DOI: 10.1111/cgf.14643
ACM SIGGRAPH / Eurographics Symposium on Computer Animation 2022
D. L. Michels and S. Pirk
(Guest Editors)
Volume 41 (2022), Number 8

Learning Physics with a Hierarchical Graph Network

Nuttapong Chentanez[1,2], Stefan Jeschke[1], Matthias Müller[1], Miles Macklin[1]

[1]NVIDIA
[2]Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University

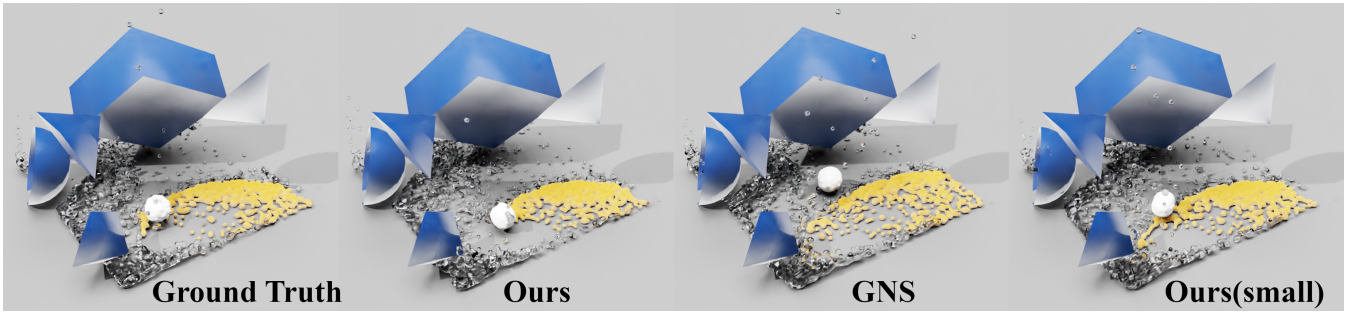

Figure 1: *Ground truth vs. Our method vs. GNS [SGGP*20] vs. Our method (smaller version) for a test scene with obstacles. The scene is not used for training. Our method yields less error, trains and inferences substantially faster than GNS and consumes less memory.*

Abstract

*We propose a hierarchical graph for learning physics and a novel way to handle obstacles. The finest level of the graph consist of the particles itself. Coarser levels consist of the cells of sparse grids with successively doubling cell sizes covering the volume occupied by the particles. The hierarchical structure allows for the information to propagate at great distance in a single message passing iteration. The novel obstacle handling allows the simulation to be obstacle aware without the need for ghost particles. We train the network to predict effective acceleration produced by multiple sub-steps of 3D multi-material material point method (MPM) simulation consisting of water, sand and snow with complex obstacles. Our network produces lower error, trains up to 7.0X faster and inferences up to 11.3X faster than [SGGP*20]. It is also, on average, about 3.7X faster compared to Taichi Elements simulation running on the same hardware in our tests.*
(see https://www.acm.org/publications/class-2012)

CCS Concepts

• ***Computing methodologies*** → ***Neural networks; Physical simulation;***

1. Introduction

Over the past few decades, there has been a large number of physics simulation research in computer graphics. One promising simulation method is the material point method (MPM), which has been demonstrated to be able to simulate wide varieties of materials such as water, sand, snow, goop, lava, elastic objects, cloth and thin shells. A MPM solver can either be explicit, which does not require a linear solver or implicit, which requires a linear solver. Explicit MPM, while simpler to implement and cheaper per sub-step, requires smaller sub-step size to keep the simulation stable. Implicit MPM, on the other hand, is more complex to implement and runs slower per sub-step. However, it can typically take larger time steps.

Recently, there has been a number of works attempting to use deep learning to learn to simulate physics such as [SGGP*20, PFSGB20, TKC21]. Benefits of using deep learning include a potential to simulate with larger time steps, differentiability and flexibility to trade-off speed and quality, to name a few.

We propose a hierarchical graph network capable of predicting the effective acceleration resulting from multiple sub time steps of MPM simulation. To the best of our knowledge, this is the first time that deep learning is demonstrated to be capable of learning 3D multi-materials MPM simulation with complex obstacles and

run faster than the simulator, on the same hardware. Our main contributions include:

- Hierarchical graph network that allows information to propagate to and be gathered from far away.
- Obstacle handling that does not require the use of ghost particles.
- Demonstration that the network can learn multi-material 3D MPM with complex obstacles.
- Demonstration of its advantage over previous method: lower error, faster training, faster inferencing and lower memory footprint, as well as being faster than the MPM simulator, on the same hardware.
- Demonstration of its ability to trade off speed and accuracy.

2. Related Works

MPM was first introduced in [SZS95]. It became popular in computer graphics when it was used for simulating snow in [SSC*13]. Affine particle-in-cell (APIC) method was introduced to improve its accuracy in [JSS*15]. To reduce computational cost, moving least square MPM (MLS-MPM) was proposed in [HFG*18]. An excellent introduction to MPM for computer graphics can be found in a course note [JST*16]. More recent works include multi-GPU MPM [WQS*20], anisotropic damage [WCL*20], arbitrary updated Lagrangian MPM [SXHA21], second order accurate in time MPM [SXH*21], integration scheme to reduce dissipation [FGW*21] and momentum conserving MPM [CKMR*21] to name a few.

MLS-MPM, in particular, has been popularized due to its speed by an open source implementation Taichi Elements [tai]. It uses a domain specific language Taichi [HLA*19, HAL*20, HLY*21] which is demonstrated to typically run faster than even a hand optimized implementation in most cases. We use it for generating our training data.

Deep learning (DL) has been applied to learn physics in recent years. A great introduction to the subject can be found in [TBMK22]. A closely related field of using physics to guide DL is surveyed in [WY21]. We focus on the related work that apply DL to learn physics of particles based simulation, as they are most relevant to our works.

Li et al. [LWT*18] use an interaction graph to learn particle based fluid-rigid body and rigid body-elastic object interactions. Ummenhofer et al. [UPTK20] use continuous convolution to learn liquid simulation with obstacles. Martinkus et al. [MLP20] learn 2D n-body problem with a graph build on a quadtree. Tumanov et al. [TKC21] uses a sub-pixel convolution to learn particle based fluid simulations with obstacles. Lino et al. [LCBF21] use a multi-scale graph network to learn 2D fluid on an unstructured point set. Park et al. [PLL21] use time-wise point net to learn physics of 2D deformable objects. Chen et al. [CCGC21] uses neural network to do model reduction for a single material deformable object simulated with MPM without obstacle interaction. Xue et a. [XAM22] use a graph network to learn physics of soft mechanical metamaterial. Odot et al. [OHC22] use a fully connected network to learn deformation of a hyper-elastic material simulated with FEM.

