using a restricted tall cell grid. *ACM TOG*, 30(4):82. 17, 59

Chentanez, N. and Müller, M. (2013). Mass-conserving eulerian liquid simulation. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 20(1):17–29. 17, 59

Chentanez, N., Müller, M., and Kim, T. (2015). Coupling 3d eulerian, heightfield and particle methods for interactive simulation of large scale liquid phenomena. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 21(10):1116–1128. 17, 60

Cohen, J. M., Tariq, S., and Green, S. (2010). Interactive fluid-particle simulation using translating eulerian grids. In *2010 Symp Interactive 3D Graphics and Games*, pages 15–22. ACM. 17, 59

Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55. 17, 59

Daviet, G. and Bertails-Descoubes, F. (2016). A semi-implicit material point method for the continuum simulation of granular materials. *ACM TOG*, 35(4):102. 2, 19

Derouillat, J., Beck, A., Pérez, F., Vinci, T., Chiaramello, M., Grassi, A., Flé, M., Bouchard, G., Plotnikov, I., Aunai, N., et al. (2018). Smilei: a collaborative, open-source, multi-purpose particle-in-cell code for plasma simulation. *Computer Physics Communications*, 222:351–373. 22

developers, O. (2021). Openssh. 82

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*. 127

Ding, M., Han, X., Wang, S., Gast, T. F., and Teran, J. M. (2019). A thermomechanical material point method for baking and cooking. *ACM TOG*, 38(6):192. 19

Ding, O. and Craig, S. (2019). Penalty force for coupling materials with coulomb friction. *IEEE Transactions on Visualization and Computer Graph (TVCG)*. 20

Domínguez, J. M., Crespo, A. J., Valdez-Balderas, D., Rogers, B. D., and Gómez-Gesteira, M. (2013). New multi-gpu implementation for smoothed particle hydrodynamics on heterogeneous clusters. *Computer Physics Communications*, 184(8):1848–1860. 17

Dritschel, D. G., Reinaud, J. N., and McKIVER, W. J. (2004). The quasi-geostrophic ellipsoidal vortex model. *Journal of Fluid Mechanics*, 505:201–223. 164

Duan, Y., Chen, X., Houthooft, R., Schulman, J., and Abbeel, P. (2016). Benchmarking deep reinforcement learning for continuous control. In *ICML*. 102

Edmonds, M., Ma, X., Qi, S., Zhu, Y., Lu, H., and Zhu, S.-C. (2020). Theory-based causal transfer: Integrating instance-level induction and abstract-level structure learning. In *AAAI Conference on Artificial Intelligence (AAAI)*. 51

Edwards, H. C., Trott, C. R., and Sunderland, D. (2014). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing*, 74(12):3202–3216. 21, 56, 60, 64, 66

Facility, O. R. L. C. (2018). Summit: Oak ridge national laboratory's 200 petaflop supercomputer. `https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/`. 22

Fang, Y., Hu, Y., Hu, S.-M., and Jiang, C. (2018). A Temporally Adaptive Material Point Method with Regional Time Stepping. *Computer Graphics Forum*. 20, 53

Fang, Y., Li, M., Gao, M., and Jiang, C. (2019). Silly rubber: an implicit material point method for simulating non-equilibrated viscoelastic and elastoplastic solids. *ACM TOG*, 38(4):1–13. 13, 15, 19, 43

Fei, Y., Huang, Y., and Gao, M. (2021). Principles towards real-time simulation of material point method on modern gpus. *arXiv preprint arXiv:2111.00699*. 53, 54, 58, 60, 62

Fei, Y. R., Batty, C., Grinspun, E., and Zheng, C. (2018). A multi-scale model for simulating liquid-fabric interactions. *ACM TOG*, 37(4):51. 2, 19

Fridovich-Keil, S., Yu, A., Tancik, M., Chen, Q., Recht, B., and Kanazawa, A. (2022). Plenoxels: Radiance fields without neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5501–5510. 146

Gao, F., Ping, Q., Thattai, G., Reganti, A., Wu, Y. N., and Natarajan, P. (2022). Transform-retrieve-generate: Natural language-centric outside-knowledge visual question answering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5067–5077. 127

Gao, M., Pradhana, A., Han, X., Guo, Q., Kot, G., Sifakis, E., and Jiang, C. (2018a). Animating fluid sediment mixture in particle-laden flows. *ACM TOG*, 37(4):149. 20

Gao, M., Tampubolon, A. P., Jiang, C., and Sifakis, E. (2017). An adaptive generalized interpolation material point method for simulating elastoplastic materials. *ACM TOG*, 36(6):223. 20

Gao, M., Wang, X., Wu, K., Pradhana, A., Sifakis, E., Yuksel, C., and Jiang, C. (2018b). Gpu optimization of material point methods. *ACM Transactions on Graphics (TOG)*, 37(6):1–12. 2, 15, 17, 18, 19, 20, 21, 23, 24, 26, 27, 28, 30, 33, 34, 41, 53, 58, 60, 61, 62, 69, 70, 83, 94, 148

Gaume, J., Gast, T., Teran, J., van Herwijnen, A., and Jiang, C. (2018). Dynamic anticrack propagation in snow. *Nature Communications*, 9(1):3047. 2, 19

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2020). Generative adversarial networks. *Communications of the ACM*, 63(11):139–144. 127, 129

