Efficient Particle Simulation using Graph Networks

Alexandre Sirois-Vigneux
alexandre.sirois-vigneux@mila.quebec
McGill University
Montréal, Canada

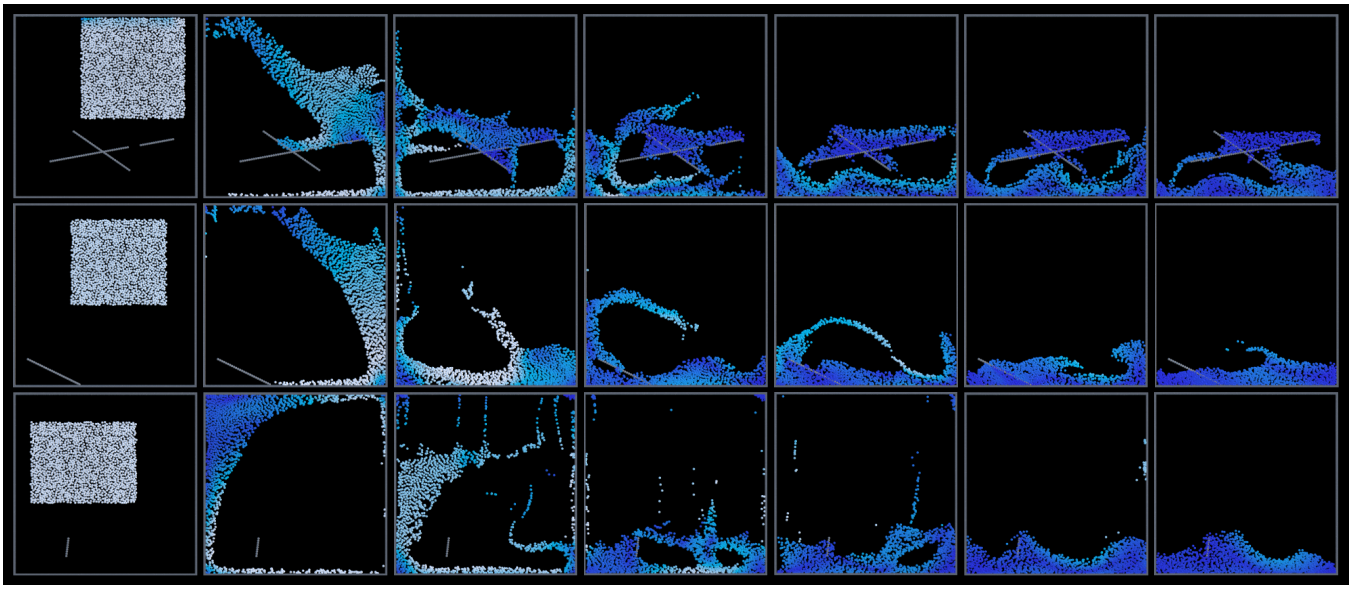

Figure 1: Shows 3 simulations, one on each row, decomposed in 7 frames equally sampled every 100 frames across the whole frame range. Those simulations were generated using our simplified GNS model provided a single state of a particle system (containing position, velocity, and material type attributes) as initial conditions. The simulation is generated by feeding back each predicted frame as input to the model in order to generate an animated sequence of arbitrary length. The active particles are colored based on their speed while the inactive (non-kinematic) particles and the domain boundaries are colored in gray.

ABSTRACT

In this work, we revisit a recently proposed architecture called Graph Network-based Simulators (GNS) which is able to accurately simulate many types of materials. Instead of trying to make the model generalize to many tasks, we attempt to simplify it and evaluate how different design configurations can impact both training and inference time. We will compare the performance of our simplified model against the original GNS to measure the impact of changes in the design space of this architecture. Apart from using common metrics such as MSE to evaluate the accuracy, we will also consider perceptual believability as a subjective metric to evaluate the quality of our results.

KEYWORDS

particle simulation, data-driven, graph networks

1 INTRODUCTION

Being able to accurately simulate physics is an essential step toward understanding our environment. Many disciplines such as physics, engineering, and even entertainment have tackled this problem with different goals in mind. In physics and engineering, the goal is often to predict how a system will behave in real life under certain conditions. The level of accuracy required for this kind of task is therefore very high. On the other hand, in entertainment, we try to create believable worlds where players or spectators can experience physical phenomena that look plausible regardless of their actual physical accuracy. Although those two objectives are strongly correlated they are fundamentally different. There are reasons to believe that perceptual believability (when things look real to us) is computationally simpler to satisfy than physical accuracy [7] [20] [14]. Although most available metrics such as MSE and Earth Mover's Distance [19] cannot be trusted as good approximators of perceptual believability, we hypothesize that some models could learn to generate simulations that look real without having to pay the price of physical accuracy.