Sanchez-Gonzalez et al. [SGGP*20] train graph based message passing networks to learn 2D multi-material MPM, 2D MPM with obstacles and 3D MPM without obstacles. They named their method graph network simulation (GNS). It is since adopted in many works. Mayr et al. [MLM*21] insert virtual boundary nodes and features in GNS to learn granular flow in triangle mesh domains. Pfaff et al. [PFSGB20] extends GNS to include material space connection allowing for learning mesh based elastic bodies. Li et al. [LF22] use GNSs to separately learn advection, collision and pressure projection steps for particle based fluid simulation. Klimesch et al. [KHT22] investigate the use of multi-step loss to improve generalizability of GNS for learning particle based fluid simulation. Li et al. [LMYBF22] use GNS to learn dynamics of Lennard-Jones systems and water particle systems. Tuomainen et al. [TBMK22] use GNS to learn granular flow and then train a controller to pour sand from a cup that matches real world behavior. The main drawback of GNS is that it only propagate information via particles immediate neighbors. This allows for information to propagate only up to the neighbor's radius in a one iteration. Moreover, the number of neighbors can be large, around a hundred in 3D, which leads to large memory usage and computation time. Our method utilizes a hierarchy of grids to allow information to propagate further distance in one iteration, while also having fewer connections. GNS handles obstacles using extra boundary particles, while our method uses additional features but no extra particles. This also leads to smaller network size, as the extra particles are not needed. As GNS is the only method that we are aware of that was demonstrated to be able to learn multi-material MPM, we compare our work with it.

3. Method

We let Δt be the time step and let x_i^n, v_i^n and a_i^n denote the position, velocity and acceleration of particle i at time step n respectively. t_i denotes the type of particle i. f_i denotes the feature vector of particle i. m is the number of time steps our network takes into account in predicting the acceleration. Our main simulation loop is summarized in Algorithm 1. The compute_features and

Algorithm 1: The main loop for our DL based simulation, where the acceleration is predicted by a neural network.

$n = 0$
while *simulating* **do**
  for *each particle i* **do**
    $f_i = \text{compute\_features}(x_i^n, x_i^{n-1}, .., x_i^{n-m}, t_i)$
  end
  $\tilde{a}^n = \text{predict\_acceleration}(f)$
  for *each particle i* **do**
    $v_i^{n+1} = v_i^n + \Delta t \tilde{a}^n$
    $x_i^{n+1} = x_i^n + \Delta t v_i^{n+1}$
  end
  $n = n + 1$
end

predict_acceleration steps are explained in Sections 3.1 and 3.2 respectively. Without loss of generality, we use $\Delta t = 1$ in training and inferencing for simplicity. We represent the obstacle as a sparse grid with grid spacing Δx where each cell stores the average normal of the obstacle within the cell or zero if there's no such obstacle. The

average estimated by sampling the obstacle surface uniformly with points so that no two points are closer than $\frac{\Delta x}{4}$. For each point, its normal is taken from the surface. The normal is negated if the dot product of the normal and the vector from the point to the center of the enclosing cell is negative. The normals of all the points in a given cell are added and normalized to obtain the cell normal. We let n_i^o denote the normal of the cell containing x_i^n. Δx is chosen to be the same as the grid spacing used for MPM simulation.

3.1. Feature Computation

The features f_i is in the form $(v_i^n, v_i^{n-1}, ..., v_i^{n-m+1}, n_i^o, d_i, e_i)$ where d_i is a 6D vector representing the signed distance of particle i to the 6 planes of the axis aligned bounding box (AABB) of the domain clamped to a value $d_{\max}$, $v_i^{n-j} = x_i^{n-j} - x_i^{n-j-1}$. e_i is an E dimension embedding of particle type, t_i. The embedding maps each particle type to a R^E vector, whose value is learned during training, similar to [SGGP*20].

We perform normalization of the input before feeding to the network as follows: v_i^{n-j} are normalized with the mean and standard deviation computed from the training data. n_i^o is already normalized and d_i is divided by $d_{\max}$. e_i are learnable parameters so they are not normalized. The network output is normalized with the mean and standard deviation of $v_i^{n+1} - v_i^n$ computed from the whole training data.

We note that [SGGP*20] treats obstacles by approximating them with particles and include them as a type of particle. We instead handle obstacles by having n_i^o and do not have any obstacle particle. Because there can be a large number of obstacle particles in a complex 3D scene, our method uses far fewer number of particles in such scene. This will be discussed in more detail in the result section.

3.2. Acceleration Prediction

We predict the acceleration using a hierarchical graph, which allows information to propagate over large distance quickly. The main architecture is shown in Figure 2. We first build a sparse grid, whose grid spacing is R. The sparse grid cells are those whose cell centers have non-zero tri-linear interpolation weights for some particles, as illustrated in Figure 3. Particle information will be transferred to these 8 nearby cell centers as will be explained shortly. The next coarser level is constructed by merging 2x2x2 cells into bigger cells. We construct L levels of grid. For the sake of clarity in explaining our method, Figure 2 demonstrates for $L = 3$ and the computation steps are numbered. Note that in the actual implementation, we use $L = 4$ in all examples.

We make use of a building block of a fully connected network (FCN) with a single hidden layer using LeakyReLU activation, optionally followed by a layer normalization, as shown in Figure 4. We will refer to this building block as FCN in the rest of the paper. Unless explicitly stated, layer normalization is always used. To avoid confusion, FCNs that appear in different places in our architecture do not share any weights.

In step 0, Particle Features -> Latent Space Encoding, we use FCN on f of each particle to compute a latent space encoding, f^p which will be used as the input to step 1.

In step 1, Particle -> Grid, we create (particle i, cell j) pairs for each particle to the nearby 8 cells centers with non-zero tri-linear interpolation weights. We then use FCN with input [f_i^p from step 0, offset vector from the particle i to cell center j] and accumulate the output to each cell, weighted by the interpolation weight. The accumulated value at each cell is referred to as f^c

In step 2, Cell -> Cell message passing (similar to steps 4,6,8,10), we perform passes of graph message passing [BHB*18] in this step. Specifically, the graph nodes are the active cells. The graph directed edges link adjacent active cells, ie. each active cell is linked to upto 6 active cells. Let f^e denote the features at the directed edges, which we initialize to the difference of the positions of the two end points of the edges divided by the grid spacing. M message passing steps are then performed with residual connections to update the values of f^c and replace/update f^{le}. The first iteration of the message passing replaces the value of f^e while the remaining iterations update it with residual connection. The process is illustrated in Figure 5, where each FCN^e operates on each directed edge (i, j) between the cell i to j and takes in the input as $[f_i^c, f_j^c, f_{i,j}^e]$. Its output is then used to replace/update $f_{i,j}^e$ and scatter to node j and passed through FCN^c to compute an update to f_j^c.

In step 3, Fine Grid -> Coarse Grid (same as 5), we create (fine cell i_{fine}, coarse cell j_{coarse}) pairs from adjacent 2x2x2 fine cells merging into a coarse cell. We then, for each pair, use an FCN with input [$f^c_{i_{\text{fine}}}$, offset vector from the fine cell center i_{fine} to coarse cell center j_{coarse}] and accumulate the output to the coarse cell j_{coarse} to obtain $f^c_{j_{\text{coarse}}}$.