Goswami, P., Schlegel, P., Solenthaler, B., and Pajarola, R. (2010). Interactive sph simulation and rendering on the gpu. In *Symposium on Computer Animation (SCA)*, pages 55–64. Eurographics Association. 17, 59

Guo, Q., Han, X., Fu, C., Gast, T., Tamstorf, R., and Teran, J. (2018). A material point method for thin shells with frictional contact. *ACM TOG*, 37(4):147. 2, 19

Han, X., Gast, T. F., Guo, Q., Wang, S., Jiang, C., and Teran, J. (2019). A hybrid material point method for frictional contact with diverse materials. *ACM TOG*, 2(2):1–24. 20

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *CVPR*. 103

Hendricks, L. A., Wang, O., Shechtman, E., Sivic, J., Darrell, T., and Russell, B. (2018). Localizing moments in video with temporal language. In *Empirical Methods in Natural Language Processing (EMNLP)*. 131

Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507. 102

Ho, J., Chan, W., Saharia, C., Whang, J., Gao, R., Gritsenko, A., Kingma, D. P., Poole, B., Norouzi, M., Fleet, D. J., et al. (2022). Imagen video: High definition video generation with diffusion models. *arXiv preprint arXiv:2210.02303*. 127, 130

Ho, J., Jain, A., and Abbeel, P. (2020). Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, 33:6840–6851. 127, 130

Hoetzlein, R. (2016). Gvdb: raytracing sparse voxel database structures on the gpu. In *Proceedings of HPG*, pages 109–117. Eurographics Association. 18, 61

Hong, F., Zhang, M., Pan, L., Cai, Z., Yang, L., and Liu, Z. (2022a). Avatarclip: Zero-shot text-driven generation and animation of 3d avatars. *ACM Transactions on Graphics (TOG)*, 41(4):1–19. 130

Hong, W., Ding, M., Zheng, W., Liu, X., and Tang, J. (2022b). Cogvideo: Large-scale pretraining for text-to-video generation via transformers. *arXiv preprint arXiv:2205.15868*. 130

Hu, Y., Fang, Y., Ge, Z., Qu, Z., Zhu, Y., Pradhana, A., and Jiang, C. (2018). A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM TOG*, 37(4):150. 17, 18, 20, 23, 148

Hu, Y., Li, T.-M., Anderson, L., Ragan-Kelley, J., and Durand, F. (2019a). Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)*, 38(6):1–16. 13, 15, 17, 18, 19, 20, 21, 24, 26, 27, 28, 30, 31, 33, 40, 43, 44, 45, 53, 60, 61, 62, 148, 168, 171

Hu, Y., Liu, J., Spielberg, A., Tenenbaum, J. B., Freeman, W. T., Wu, J., Rus, D., and Matusik, W. (2019b). Chainqueen: A real-time differentiable physical simulator for soft robotics. In *International Conference on Robotics and Automation (ICRA)*. 51

Hu, Y., Liu, J., Yang, X., Xu, M., Kuang, Y., Xu, W., Dai, Q., Freeman, W. T., and Durand, F. (2021). Quantaichi: a compiler for quantized simulations. *ACM Transactions on Graphics (TOG)*, 40(4):1–16. 53, 60, 62

Jain, A., Mildenhall, B., Barron, J. T., Abbeel, P., and Poole, B. (2022). Zero-shot text-guided object generation with dream fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 867–876. 127, 130

Jambon, C., Kerbl, B., Kopanas, G., Diolatzis, S., Leimkühler, T., and Drettakis, G. (2023). Nerfshop: Interactive editing of neural radiance fields". *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 6(1). 144

Jiang, C., Gast, T., and Teran, J. (2017). Anisotropic elastoplasticity for cloth, knit and hair frictional contact. *ACM TOG*, 36(4):152. 2, 19, 41, 148

Jiang, C., Qi, S., Zhu, Y., Huang, S., Lin, J., Yu, L.-F., Terzopoulos, D., and Zhu, S.-C. (2018). Configurable 3d scene synthesis and 2d image rendering with per-pixel ground truth using stochastic grammars. *IJCV*, 126(9):920–941. 107

Jiang, C., Schroeder, C., Selle, A., Teran, J., and Stomakhin, A. (2015). The affine particle-in-cell method. *ACM Transactions on Graphics (TOG)*, 34(4):1–10. 9, 11, 12, 148

Jiang, C., Schroeder, C., Teran, J., Stomakhin, A., and Selle, A. (2016). The material point method for simulating continuum materials. In *ACM SIGGRAPH 2016 Courses*, SIGGRAPH '16, New York, NY, USA. Association for Computing Machinery. 6, 62, 150

Johnson, J., Hariharan, B., Van Der Maaten, L., Fei-Fei, L., Lawrence Zitnick, C., and Girshick, R. (2017). Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2901–2910. 130

Jun, H. and Nichol, A. (2023a). Shap-e: Generating conditional 3d implicit functions. *arXiv preprint arXiv:2305.02463*. 127

Jun, H. and Nichol, A. (2023b). Shap-e: Generating conditional 3d implicit functions. 130, 141

Kale, L. V. and Krishnan, S. (1993). Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108. 60

Kamath, A., Singh, M., LeCun, Y., Synnaeve, G., Misra, I., and Carion, N. (2021). Mdetr-modulated detection for end-to-end multi-modal understanding. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1780–1790. 127

Karypis, G. and Kumar, V. (1997). A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In *PPSC*. 61