Many traditional solvers have been proposed to simulate particle physics in recent years [11] [17] [12]. The most flexible solutions available today require complex data structures and very careful

engineering in order to produce visually interesting results. It is not uncommon for those solvers to take many years to write and those are often hard to maintain due to the complexity of their highly optimized code-base.

Since 2017, Graph Neural Networks (GNNs), or more generally Graph Networks (GNs), have seen tremendous progress in terms of their ability to model and predict complex relations between multiple types of entities [3]. The graph naturally appears to be a good structure to model particle-particle interactions. Some recent contributions have shown it is even possible to learn the underlying physics behind particle solvers using GNs [15] [16] [2]. However, it is unclear how design configurations influence GN's ability to learn to solve physical systems.

Many authors have been focusing on the generalization power of those models, so they can solve many materials in different contexts [15]. The idea of having a single architecture solving multiple problems comes at a non-negligible cost in terms of capacity requirements, model complexity, and ultimately training and inference computation time. In this work, we will study the scalability and efficiency of GNs applied to particle simulation. In our case, this scalability could translate into bigger simulations in terms of particle count, larger integration time steps, or faster training and inference time. We will also investigate how the latent representation of the graph could be further compressed in order to more efficiently utilize the limited memory of modern GPUs.

More generally, the proposed simplifications should be applicable to vertex regression tasks on other datasets and most findings in the context of particle-particle interactions could transfer to other research fields outside of physics. Studying how vertex features evolves over time with respect to their local graph structure could lead to powerful models in social science to predict how beliefs, personality, interests, and moral values propagate or diffuse in dynamic social networks.

2 RELATED WORK

Multiple techniques to simulate particle-particle interactions have emerged in the last 3 decades. In 1992 Smoothed Particle Hydrodynamics (SPH) [11] is one of the first algorithms to successfully solve complex 3-dimensional particle physics problems with ease. This is then followed in 1995 by a much more general approach named Material Point Method (MPM) [17] which is an extension to solid mechanics of the Fluid-Implicit-Particle particle-in-cell method. This type of solver unfortunately requires a very small integration time step and heavy data structures which makes it very slow and less practical considering computational resources at the time. In 2007 Muller et al. propose a new technique called Position-based dynamics (PBD) [12] that directly manipulates the vertex positions of object meshes instead of forces or impulses. This approach is taken a step further in 2014 when Nvidia built Flex [10]. Their framework is using particles connected by constraints as their fundamental building blocks to treat contact and collisions in a unified manner to model gases, liquids, deformable solids, rigid bodies, and cloth with two-way interactions at 60 fps.

In 2015, a data-driven solution that formulates physics-based fluid simulation as a regression problem estimating the acceleration of every particle is proposed for the first time [9]. This technique is able to outperform state-of-the-art position-based fluid solver at one to three orders of magnitude using regression forest. From this point going forward, many contributions were made in applying machine learning to physic problems.

A year later Interaction Networks are introduced and tested in many physical domains including n-body problems, rigid-body collision, and non-rigid dynamics [3]. Their model can learn to accurately simulate the physical trajectories of dozens of objects over thousands of time steps. The year after, Gilmer et al. reformulated existing GNN models into a single common framework they called Message Passing Neural Networks (MPNNs) that could beat state of the art results on multiple tasks [6]. In 2018, Graph Networks (GNs) were introduced as a new building block with a strong relational inductive bias that generalizes and extends various approaches for neural networks [2]. The same year, Sanchez et al. presented a new class of learnable models based on GNs which implement an inductive bias for object- and relation-centric representations of dynamical systems [16] that pushed the state of the art one step further. In 2019, continuous convolution is presented as a way to learn to efficiently simulate Lagrangian fluid [18]. Although their technique used an extension of point cloud convolution to the continuous domain instead of a graph, their approach could still be reformulated in terms of a GN where the depth of message passing is limited to one hop in the predefined local neighborhood of each particle. In the same year, DeepMind proposed a very general approach to simulate many complex particle interactions based on GNs and message passing [15]. Their technique could accurately predict simulations over long horizons while closely matching dynamic behavior captured with SPH, PBD, and MPM.

3 PROBLEM DEFINITION

A particle solver tries to predict the acceleration of each particle at the current time step k based on its physical property and the physical properties of all objects contained in its local neighborhood such as other particles and colliders. The predicted acceleration can then be used to move the particle forward in time by integrating the acceleration to predict the next velocity and integrating the updated velocity to get the next position.

This problem can easily be modeled as a graph where potential particle-particle interactions within a predefined radius are encoded as edges. In the context of a GN, this is a vertex regression task where we train a GN to predict particle acceleration through message passing on the underlying constructed graph.