In step 7, Coarse Grid -> Fine Grid (same as 9), we create (coarse cell i_{coarse}, fine cell j_{fine}) pairs from coarse cell splitting into 2x2x2 fine cells. We then, for each pair, use an FCN with input [$f^c_{i_{\text{coarse}}}$, $f^c_{j_{\text{fine}}}$, offset vector from the coarse cell center i_{coarse} to fine cell center j_{fine}] and add the output to $f^c_{j_{\text{fine}}}$ to refine its value.

In step 11, Grid -> Particles, we create (cell i, particle j) pairs for each particle to the nearby 8 cell centers with non-zero tri-linear interpolation weights. We then, for each pair, use an FCN with input [f_i^c, f_j^p, offset vector from the cell center i to particle j] and accumulate the output, weighted by the interpolation weight, to Δf_j^p.

In step 12, Particle Latent Space Encoding -> Acceleration, for each particle i, we use an FCN with input $f_i^p + \Delta f_i^p$ to predict the acceleration of the particle.

3.3. Training

Our training data consists of 800 randomly generated scenes of 1x1x1 unit size. Each scene has 5 randomly chosen obstacles from half box, square, half sphere, cone, pyramid with randomized size between 0.1 to 0.5 units placed randomly with random orientation, such that they are not overlapping. Each scene consists of 4 groups of particles either a sphere or a cube shape with side/diameter of 0.08 units randomly placed so that they do not overlap with obstacles. Each group is randomly chosen to be water, sand or snow.

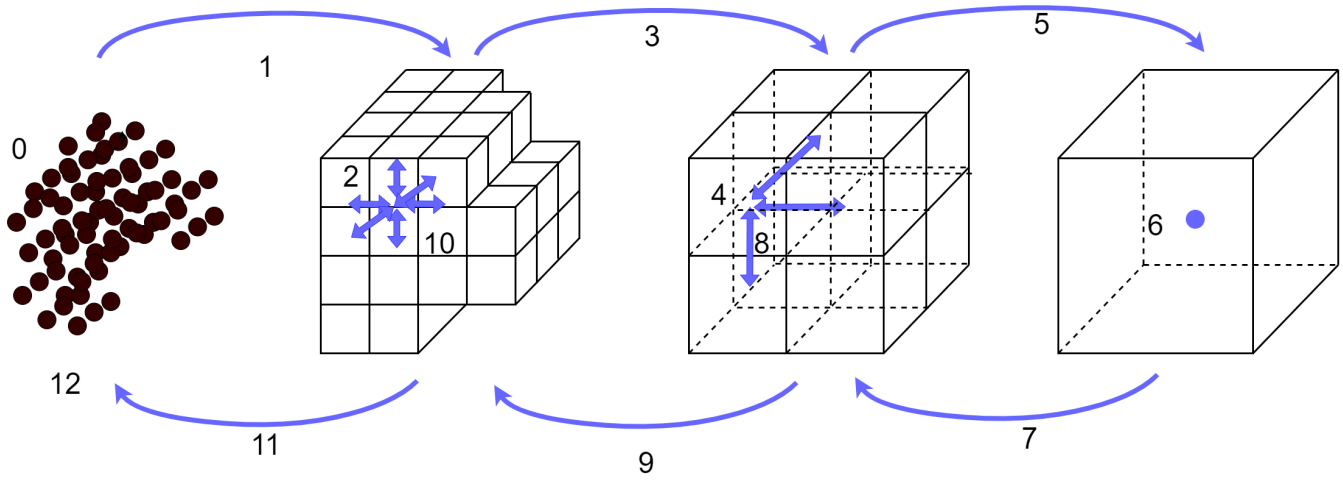

Figure 2: *Illustration of the main idea of our method. First, latent space encoding is computed from the features at each particle (0). Information is then transferred to the cell centers of the sparse grid enclosing the particles (1). The information then flows to progressively coarser grids (3,5), with message passing to propagate information at each level (2,4,6). It then flows back to finer grids (7,9), again with message passing at each level (8,10), and flows back to particles (11). Finally, the acceleration is computed from the updated latent space encoding (12).*

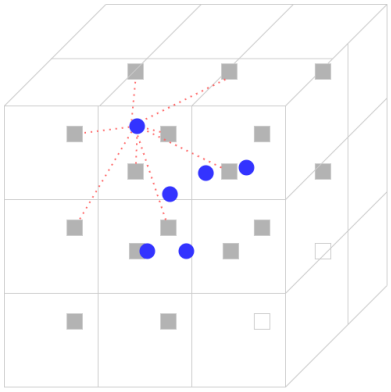

Figure 3: *Illustration of the tri-linear interpolation weights and active cells calculation. Cell centers are shown as squares. A particle has non zero tri-linear interpolation with the nearby cells, shown as red dotted lines for one particle. In this particular case, all cells except the two bottom right cells are active.*

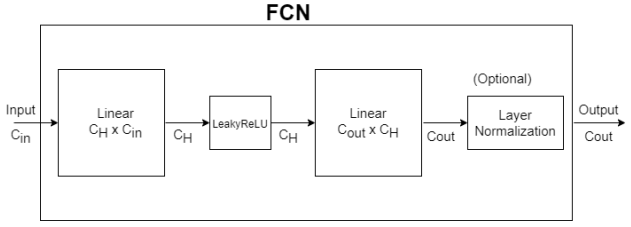

Figure 4: *Fully Connected Network (FCN) with 1 hidden layer using LeakyLeRU activation, optionally with layer normalization layer.*

The total number of particles ranges from 2-5k particles in each scene, with the average of 3610 particles. The scene is then simulated with Taichi Elements [tai] using a frame time step of $\frac{1}{60}s$, which Taichi Elements will use multiple sub steps of MPM simulation to simulate each frame. We run each simulation for 5 seconds, which consists of 300 frames. We additionally generate 5 random scenes for validation and a number of test scenes for generating the accompany videos. We also generate another dataset without obstacles for comparing our method with GNS. It is done in a similar manner to the above, except that obstacles are excluded.

We use a mini batch consisting of all particles of a randomly sampled time step of a randomly chosen scene for training. Using mini batch of 2 or more scenes typically cause GNS [SGGP*20] to run out of memory during training in our 32GB GPUs, therefore, we stick with 1 scene per mini-batch for the purpose of comparison.

We also randomly add noise to perturb input in the same manner as [SGGP*20].

We also randomly jitter the origin of the sparse grid with a random offset chosen within $[0,-R]x[-R,0]x[-R,0]$. This allows the networks to see more variations in particle position relative to grid centers and be invariant to the absolute position of particles.

We use the one step Mean Square Error (MSE) of acceleration computed as $\frac{\sum_i ||\tilde{a}_i^n-(v_i^{n+1}-v_i^n)||^2}{\text{number of particles}}$, as the loss function similar to [SGGP*20]. We however do not need to mask out kinematic particles, as all particles are part of the simulation. Adam optimizer with learning rate decay from 10^{-4} to 10^{-6} over 20M updates is used for training as in [SGGP*20].