Kerbl, B., Kopanas, G., Leimkühler, T., and Drettakis, G. (2023). 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics (ToG)*, 42(4):1–14. 144, 146, 147, 148, 152, 154

Khachatryan, L., Movsisyan, A., Tadevosyan, V., Henschel, R., Wang, Z., Navasardyan, S., and Shi, H. (2023). Text2video-zero: Text-to-image diffusion models are zero-shot video generators. *arXiv preprint arXiv:2303.13439*. 127, 130, 142

Klár, G., Budsberg, J., Titus, M., Jones, S., and Museth, K. (2017). Production ready mpm simulations. In *ACM SIGGRAPH 2017 Talks*, SIGGRAPH '17, New York, NY, USA. Association for Computing Machinery. 58, 62

Klár, G., Gast, T., Pradhana, A., Fu, C., Schroeder, C., Jiang, C., and Teran, J. (2016). Drucker-prager elastoplasticity for sand animation. *ACM TOG*, 35(4):103. 2, 9, 19, 48, 80, 122, 123, 148, 157

Kolve, E., Mottaghi, R., Gordon, D., Zhu, Y., Gupta, A., and Farhadi, A. (2017). Ai2-thor: An interactive 3d environment for visual ai. 103

Krishna, R., Zhu, Y., Groth, O., Johnson, J., Hata, K., Kravitz, J., Chen, S., Kalantidis, Y., Li, L.-J., Shamma, D. A., et al. (2017). Visual genome: Connecting language and vision using crowdsourced dense image annotations. *International journal of computer vision*, 123(1):32–73. 130

Laine, S., Karras, T., and Aila, T. (2013). Megakernels considered harmful: Wavefront path tracing on gpus. In *Proceedings of HPG*, pages 137–143. 44

Legg, S. and Hutter, M. (2007). Universal intelligence: A definition of machine intelligence. *Minds and Machines*, 17(4):391–444. 103

Lenz, I., Lee, H., and Saxena, A. (2015). Deep learning for detecting robotic grasps. *The International Journal of Robotics Research*, 34(4-5):705–724. 102

Lesser, S., Stomakhin, A., Daviet, G., Wretborn, J., Edholm, J., Lee, N.-H., Schweickart, E., Zhai, X., Flynn, S., and Moffat, A. (2022). Loki: a unified multiphysics simulation framework for production. *ACM Transactions on Graphics (TOG)*, 41(4):1–20. 60

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., and Zettlemoyer, L. (2019). Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*. 127

Li, C., Tang, M., Tong, R., Cai, M., Zhao, J., and Dinesh, M. (2020a). P-cloth: Interactive complex cloth simulation on multi-gpu systems using dynamic matrix assembly and pipelined implicit integrators. Technical report, Zhejiang University. 17

Li, L. H., Yatskar, M., Yin, D., Hsieh, C.-J., and Chang, K.-W. (2019a). Visualbert: A simple and performant baseline for vision and language. *arXiv preprint arXiv:1908.03557*. 127

Li, M., Ferguson, Z., Schneider, T., Langlois, T. R., Zorin, D., Panozzo, D., Jiang, C., and Kaufman, D. M. (2020b). Incremental potential contact: intersection-and inversion-free, large-deformation dynamics. *ACM Trans. Graph.*, 39(4):49. 129, 137

Li, M., Gao, M., Langlois, T., Jiang, C., and Kaufman, D. M. (2019b). Decomposed optimization time integrator for large-step elastodynamics. *ACM TOG*, 38(4):1–10. 17

Li, X., Cao, Y., Li, M., Yang, Y., Schroeder, C., and Jiang, C. (2022a). Plasticitynet: Learning to simulate metal, sand, and snow for optimization time integration. *Advances in Neural Information Processing Systems*, 35:27783–27796. 164

Li, X., Li, M., and Jiang, C. (2022b). Energetically consistent inelasticity for optimization time integration. *ACM Transactions on Graphics (TOG)*, 41(4):1–16. 80

Li, X., Qiao, Y.-L., Chen, P. Y., Jatavallabhula, K. M., Lin, M., Jiang, C., and Gan, C. (2023). PAC-neRF: Physics augmented continuum neural radiance fields for geometry-agnostic system identification. In *The Eleventh International Conference on Learning Representations*. 147, 160, 161

Li, Y., Min, M., Shen, D., Carlson, D., and Carin, L. (2018). Video generation from text. *Proceedings of the AAAI conference on artificial intelligence*, 32(1). 130

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2016). Continuous control with deep reinforcement learning. In *ICLR*. 118

Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. (2014). Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer. 127, 130

Liu, H., Hu, Y., Zhu, B., Matusik, W., and Sifakis, E. (2018). Narrow-band topology optimization on a sparsely populated grid. In *SIGGRAPH Asia 2018*, page 251. ACM. 18, 61

Liu, H., Mitchell, N., Aanjaneya, M., and Sifakis, E. (2016). A scalable schur-complement fluids solver for heterogeneous compute platforms. *ACM Transactions on Graphics (TOG)*, 35(6):1–12. 17, 18, 60

Liu, H., Zhang, C., Zhu, Y., Jiang, C., and Zhu, S.-C. (2019a). Mirroring without overimitation: Learning functionally equivalent manipulation actions. In *AAAI Conference on Artificial Intelligence (AAAI)*. 51