For consitency with the literature we will adopt the notation used by Sanchez et al. [15]. We denote the state of the particle simulation at time t with $X^t \in \mathcal{X}$ such that the whole simulation frame range is denoted with $\mathbf{X}^{t_{0:K}} = (X^{t_0}, \ldots, X^{t_K})$. A typical simulator $s : \mathcal{X} \rightarrow \mathcal{X}$ can generate a trajectory $\tilde{\mathbf{X}}^{t_{0:K}} = (X^{t_0}, \tilde{X}^{t_1}, \ldots, \tilde{X}^{t_K})$ from an initial state X^{t_0} such that $\tilde{X}^{t_{k+1}} = s(\tilde{X}^{t_k})$. On the other hand, a learned simulator $d_\theta : \mathcal{X} \rightarrow \mathcal{Y}$ will have his parameters θ optimized during training to accuralty predict current acceleration $Y \in \mathcal{Y}$. This acceleration will update position using semi-implicit Euler integration such that $\tilde{X}^{t_{k+1}} = Integrate(\tilde{X}^{t_k}, d_\theta)$.

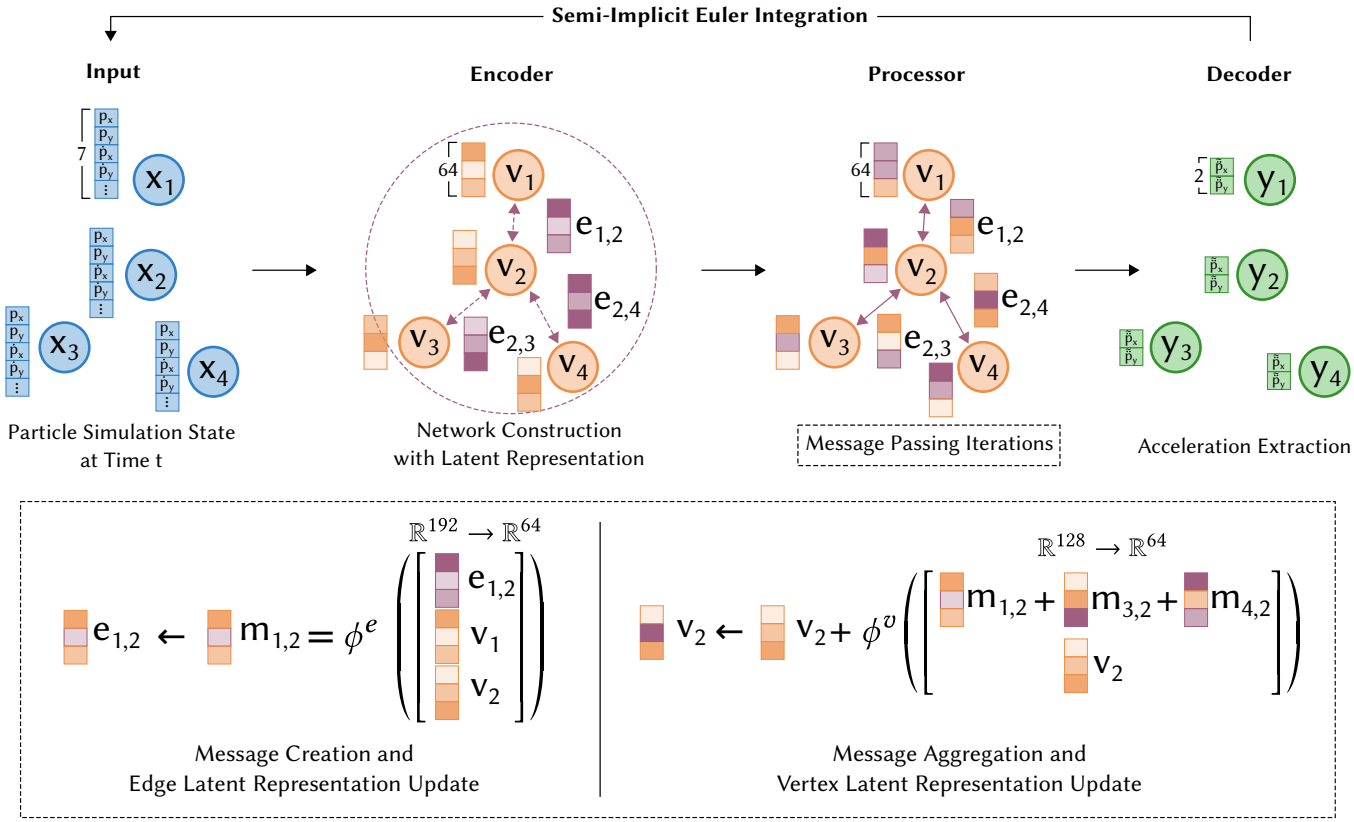

Figure 2: Graphic overview of the GNS model. The particle system is fed as input to the Encoder which constructs a graph from it. The Processor performs multiple iterations of message passing on the graph to update both the edge and vertex latent representations. The Decoder then uses the final latent representation of the vertices to predict the per-particle normalized acceleration. These accelerations can be used to train the model or to step the particle system forward in time so it can be fed back to the model to generate a full sequence of animation. Note that the dimensions presented in this figure assume the particle dataset to be 2D and that the embeddings of size 64 were empirically determined based on hyperparameter grid searches, see section 9 for more information.