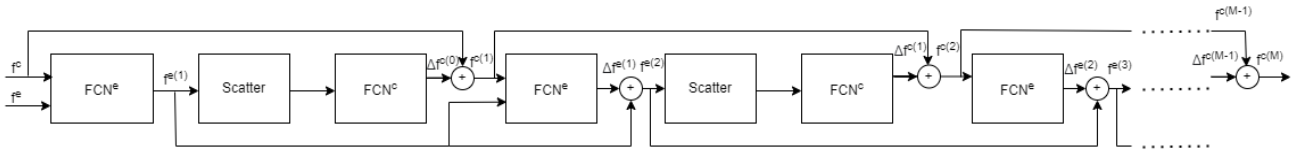

Figure 5: *Message passing steps. M iterations of message passing are done on features on cell center, f^c, and features on edges, f^e. The first step replaces the value of f^e, while other steps update it.*

3.4. Inference

We save the network weights and structure out to binary files and write a CUDA program to evaluate the forward pass. We also evaluate particle features and construct the hierarchy of sparse grid in C++ and CUDA, so that Algorithm 1 runs entirely on the GPU.

4. Results

We use $\Delta x = 1/64 = 0.015625$, the same as the grid spacing for MPM and $R = 0.035$, $E = 16$, $m = 5$ which was used in GNS [SGGP*20]. We use $d_{\max} = 8R$, $M = 4$, $\delta v_{\text{noise}} = 0.000848528$. We use NVIDIA DGX1 machine with 8 V100 GPUs for training in Pytorch and use NVIDIA RTX3090 for inferencing with C++ and CUDA.

We compare our method with GNS for both the training and inference phases. As with our method, we use Pytorch for training and output the network weights to binary files and write a C++ and CUDA program for inferencing. We also use an optimized GPU spatial hash grid [THM*03] to locate particle neighbors and construct the graph. The entire simulation using GNS runs on a GPU with CUDA, so the comparison with our proposed method is fair.

For GNS, we use the connectivity radius of $R = 0.035$, the same as what they use in their paper [SGGP*20] and the same as what we use for the finest sparse grid cell size. We set the distance for normalizing and clamping the distance to wall to $d_{\max} = 8R$ because we first experiment with GNS and tune $d_{\max}$ so that their method produce the smallest error and we did not tune them further when experimenting with our method. $d_{\max}$ needs to be set substantially higher than used in their paper, which used R, because the time step we use for generating MPM data is much larger than theirs, namely 1/60 = 0.01666s vs 0.005 (for water) or 0.0025 (for goop and sand). The number of message passing steps is 10, the same as what they use in their paper. We use a single hidden layer for all fully connected networks appeared in GNS, similar to our FCN. This is because 1 hidden layer yields similar error to 2 hidden layers and runs substantially faster, as also reported in [SGGP*20].

We monitor our network and GNS training by computing the one step MSE of acceleration, measured by taking the average across all frames of the 5 fixed scenes from the training set every 100k iterations. We also compute the error using 5 scenes from the validation set. They are referred to as one-step training and validation errors respectively. We stop training after the validation error plateaus out. This happens before 12M iterations in all experiments. For each experiment, we keep the network that yields the smallest validation error and use them for measuring inference time and for producing the accompanying videos. We also compute the rollout MSEs of the training and validation set, which is the MSE of the particle position across 300 frames of a full roll out of either our method or GNS, every 100k iterations. Throughout this section, training time refers to the average time it takes for each iteration of training measured by running the training on all frames from the 5 fixed training scenes used for measuring the training error. Inferencing time refers to the average time per frame for executing Algorithm 1, including the time for building the graphs or sparse grid, evaluating the network and updating velocity and position, measured over the 5 fixed validation scenes for computing the validation error.

We first compare our method with GNS for the case without obstacles. The dataset in this case is generated without obstacles and the input features of obstacle normals are removed. Figure 6 shows the training error and the validation one-step error. Our network yields lower errors in both cases. Our method also generally yields lower rollout MSE, as shown in Figure 7. On average, the training time and inferencing time per step for our method are 157.84ms and 12.28ms respectively. GNS training and inferencing time are 448.38ms and 52.98ms respectively. Therefore, our method is 2.84 and 4.31 times faster for training and inferencing respectively. Hence, our network is both several times faster and produces lower error. A screenshot of the ground truth, our method and GNS of a frame from a test scene with obstacles is shown in Figure 8 and the accompanying video. One can notice in the accompanying video that the snow deform erroneously for GNS, while our method better preserves the shape like in the ground truth simulation.. The scene is from a test set, not used during training nor validation.

For scenes with obstacles, our method has even a greater advantage compared to GNS as presented in [SGGP*20], as they need to include static particles in their system. As they use 10 message passing steps, one would need to include static particles that are within $10R$ from simulation particles. In our training dataset with an average of 3610 simulation particles, across all scenes and all frames, the average number of static particles needed to be included is 6160, if we were to sample the surface of the obstacle uniformly with particles so that no two particles are closer than $\frac{\Delta x}{4}$. The size of the GNS network was sometimes even too large to fit within 32GB memory. Therefore, we can't directly train their network on our dataset with obstacles. However, we can estimate its running time, as the running time grows approximately linearly with the number of particles. Our method, for the scenes with obstacles, takes 139.22ms for training and 11.01ms testing, as will be discussed in more details in the next paragraph. The 5 fixed training scenes with obstacles have on average 3683 particles and would have 5714 static particles within $10R$ of simulation particles averaged across

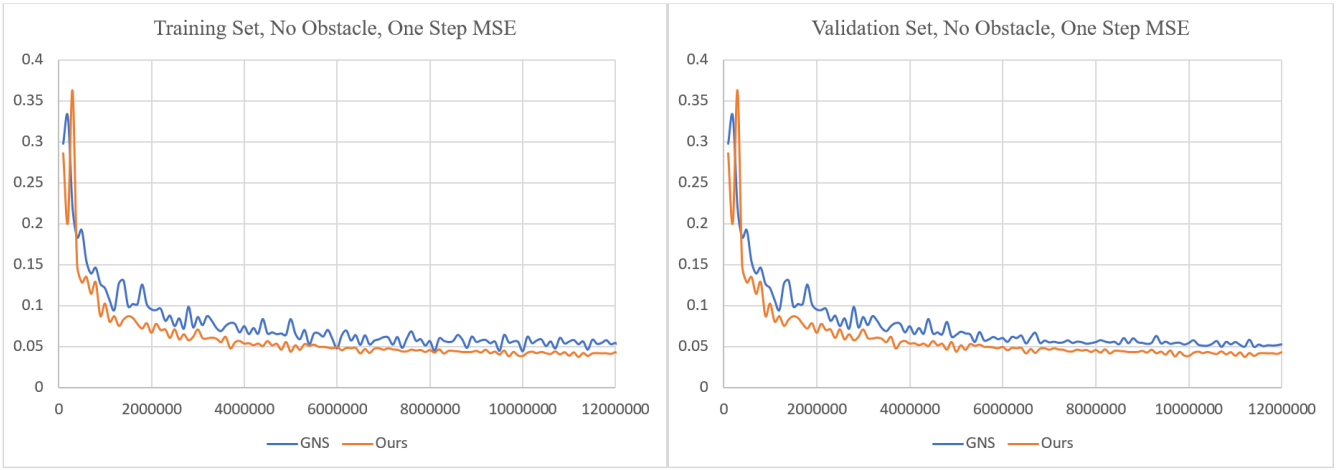

Figure 6: *One step MSE of the acceleration predicted by GNS and our method without obstacle of the training and validation set, averaged over 5 fixed scenes, measured every 100k iterations. Our method yields lower error, as well as being several times faster.*