Liu, H., Zhang, Z., Xu, X., Zhu, Y., Liu, Y., Wang, Y., and Zhu, S.-C. (2019b). High-fidelity grasping in virtual reality using a glove-based system. In *ICRA*. 109, 111

Liu, M., Shi, R., Kuang, K., Zhu, Y., Li, X., Han, S., Cai, H., Porikli, F., and Su, H. (2023a). Openshape: Scaling up 3d shape representation towards open-world understanding. 141

Liu, R., Xiang, J., Zhao, B., Zhang, R., Yu, J., and Zheng, C. (2023b). Neural impostor: Editing neural radiance fields with explicit shape manipulation. *arXiv preprint arXiv:2310.05391*. 147

Lu, J., Batra, D., Parikh, D., and Lee, S. (2019). Vilbert: Pretraining task-agnostic visiolinguistic representations for vision-and-language tasks. *Advances in neural information processing systems*, 32. 127

Luiten, J., Kopanas, G., Leibe, B., and Ramanan, D. (2023). Dynamic 3d gaussians: Tracking by persistent dynamic view synthesis. *arXiv preprint arXiv:2308.09713*. 147, 160

Ma, P., Chen, P. Y., Deng, B., Tenenbaum, J. B., Du, T., Gan, C., and Matusik, W. (2023). Learning neural constitutive laws from motion observations for generalizable pde dynamics. *arXiv preprint arXiv:2304.14369*. 164

Macklin, M., Müller, M., Chentanez, N., and Kim, T. (2014). Unified particle physics for real-time applications. *ACM TOG*, 33(4):153. 17, 119, 137

Mahler, J., Liang, J., Niyaz, S., Laskey, M., Doan, R., Liu, X., Ojea, J. A., and Goldberg, K. (2017). Dex-net 2.0: Deep learning to plan robust grasps with synthetic point clouds and analytic grasp metrics. In *RSS*. 102

Mashayekhi, O., Qu, H., Shah, C., and Levis, P. (2017). Execution templates: Caching control plane decisions for strong scaling of data analytics. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 513–526. 60

Mashayekhi, O., Shah, C., Qu, H., Lim, A., and Levis, P. (2018). Automatically distributing eulerian and hybrid fluid simulations in the cloud. *ACM Trans. Graph.*, 37(2):24–1. 18, 60

McKIVER, W. J. and Dritschel, D. G. (2003). The motion of a fluid ellipsoid in a general linear background flow. *Journal of Fluid Mechanics*, 474:147–173. 153

Medina, D. S., St-Cyr, A., and Warburton, T. (2014). Occa: A unified approach to multi-threading languages. *arXiv preprint arXiv:1403.0968*. 60

Members, O. T. (2021). Openmpi. 82

Michel, O., Bar-On, R., Liu, R., Benaim, S., and Hanocka, R. (2022). Text2mesh: Text-driven neural stylization for meshes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13492–13502. 130

Miech, A., Zhukov, D., Alayrac, J.-B., Tapaswi, M., Laptev, I., and Sivic, J. (2019). Howto100m: Learning a text-video embedding by watching hundred million narrated video clips. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2630–2640. 131

Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., and Ng, R. (2021). Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106. 144, 146, 148

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *ICML*. 118

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533. 102, 118

Mniszewski, S. M., Belak, J., Fattebert, J.-L., Negre, C. F., Slattery, S. R., Adedoyin, A. A., Bird, R. F., Chang, C., Chen, G., Ethier, S., Fogerty, S., Habib, S., Junghans, C., Lebrun-Grandié, D., Mohd-Yusof, J., Moore, S. G., Osei-Kuffuor, D., Plimpton, S. J., Pope, A., Reeve, S. T., Ricketson, L., Scheinberg, A., Sharma, A. Y., and Wall, M. E. (2021). Enabling particle applications for exascale computing platforms. *The International Journal of High Performance Computing Applications*, 0(0):10943420211022829. 56, 60, 66

Mohammad Khalid, N., Xie, T., Belilovsky, E., and Popa, T. (2022). Clip-mesh: Generating textured meshes from text using pretrained image-text models. In *SIGGRAPH Asia 2022 conference papers*, pages 1–8. 130

Montazeri, Z., Xiao, C., Zheng, C., Zhao, S., et al. (2019). Mechanics-aware modeling of cloth appearance. *arXiv preprint arXiv:1904.11116*. 2, 19

Moravčík, M., Schmid, M., Burch, N., Lisỳ, V., Morrill, D., Bard, N., Davis, T., Waugh, K., Johanson, M., and Bowling, M. (2017). Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513. 103

Müller, T., Evans, A., Schied, C., and Keller, A. (2022). Instant neural graphics primitives with a multiresolution hash encoding. *ACM Transactions on Graphics (ToG)*, 41(4):1–15. 146, 155

Museth, K. (2013). Vdb: High-resolution sparse volumes with dynamic topology. *ACM transactions on graphics (TOG)*, 32(3):1–22. 18, 61

Museth, K. (2021). Nanovdb: A gpu-friendly and portable vdb data structure for real-time rendering and simulation. In *ACM SIGGRAPH 2021 Talks*, SIGGRAPH '21, New York, NY, USA. Association for Computing Machinery. 61

Nagasawa, K., Suzuki, T., Seto, R., Okada, M., and Yue, Y. (2019). Mixing sauces: a viscosity blending model for shear thinning fluids. *ACM TOG*, 38(4):1–17. 2, 19