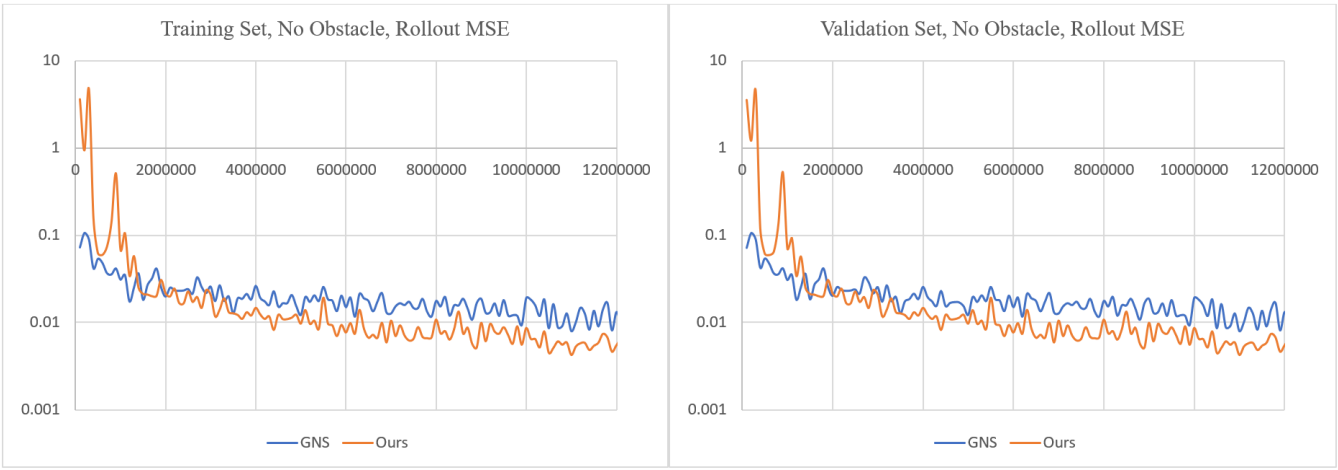

Figure 7: *MSE of the position across 300 time steps over a full rollout of GNS and our method without obstacle of the training and validation set, averaged over 5 fixed scenes, measured every 100k iterations. Our method yields lower error, as well as being several times faster.*

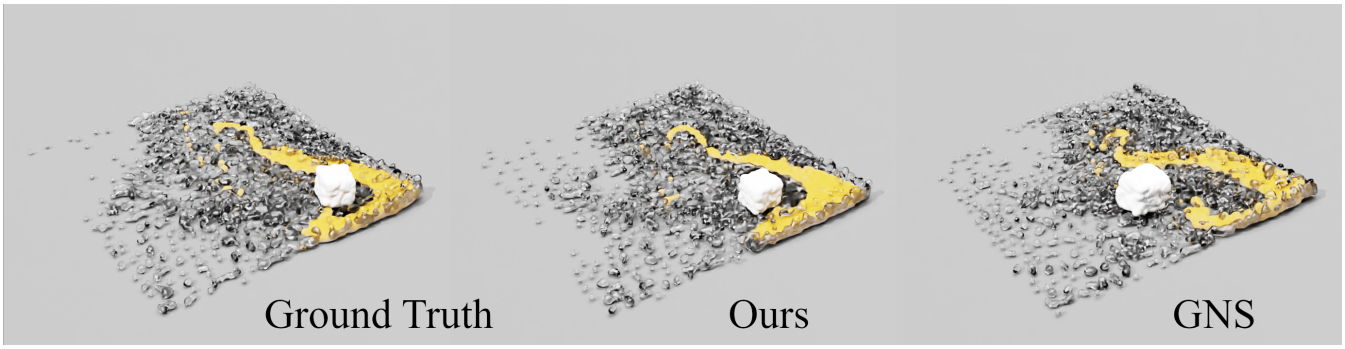

Figure 8: *Ground truth vs. Our method vs. GNS for test scene without obstacle. The scene is not used for training. On average, our method yields less error, train and inference substantially faster than GNS and consume much less memory.*

all frames. The numbers for the 5 valiging scenes with obstacles are 3887 and 6174 respectively. The 5 fixed training scenes without obstacles have on average 4296 particles and the 5 validating scenes without any obstacle also have on average 4296 particles. Therefore, the estimated training time for GNS of [SGGP*20] would be $448.38 \times \frac{3683+5714}{4296} = 980.78ms$ and the estimated inferencing time would be $52.98 \times \frac{3887+6174}{4298} = 124.02ms$. Therefore, our training is expected to be 7.04 times faster and inferencing would be 11.26 times faster compared to [SGGP*20], if the required memory was available.

Now, we focus on analyzing the speedup gain due to the use of a hierarchical network alone. We train GNS using our input representations, which has per particle features for obstacles, but do not require including the obstacle particles in the system. Figure 9 shows the training and validation one step MSE, where our method produces smaller error. Our method, however, yields similar rollout MSE to GNS, as shown in Figure 10. We argue though that the rollout MSE is more noisy and is affected a lot by a hit or miss collision with obstacles, which can change the trajectory of particles significantly. Therefore, we think one-step MSE is more reliable in this case. The training time and inferencing time per iteration for our method is 139.22ms and 11.01ms, while for GNS, they are 412.26ms and 48.56ms. Therefore, our method is 2.96 and 4.41 times faster in training and inferencing respectively. The maximum memory usage for our method during training and inference are 0.93GB and 0.21GB respectively, while GNS use 14.04GB and 1.89GB. Our network is several times faster, produces lower one step MSE and consume much less memory. This demonstrates that our hierarchical graph provides a significant advantage over GNS. A screenshot of ground truth, our method and GNS of a frame from a test scene with obstacles is shown in Figure 1. The scene is from a test set, not used during training nor validation.

The main reasons our method is significantly faster is because of its hierarchical nature and the fewer connections. For our dataset, the average number of particles is 3610, while there are only 1554 finest sparse grid cells. Steps 1 and 11, in Figure 2, are the most expensive steps, but they only involve the data of size 8 times the number of particles (as each particle connects to 8 cells). Step 0 and 12 run independently per particle. The rest of the layers involve the hierarchy grids, with 8 connections to the adjacent level and 6 connections to neighboring cells. For GNS, the data is typically around 103 times the number of particles, as there are around 103 neighbors per particles on average and most computations operate on the connection between neighboring particles.