Newton, I. and Colson, J. (1736). *The Method of Fluxions and Infinite Series; with Its Application to the Geometry of Curve-lines*. Henry Woodfall; and sold by John Nourse. 108

Nvidia (2019). Nccl library. `https://github.com/NVIDIA/nccl`. 40

Ordonez, V., Kulkarni, G., and Berg, T. (2011). Im2text: Describing images using 1 million captioned photographs. *Advances in neural information processing systems*, 24. 127, 130

Pan, Y., Qiu, Z., Yao, T., Li, H., and Mei, T. (2017). To create what you tell: Generating videos from captions. In *Proceedings of the 25th ACM international conference on Multimedia*, pages 1789–1798. 130

Park, K., Sinha, U., Barron, J. T., Bouaziz, S., Goldman, D. B., Seitz, S. M., and Martin-Brualla, R. (2021). Nerfies: Deformable neural radiance fields. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5865–5874. 146

Peng, Y., Yan, Y., Liu, S., Cheng, Y., Guan, S., Pan, B., Zhai, G., and Yang, X. (2022). Cagenerf: Cage-based neural radiance field for generalized 3d deformation and animation. *Advances in Neural Information Processing Systems*, 35:31402–31415. 144, 147

Pfaff, T., Thuerey, N., Cohen, J., Tariq, S., and Gross, M. (2010). Scalable fluid simulation using anisotropic turbulence particles. *ACM TOG*, 29(6):174. 17, 59

Pilkington, N. (2022). Dronedeploy nerf dataset. 155

Poole, B., Jain, A., Barron, J. T., and Mildenhall, B. (2022). Dreamfusion: Text-to-3d using 2d diffusion. *arXiv preprint arXiv:2209.14988*. 127, 130

Pumarola, A., Corona, E., Pons-Moll, G., and Moreno-Noguer, F. (2021). D-nerf: Neural radiance fields for dynamic scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10318–10327. 146

Qi, S., Huang, S., Wei, P., and Zhu, S.-C. (2017). Predicting human activities using stochastic grammar. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 1164–1172. 115

Qi, S., Zhu, Y., Huang, S., Jiang, C., and Zhu, S.-C. (2018). Human-centric indoor scene synthesis using stochastic grammar. In *CVPR*. 107

Qiao, Y.-L., Gao, A., and Lin, M. (2022). Neuphysics: Editable neural geometry and physics from monocular videos. *Advances in Neural Information Processing Systems*, 35:12841–12854. 147

Qiao, Y.-L., Gao, A., Xu, Y., Feng, Y., Huang, J.-B., and Lin, M. C. (2023). Dynamic mesh-aware radiance fields. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 385–396. 147

Qiu, Y., Reeve, S. T., Li, M., Yang, Y., Slattery, S. R., and Jiang, C. (2023). A sparse distributed gigascale resolution material point method. *ACM Transactions on Graphics*, 42(2):1–21. 137, 148

Qu, H., Mashayekhi, O., Shah, C., and Levis, P. (2018). Decoupling the control plane from program control flow for flexibility and performance in cloud computing. In *Proceedings of the thirteenth euroSys conference*, pages 1–13. 60

Qu, H., Mashayekhi, O., Shah, C., and Levis, P. (2020). Accelerating distributed graphical fluid simulations with micro-partitioning. *Computer Graphics Forum*, 39(1):375–388. 60, 61

Raafat, M. (2023). BlenderNeRF. 155, 158

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., et al. (2021). Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, pages 8748–8763. PMLR. 127

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. (2019). Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9. 127

Ram, D., Gast, T., Jiang, C., Schroeder, C., Stomakhin, A., Teran, J., and Kavehpour, P. (2015). A material point method for viscoelastic fluids, foams and sponges. In *ACM SIGGRAPH / Eurographics Symposium on Computer Animation (SCA)*, pages 157–163. ACM. 2, 19

Ramachandran, D. and Amir, E. (2007). Bayesian inverse reinforcement learning. In *IJCAI*, volume 51, pages 1–4. 119

Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., and Chen, M. (2022). Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*. 127, 130

Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., Chen, M., and Sutskever, I. (2021). Zero-shot text-to-image generation. In *International Conference on Machine Learning*, pages 8821–8831. PMLR. 127, 129

Reed, S., Akata, Z., Yan, X., Logeswaran, L., Schiele, B., and Lee, H. (2016). Generative adversarial text to image synthesis. In *International conference on machine learning*, pages 1060–1069. PMLR. 129

Rombach, R., Blattmann, A., Lorenz, D., Esser, P., and Ommer, B. (2022). High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10684–10695. 127, 130

Rustico, E., Bilotta, G., Herault, A., Del Negro, C., and Gallo, G. (2012). Advances in multi-gpu smoothed particle hydrodynamics simulations. *IEEE TPDS*, 25(1):43–52. 17

Saharia, C., Chan, W., Saxena, S., Li, L., Whang, J., Denton, E. L., Ghasemipour, K., Gontijo Lopes, R., Karagol Ayan, B., Salimans, T., et al. (2022). Photorealistic text-to-image diffusion models with deep language understanding. *Advances in Neural Information Processing Systems*, 35:36479–36494. 127

Schönberger, J. L. and Frahm, J.-M. (2016). Structure-from-motion revisited. In *Conference on Computer Vision and Pattern Recognition (CVPR)*. 157

Schönberger, J. L., Zheng, E., Pollefeys, M., and Frahm, J.-M. (2016). Pixelwise view selection for unstructured multi-view stereo. In *European Conference on Computer Vision (ECCV)*. 157

Schuhmann, C., Beaumont, R., Vencu, R., Gordon, C., Wightman, R., Cherti, M., Coombes, T., Katta, A., Mullis, C., Wortsman, M., et al. (2022). Laion-5b: An open large-scale dataset for training next generation image-text models. *arXiv preprint arXiv:2210.08402*. 127, 130

Setaluri, R., Aanjaneya, M., Bauer, S., and Sifakis, E. (2014). Spgrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Transactions on Graphics (TOG)*, 33(6):1–12. 18, 23, 31, 61, 70, 94

Shah, C., Hyde, D., Qu, H., and Levis, P. (2018a). Distributing and load balancing sparse fluid simulations. *Computer Graphics Forum*, 37(8):35–46. 18, 60, 61

Shah, S., Dey, D., Lovett, C., and Kapoor, A. (2018b). Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*, pages 621–635. Springer. 103

Sharma, P., Ding, N., Goodman, S., and Soricut, R. (2018). Conceptual captions: A cleaned, hypernymed, image alt-text dataset for automatic image captioning. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2556–2565. 127, 130

Shu, T., Gao, X., Ryoo, M. S., and Zhu, S.-C. (2017). Learning social affordance grammar from videos: Transferring human interactions to human-robot interactions. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 1669–1676. IEEE. 116

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484. 103

Simo, J. C. and Hughes, T. J. (2006). *Computational inelasticity*, volume 7. Springer Science & Business Media. 150

Singer, U., Polyak, A., Hayes, T., Yin, X., An, J., Zhang, S., Hu, Q., Yang, H., Ashual, O., Gafni, O., et al. (2022). Make-a-video: Text-to-video generation without text-video data. *arXiv preprint arXiv:2209.14792*. 127, 130

Slattery, S., Junghans, C., Reeve, S., Lebrun-Grandié, D., Fogerty, S., Bird, R., Qiu, Y., Chen, G., Halver, R., Scheinberg, A., Smith, C., Medwedeff, E., Prokopenko, A., Weinberg, E., Eibl, S., and Fernandez, J. D. C. (2021). Ecp-copa/cabana: Version 0.4.0. `https://github.com/ECP-copa/Cabana`. 21

Slattery, S., Reeve, S. T., Junghans, C., Lebrun-Grandié, D., Bird, R., Chen, G., Fogerty, S., Qiu, Y., Schulz, S., Scheinberg, A., Isner, A., Chong, K., Moore, S., Germann, T., Belak, J., and Mniszewski, S. (2022). Cabana: A performance portable library for particle-based simulations. *Journal of Open Source Software*, 7(72):4115. 56, 60

Snir, M., Gropp, W., Otto, S., Huss-Lederman, S., Dongarra, J., and Walker, D. (1998). *MPI–the Complete Reference: the MPI core*, volume 1. MIT press. 60, 66

Song, S., Yu, F., Zeng, A., Chang, A. X., Savva, M., and Funkhouser, T. (2017). Semantic scene completion from a single depth image. In *CVPR*. 107

Sorkine, O. and Alexa, M. (2007). As-rigid-as-possible surface modeling. In *Symposium on Geometry processing*, volume 4, pages 109–116. Citeseer. 163

Srinivasan, K., Raman, K., Chen, J., Bendersky, M., and Najork, M. (2021). Wit: Wikipedia-based image text dataset for multimodal multilingual machine learning. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2443–2449. 130

Stomakhin, A., Howes, R., Schroeder, C., and Teran, J. (2012a). Energetically consistent invertible elasticity. In *Proc Symp Comp Anim*, pages 25–32. 42, 48

Stomakhin, A., Howes, R., Schroeder, C., and Teran, J. M. (2012b). Energetically consistent invertible elasticity. In *Proceedings of the 11th ACM SIGGRAPH/Eurographics conference on Computer Animation*, pages 25–32. 82

Stomakhin, A., Schroeder, C., Chai, L., Teran, J., and Selle, A. (2013). A material point method for snow simulation. *ACM Transactions on Graphics (TOG)*, 32(4):1–10. 2, 19, 62, 80, 129, 148, 150

Stomakhin, A., Schroeder, C., Jiang, C., Chai, L., Teran, J., and Selle, A. (2014). Augmented mpm for phase-change and varied materials. *ACM TOG*, 33(4):138. 19

Sulsky, D., Chen, Z., and Schreyer, H. L. (1994). A particle method for history-dependent materials. *Computer methods in applied mechanics and engineering*, 118(1-2):179–196. 19

Sulsky, D., Zhou, S., and Schreyer, H. L. (1995). Application of a particle-in-cell method to solid mechanics. *Computer physics communications*, 87(1-2):236–252. 19, 129

Sun, C., Sun, M., and Chen, H.-T. (2022). Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5459–5469. 146

Surmin, I., Bashinov, A., Bastrakov, S., Efimenko, E., Gonoskov, A., and Meyerov, I. (2015). Dynamic load balancing based on rectilinear partitioning in particle-in-cell plasma simulation. In *International Conference on Parallel Computing Technologies*, pages 107–119. Springer. 73, 77, 78