Taichi Elements has substantial launch overhead regardless of the number of particles. Therefore, to be more fair to them, we compare our inference time to Taichi Elements simulation time by using 10x more particles , which makes the launch overhead smaller relative to the total running time. This is done by placing ten times more initial spheres/cubes on the 5 validing scenes, so the average number of particles per scene is now 39752 particles. On average, our method takes 121.01ms per step for inferencing, while Taichi Elements uses 449.83ms per step, so in this case, our method is 3.72 times faster than Taichi Elements.

We also experiment with using a smaller network which uses $M = 2$ denoted as Ours (small) in Figure 1,9,10 and the accompanying videos. It produces similar, or only slightly worse, one step MSE and rollout MSE to GNS. The training time per iteration is 98.72ms and the inference time is 8.11ms per frame, which makes the small network about 1.41 and 1.35 times faster than our default network for training and inferencing respectively. This demonstrates the possibility of having the ability to trade-off accuracy with speed of DL.

We also run our network on scenes with more particles than used during training as shown in Figure 11 and also on much larger scenes in Figure 12 and the accompanying video. Non-zero obstacle normal of cells are stored in a spatial hash table [THM*03], indexed by the integer coordinate of the cell, so we never need a dense grid. Our method is able to produce plausible results in these challenging cases, even though they are quite different from the training set.

5. Discussion

While our method yields lower error, runs significantly faster and consumes less memory compared to GNS, it still has some drawbacks. First, our experiment currently consists of 3 materials, water, sand and snow with fixed parameters. It will be interesting for future work to experiment with varying material parameters for each of these materials and include them in the input too. Second our method, like GNS, when applied to learn physics of perfectly elastic material, the object will still lose its rest shape slowly over time. This is because it does't have material space information. [PFSGB20] addressed this issue with material space connections. However, we focus on the multi-material case in this work and the material space connection is almost meaningless for water and sand. More work will be required to incorporate this into the hierarchical graph and make it work well with elastic materials and other types of materials at the same time. Third, while our network consumes less memory compared to GNS, it still requires more memory than a well coded explicit MPM solver, due to the large dimensionality of features. It may be interesting in a future work to experiment with using lower precision floating point representations and quantization to reduce this requirement. Forth, there is currently no guarantees that the network will never produce ghost acceleration. Ghost acceleration can be clearly observed during early stage of training of large network and also for converged smaller networks that have lower capacity. A novel loss function that considers this may be required to completely eliminate this issue, and is an interesting venue for future work.

In conclusion, we propose a new DL method for learning 3D multi-material MPM with obstacles. This is achieved by handling obstacles without ghost particles and a hierarchical graph. This results in lower error, faster training, faster inferencing and smaller memory footprint compared to GNS [PFSGB20]. We also demonstrate the ability to achieve a trade-off between speed and accuracy with DL.

References

[BHB*18] BATTAGLIA P. W., HAMRICK J. B., BAPST V., SANCHEZ-GONZALEZ A., ZAMBALDI V. F., MALINOWSKI M., TACCHETTI A., RAPOSO D., SANTORO A., FAULKNER R., ÇAGLAR GÜLÇEHRE, SONG H. F., BALLARD A. J., GILMER J., DAHL G. E., VASWANI A.,

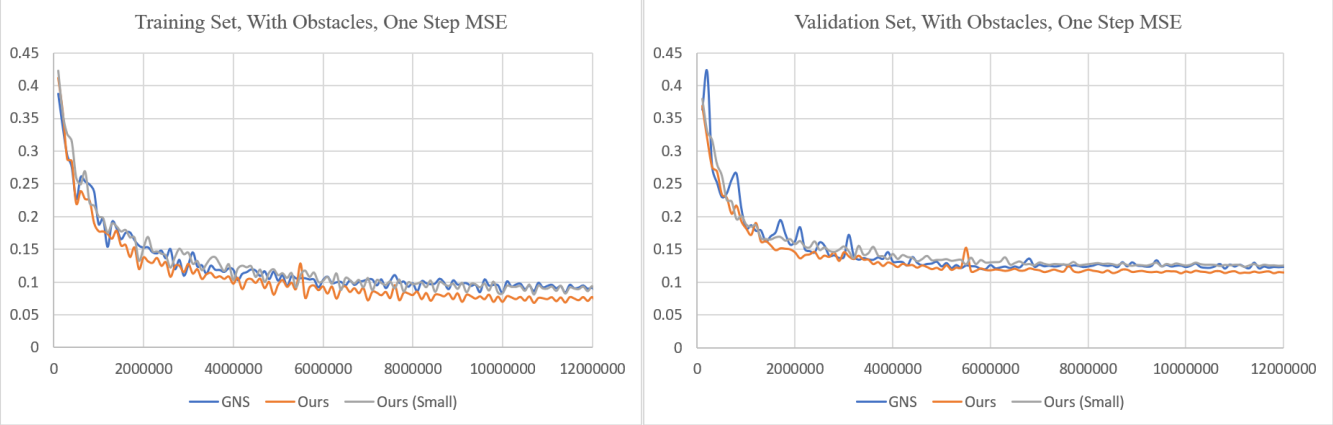

Figure 9: *One step MSE of the acceleration predicted by GNS, our method and our method using smaller network with obstacles of the training and validation set, averaged over 5 fixed scenes, measured every 100k iterations. Our method yields lower error, as well as being several times faster than GNS. Our method using smaller network has similar error to GNS and run even faster.*

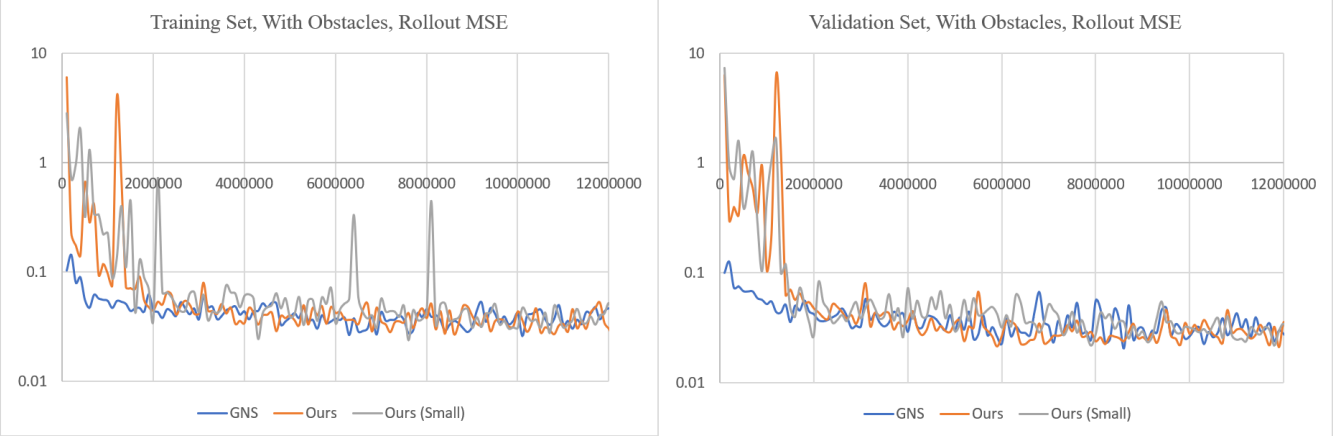

Figure 10: *MSE of the position across 300 time steps over a full rollout of GNS, our method and our method using smaller network with obstacles of the training and validation set, averaged over 5 fixed scenes, measured every 100k iterations. The errors are somewhat similar. We argue though that rollout MSE is quite noisy and is affected a lot by collision against obstacles, so it may not be as reliable as one-step acceleration MSE in this case.*