Tampubolon, A. P., Gast, T., Klár, G., Fu, C., Teran, J., Jiang, C., and Museth, K. (2017). Multi-species simulation of porous sand and water mixtures. *ACM TOG*, 36(4):105. 20, 48, 82, 123

Tancik, M., Weber, E., Ng, E., Li, R., Yi, B., Wang, T., Kristoffersen, A., Austin, J., Salahi, K., Ahuja, A., et al. (2023). Nerfstudio: A modular framework for neural radiance field development. In *ACM SIGGRAPH 2023 Conference Proceedings*, pages 1–12. 155

Tang, J., Ren, J., Zhou, H., Liu, Z., and Zeng, G. (2023). Dreamgaussian: Generative gaussian splatting for efficient 3d content creation. *arXiv preprint arXiv:2309.16653*. 153

Tang, M., Wang, H., Tang, L., Tong, R., and Dinesh, M. (2016). Cama: Contact-aware matrix assembly with unified collision handling for gpu-based cloth simulation. *Computer Graphics Forum*, 35(2):511–521. 18

Tevet, G., Gordon, B., Hertz, A., Bermano, A. H., and Cohen-Or, D. (2022). Motionclip: Exposing human motion generation to clip space. *arXiv preprint arXiv:2203.08063*. 130

Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *IROS*. 103

TOP500.org (2019). Top500 list november 2019. 21

Trott, C. R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D. S., Ibanez, D., Liber, N., Madsen, J., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., and Wilke, J. (2022). Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):805–817. Conference Name: IEEE Transactions on Parallel and Distributed Systems. 56, 60, 64, 65

Tutorials, N. Niagara tutorials. 122

Valko, M., Ghavamzadeh, M., and Lazaric, A. (2012). Semi-supervised apprenticeship learning. In *EWRL*. 119

Vantzos, O., Raz, S., and Ben-Chen, M. (2018). Real-time viscous thin films. *ACM TOG*, 37(6):1–10. 17, 59

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30. 127

Verma, K., Szewc, K., and Wille, R. (2017). Advanced load balancing for sph simulations on multi-gpu architectures. In *High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE. 17

Wald, I. (2010). Fast construction of sah bvhs on the intel many integrated core (mic) architecture. *IEEE Transactions on Visualization and Computer Graph (TVCG)*, 18(1):47–57. 21

Wang, C., Chai, M., He, M., Chen, D., and Liao, J. (2022). Clip-nerf: Text-and-image driven manipulation of neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3835–3844. 130

Wang, H. (2018). Rule-free sewing pattern adjustment with precision and efficiency. *ACM TOG*, 37(4):53. 18

Wang, P., Liu, L., Liu, Y., Theobalt, C., Komura, T., and Wang, W. (2021). Neus: Learning neural implicit surfaces by volume rendering for multi-view reconstruction. In Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W., editors, *Advances in Neural Information Processing Systems*. 160

Wang, S., Ding, M., Gast, T. F., Zhu, L., Gagniere, S., Jiang, C., and Teran, J. M. (2019). Simulation and visualization of ductile fracture with the material point method. *ACM TOG*, 2(2):1–20. 20

Wang, X., Qiu, Y., Slattery, S. R., Fang, Y., Li, M., Zhu, S.-C., Zhu, Y., Tang, M., Manocha, D., and Jiang, C. (2020). A massively parallel and scalable multi-gpu material point method. *ACM Transactions on Graphics (TOG)*, 39(4):30–1. 53, 54, 58, 60, 62, 66, 69, 70, 72, 148

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., and Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In *ICML*. 118

Weber, N. and Goesele, M. (2014). Auto-tuning complex array layouts for gpus. In *EGPGV*, pages 57–64. 21

Weller, R., Debowski, N., and Zachmann, G. (2017). kdet: Parallel constant time collision detection for polygonal objects. *Computer Graphics Forum*, 36(2):131–141. 18

Willhalm, T. and Popovici, N. (2008). Putting intel® threading building blocks to work. In *Proceedings of the international workshop on Multicore software engineering*, pages 3–4. ACM. 17, 59

Winchenbach, R., Hochstetter, H., and Kolb, A. (2016). Constrained neighbor lists for sph-based fluid simulations. In *Symposium on Computer Animation (SCA)*, pages 49–56. Eurographics Association. 17, 60

Wolper, J., Fang, Y., Li, M., Lu, J., Gao, M., and Jiang, C. (2019). Cd-mpm: continuum damage material point methods for dynamic fracture animation. *ACM TOG*, 38(4):1–15. 15, 20, 41, 48, 49, 80, 94, 122, 123

Wretborn, J., Armiento, R., and Museth, K. (2017). Animation of crack propagation by means of an extended multi-body solver for the material point method. *Computers & Graphics*, 69:131–139. 20

Wu, C., Huang, L., Zhang, Q., Li, B., Ji, L., Yang, F., Sapiro, G., and Duan, N. (2021). Godiva: Generating open-domain videos from natural descriptions. *arXiv preprint arXiv:2104.14806*. 130

Wu, G., Yi, T., Fang, J., Xie, L., Zhang, X., Wei, W., Liu, W., Tian, Q., and Wang, X. (2023). 4d gaussian splatting for real-time dynamic scene rendering. *arXiv preprint arXiv:2310.08528*. 147, 149, 152, 160

Wu, K., Truong, N., Yuksel, C., and Hoetzlein, R. (2018). Fast fluid simulations with sparse volumes on the gpu. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2018)*, 37(2):157–167. 17, 60