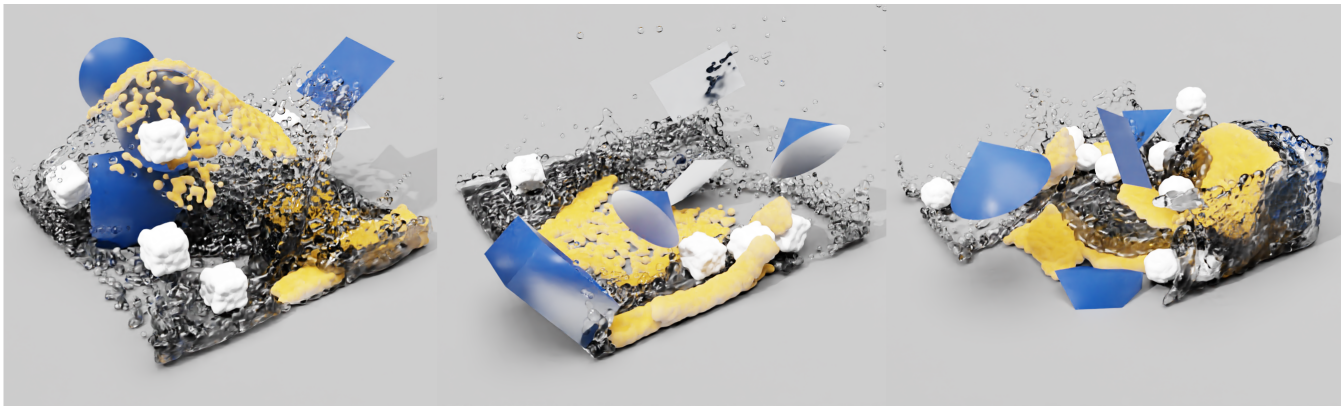

Figure 11: *Our method is used for simulating various scenes with more particles than those used during training.*

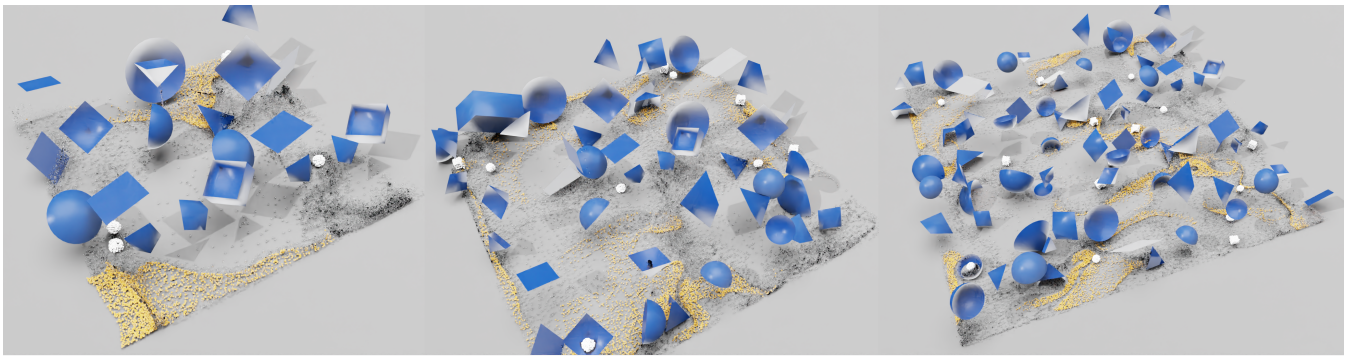

Figure 12: *Our method is used for simulating various scenes with more particles and 4x, 9x and 16x bigger domain size than those used during training.*

ALLEN K. R., NASH C., LANGSTON V., DYER C., HEESS N. M. O., WIERSTRA D., KOHLI P., BOTVINICK M. M., VINYALS O., LI Y., PASCANU R.: Relational inductive biases, deep learning, and graph networks. *ArXiv abs/1806.01261* (2018). 3

[CCGC21] CHEN P. Y., CHIARAMONTE M., GRINSPUN E., CARLBERG K.: Model reduction for the material point method via an implicit neural representation of the deformation map, 2021. URL: https://arxiv.org/abs/2109.12390, doi:10.48550/ARXIV.2109.12390. 2

[CKMR*21] CHEN J., KALA V., MARQUEZ-RAZON A., GUEIDON E., HYDE D. A. B., TERAN J.: A momentum-conserving implicit material point method for surface tension with contact angles and spatial gradients. *ACM Trans. Graph. 40*, 4 (jul 2021). URL: https://doi.org/10.1145/3450626.3459874, doi:10.1145/3450626.3459874. 2

[FGW*21] FEI Y. R., GUO Q., WU R., HUANG L., GAO M.: Revisiting integration in the material point method: A scheme for easier separation and less dissipation. *ACM Trans. Graph. 40*, 4 (jul 2021). URL: https://doi.org/10.1145/3450626.3459678, doi:10.1145/3450626.3459678. 2

[HAL*20] HU Y., ANDERSON L., LI T.-M., SUN Q., CARR N., RAGAN-KELLEY J., DURAND F.: Difftaichi: Differentiable programming for physical simulation. *ICLR* (2020). 2

[HFG*18] HU Y., FANG Y., GE Z., QU Z., ZHU Y., PRADHANA A., JIANG C.: A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Trans. Graph. 37*, 4 (jul 2018). URL: https://doi.org/10.1145/3197517.3201293, doi:10.1145/3197517.3201293. 2

[HLA*19] HU Y., LI T.-M., ANDERSON L., RAGAN-KELLEY J., DURAND F.: Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG) 38*, 6 (2019), 201. 2

[HLY*21] HU Y., LIU J., YANG X., XU M., KUANG Y., XU W., DAI Q., FREEMAN W. T., DURAND F.: Quantaichi: A compiler for quantized simulations. *ACM Transactions on Graphics (TOG) 40*, 4 (2021). 2

[JSS*15] JIANG C., SCHROEDER C., SELLE A., TERAN J., STOMAKHIN A.: The affine particle-in-cell method. *ACM Trans. Graph. 34*, 4 (jul 2015). URL: https://doi.org/10.1145/2766996, doi:10.1145/2766996. 2

[JST*16] JIANG C., SCHROEDER C., TERAN J., STOMAKHIN A., SELLE A.: The material point method for simulating continuum materials. In *ACM SIGGRAPH 2016 Courses* (New York, NY, USA, 2016), SIGGRAPH '16, Association for Computing Machinery. URL: https://doi.org/10.1145/2897826.2927348, doi:10.1145/2897826.2927348. 2

[KHT22] KLIMESCH J., HOLL P., THUEREY N.: Simulating liquids with graph networks. *CoRR abs/2203.07895* (2022). URL: https://doi.org/10.48550/arXiv.2203.07895, arXiv:2203.07895, doi:10.48550/arXiv.2203.07895. 2

[LCBF21] LINO M., CANTWELL C., BHARATH A. A., FOTIADIS S.: Simulating continuum mechanics with multi-scale graph neural networks, 2021. URL: https://arxiv.org/abs/2106.04900, doi:10.48550/ARXIV.2106.04900. 2

[LF22] LI Z., FARIMANI A. B.: Graph neural network-accelerated lagrangian fluid simulation. *Computers Graphics 103* (2022), 201–211. URL: https://www.sciencedirect.com/science/article/pii/S0097849322000206, doi:https://doi.org/10.1016/j.cag.2022.02.004. 2

[LMYBF22] LI Z., MEIDANI K., YADAV P., BARATI FARIMANI A.: Graph neural networks accelerated molecular dynamics. *The Journal of Chemical Physics 156*, 14 (2022), 144103. URL: https://doi.org/10.1063/5.0083060, arXiv:https://doi.org/10.1063/5.0083060, doi:10.1063/5.0083060. 2

[LWT*18] LI Y., WU J., TEDRAKE R., TENENBAUM J. B., TORRALBA A.: Learning particle dynamics for manipulating rigid bodies, deformable objects, and fluids, 2018. URL: https://arxiv.org/abs/1810.01566, doi:10.48550/ARXIV.1810.01566. 2

[MLM*21] MAYR A., LEHNER S., MAYRHOFER A., KLOSS C., HOCHREITER S., BRANDSTETTER J.: Boundary graph neural networks for 3d simulations, 2021. URL: https://arxiv.org/abs/2106.11299, doi:10.48550/ARXIV.2106.11299. 2

[MLP20] MARTINKUS K., LUCCHI A., PERRAUDIN N.: Scalable graph networks for particle simulations, 2020. URL: https://arxiv.org/abs/2010.06948, doi:10.48550/ARXIV.2010.06948. 2

[OHC22] ODOT A., HAFERSSAS R., COTIN S.: Deepphysics: A physics aware deep learning framework for real-time simulation. *International Journal for Numerical Methods in Engineering 123* (2022), 2381 – 2398. 2

[PFSGB20] PFAFF T., FORTUNATO M., SANCHEZ-GONZALEZ A., BATTAGLIA P. W.: Learning mesh-based simulation with graph networks. URL: https://arxiv.org/abs/2010.03409, doi:10.48550/ARXIV.2010.03409. 1, 2, 7

[PLL21] PARK J., LEE D., LEE I.-K.: Flexible networks for learning physical dynamics of deformable objects, 2021. URL: https://arxiv.org/abs/2112.03728, doi:10.48550/ARXIV.2112.03728. 2

[SGGP*20] SANCHEZ-GONZALEZ A., GODWIN J., PFAFF T., YING R., LESKOVEC J., BATTAGLIA P.: Learning to simulate complex

physics with graph networks. In *Proceedings of the 37th International Conference on Machine Learning* (13–18 Jul 2020), III H. D., Singh A., (Eds.), vol. 119 of *Proceedings of Machine Learning Research*, PMLR, pp. 8459–8468. URL: https://proceedings.mlr.press/v119/sanchez-gonzalez20a.html. 1, 2, 3, 4, 5, 7

[SSC*13] STOMAKHIN A., SCHROEDER C., CHAI L., TERAN J., SELLE A.: A material point method for snow simulation. *ACM Trans. Graph. 32*, 4 (jul 2013). URL: https://doi.org/10.1145/2461912.2461948, doi:10.1145/2461912.2461948. 2

[SXH*21] SU H., XUE T., HAN C., JIANG C., AANJANEYA M.: A unified second-order accurate in time mpm formulation for simulating viscoelastic liquids with phase change. *ACM Trans. Graph. 40*, 4 (Aug. 2021). 2

[SXHA21] SU H., XUE T., HAN C., AANJANEYA M.: A-ulmpm: An arbitrary updated lagrangian material point method for efficient simulation of solids and fluids, 2021. URL: https://arxiv.org/abs/2108.00388, doi:10.48550/ARXIV.2108.00388. 2

[SZS95] SULSKY D., ZHOU S.-J., SCHREYER H. L.: Application of a particle-in-cell method to solid mechanics. *Computer Physics Communications 87*, 1 (1995), 236–252. Particle Simulation Methods. URL: https://www.sciencedirect.com/science/article/pii/0010465594001707, doi:https://doi.org/10.1016/0010-4655(94)00170-7. 2

[tai] Taichi elements. https://github.com/taichi-dev/taichi_elements. 2, 4

[TBMK22] TUOMAINEN N., BLANCO-MULERO D., KYRKI V.: Manipulation of granular materials by learning particle interactions. *IEEE Robotics and Automation Letters 7*, 2 (2022), 5663–5670. doi:10.1109/LRA.2022.3158382. 2

[THM*03] TESCHNER M., HEIDELBERGER B., MÜLLER M., POMERANTES D., GROSS M. H.: Optimized spatial hashing for collision detection of deformable objects. In *VMV* (2003). 5, 7

[TKC21] TUMANOV E., KOROBCHENKO D., CHENTANEZ N.: Data-driven particle-based liquid simulation with deep learning utilizing sub-pixel convolution. *Proc. ACM Comput. Graph. Interact. Tech. 4*, 1 (apr 2021). URL: https://doi.org/10.1145/3451261, doi:10.1145/3451261. 1, 2

[UPTK20] UMMENHOFER B., PRANTL L., THUEREY N., KOLTUN V.: Lagrangian fluid simulation with continuous convolutions. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020* (2020), OpenReview.net. URL: https://openreview.net/forum?id=B1lDoJSYDH. 2

[WCL*20] WOLPER J., CHEN Y., LI M., FANG Y., QU Z., LU J., CHENG M., JIANG C.: Anisompm: Animating anisotropic damage mechanics. *ACM Trans. Graph. 39*, 4 (jul 2020). URL: https://doi.org/10.1145/3386569.3392428, doi:10.1145/3386569.3392428. 2

[WQS*20] WANG X., QIU Y., SLATTERY S. R., FANG Y., LI M., ZHU S.-C., ZHU Y., TANG M., MANOCHA D., JIANG C.: A massively parallel and scalable multi-gpu material point method. *ACM Trans. Graph. 39*, 4 (jul 2020). URL: https://doi.org/10.1145/3386569.3392442, doi:10.1145/3386569.3392442. 2

[WY21] WANG R., YU R.: Physics-guided deep learning for dynamical systems: A survey, 2021. URL: https://arxiv.org/abs/2107.01272, doi:10.48550/ARXIV.2107.01272. 2

[XAM22] XUE T., ADRIAENSSENS S., MAO S.: Learning the nonlinear dynamics of soft mechanical metamaterials with graph networks. *CoRR abs/2202.13775* (2022). URL: https://arxiv.org/abs/2202.13775, arXiv:2202.13775. 2