Xia, F., Zamir, A. R., He, Z., Sax, A., Malik, J., and Savarese, S. (2018). Gibson env: Real-world perception for embodied agents. In *CVPR*. 103

Xie, X., Liu, H., Zhang, Z., Qiu, Y., Gao, F., Qi, S., Zhu, Y., and Zhu, S.-C. (2019). Vrgym: A virtual testbed for physical and interactive ai. In *Proceedings of the ACM Turing Celebration Conference-China*. 51

Xiong, Q., Li, B., and Xu, J. (2013). Gpu-accelerated adaptive particle splitting and merging in sph. *Computer Physics Communications*, 184(7):1701–1707. 17

Xu, J., Mei, T., Yao, T., and Rui, Y. (2016). Msr-vtt: A large video description dataset for bridging video and language. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5288–5296. 131

Xu, Q., Xu, Z., Philip, J., Bi, S., Shu, Z., Sunkavalli, K., and Neumann, U. (2022). Point-nerf: Point-based neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5438–5448. 146, 147

Xu, T. and Harada, T. (2022). Deforming radiance fields with cages. In *European Conference on Computer Vision*, pages 159–175. Springer. 144, 160, 161

Xu, T., Zhang, P., Huang, Q., Zhang, H., Gan, Z., Huang, X., and He, X. (2018). Attngan: Fine-grained text to image generation with attentional generative adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1316–1324. 129

Yan, X., Li, C.-F., Chen, X.-S., and Hu, S.-M. (2018). Mpm simulation of interacting fluids and solids. *Computer Graphics Forum*, 37(8):183–193. 20

Yang, Z., Gao, X., Zhou, W., Jiao, S., Zhang, Y., and Jin, X. (2023a). Deformable 3d gaussians for high-fidelity monocular dynamic scene reconstruction. *arXiv preprint arXiv:2309.13101*. 147

Yang, Z., Yang, H., Pan, Z., Zhu, X., and Zhang, L. (2023b). Real-time photorealistic dynamic scene representation and rendering with 4d gaussian splatting. *arXiv preprint arXiv:2310.10642*. 147

Yu, L.-F., Yeung, S.-K., Tang, C.-K., Terzopoulos, D., Chan, T. F., and Osher, S. J. (2011). Make it home: automatic optimization of furniture arrangement. *TOG*, 30(4):86. 107

Yuan, Y.-J., Sun, Y.-T., Lai, Y.-K., Ma, Y., Jia, R., and Gao, L. (2022). Nerf-editing: geometry editing of neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18353–18364. 144, 147, 160, 161, 162, 163

Yue, Y., Smith, B., Batty, C., Zheng, C., and Grinspun, E. (2015). Continuum foam: A material point method for shear-dependent flows. *ACM TOG*, 34(5):160. 2, 10, 19, 157

Yue, Y., Smith, B., Chen, P. Y., Chantharayukhonthorn, M., Kamrin, K., and Grinspun, E. (2018). Hybrid grains: Adaptive coupling of discrete and continuum simulations of granular media. In *SIGGRAPH Asia 2018*, page 283. ACM. 2, 19, 20

Zenker, E., Worpitz, B., Widera, R., Huebl, A., Juckeland, G., Knüpfer, A., Nagel, W. E., and Bussmann, M. (2016). Alpaka–an abstraction library for parallel kernel acceleration. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 631–640. IEEE. 60

Zhang, H., Xu, T., Li, H., Zhang, S., Wang, X., Huang, X., and Metaxas, D. N. (2017). Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 5907–5915. 129

Zhang, P., Li, X., Hu, X., Yang, J., Zhang, L., Wang, L., Choi, Y., and Gao, J. (2021). Vinvl: Making visual representations matter in vision-language models. *CVPR 2021*. 127

Zhao, J., Chen, Y., Zhang, H., Xia, H., Wang, Z., and Peng, Q. (2019). Physically based modeling and animation of landslides with mpm. *The Visual Computer*, 35(9):1223–1235. 2, 19

Zhou, Q. and Jacobson, A. (2016). Thingi10k: A dataset of 10,000 3d-printing models. *arXiv preprint arXiv:1605.04797*. 141

Zhu, J.-Y., Park, T., Isola, P., and Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232. 127

Zhu, Y. and Bridson, R. (2005). Animating sand as a fluid. *ACM TOG*, 24(3):965–972. 19

Zhu, Y., Jiang, C., Zhao, Y., Terzopoulos, D., and Zhu, S.-C. (2016). Inferring forces and learning human utilities from videos. In *Conference on Computer Vision and Pattern Recognition (CVPR)*. 51

Zhu, Y., Zhao, Y., and Zhu, S.-C. (2015). Understanding tools: Task-oriented object modeling, learning and recognition. In *Conference on Computer Vision and Pattern Recognition (CVPR)*. 51

Ziebart, B. D., Maas, A. L., Bagnell, J. A., and Dey, A. K. (2008). Maximum entropy inverse reinforcement learning. In *AAAI*, volume 8, pages 1433–1438. Chicago, IL, USA. 119

Zong, Z., Li, X., Li, M., Chiaramonte, M. M., Matusik, W., Grinspun, E., Carlberg, K., Jiang, C., and Chen, P. Y. (2023). Neural stress fields for reduced-order elastoplasticity and fracture. *arXiv preprint arXiv:2310.17790*. 154, 157