4 MAIN CONTRIBUTIONS

In this work, we propose a new set of hyperparameters to simplify the architecture of the original GNS model. These changes will be based on their impact on both the speed and perceptual believability of the model. Furthermore, we provide a detailed exploration of the design space where sensitive hyperparameters are thoroughly discussed in section 9.

5 DATASET DESCRIPTION

To avoid having to generate our own dataset, we will use the **WaterRamps** simulations provided by Sanchez et al. [15]. This dataset contains the usual train, valid, and test sets containing respectively 1000, 100, and 100 simulations. Each simulation contains 601 frames and a maximum of ±2300 particles per frame. The graph is implicitly contained in the simulation data where particles or converted to vertices and the proximity between vertices is translated into edges in the graph representation. Since particles are moving in space their local neighborhood is changing over time and so is their respective edge set. The explicit dynamic graph is therefore constructed for each frame using nearest neighbor algorithms [4].

Some of those simulations contain ramps of static particles to act as obstacles to the dynamic particles. The data were simulated using the Taichi-MPM engine [8] with a timestep of 2.5 ms. The data contains per particle position and material type information as well as global simulation statistics computed on the training set such as element-wise mean and standard deviation for both the velocity and acceleration. Although the particles only contain positional information we can easily derive both the velocity and acceleration from the position using finite differences.

6 METHODOLOGY

This section outlines the core architectural decisions regarding our simplified GNS model from a mathematical standpoint as well as an overview of the key components of its implementation using PyG [5].

6.1 Model

Our model d_θ is composed of three parts, the $Encoder : \mathcal{X} \rightarrow \mathcal{G}$, the $Processor : \mathcal{G} \rightarrow \mathcal{G}$ and the $Decoder : \mathcal{G} \rightarrow \mathcal{Y}$. Our model is a simplified version of the work from Sanchez et al. [15] which follows the general achitecture of the Interaction Network [3] formulated as a Graph Network [2]. To simplify the notation, the model is presented in a 2D context, but can easily be extended to 3D datasets. To help the model at generalizing, both the velocity $\dot{\mathbf{p}}$ and acceleration $\ddot{\mathbf{p}}$ are normalized using statistics precomputed on the training set such that

$$\hat{\dot{\mathbf{p}}} = \frac{\dot{\mathbf{p}} - \mu_{\dot{\mathbf{p}}}}{\sigma_{\dot{\mathbf{p}}}} \qquad \hat{\ddot{\mathbf{p}}} = \frac{\ddot{\mathbf{p}} - \mu_{\ddot{\mathbf{p}}}}{\sigma_{\ddot{\mathbf{p}}}}. \tag{1}$$

6.1.1 **The Encoder.** This part of the network builds a graph representation $G = (V, E)$ from the physical state of a particle system $X = (\mathbf{x}_0, \ldots, \mathbf{x}_N)$ where each $\mathbf{x}_i = [\mathbf{p}_i, \dot{\mathbf{p}}_i, \ddot{\mathbf{p}}_i, m_i]$ is a vector containing the flatten particle's position, velocity, acceleration and material type. Since we want to predict $\ddot{\mathbf{p}}$, some transformations are applied to the particle representation such that $\mathbf{x'}_i = [\hat{\dot{\mathbf{p}}}_i, \hat{\mathbf{b}}_i, m_i]$, will be used instead of $\mathbf{x}_i$. The term $\hat{\mathbf{b}}_i$ is the distance of each particle from the domain's boundaries normalized in the $[0, 1]$ range by the search radius used to build the graph. When particles fall within this search radius from each other they are considered to potentially interact. The relation between particles is defined as $\mathbf{r}_{i,j}$ and contains the normalized vector distance between them as well as the normalized eucleadian distance such that $\mathbf{r}_{i,j} = [(\widehat{\mathbf{p}_i - \mathbf{p}_j}), ||\widehat{\mathbf{p}_i - \mathbf{p}_j}||]$.

The particles get converted to graph vertices $\mathbf{v}_i = \epsilon^v(\mathbf{x'}_i)$ and the relations between particles are converted to graph edges $\mathbf{e}_{i,j} = \epsilon^e(\mathbf{r}_{i,j})$ where $\epsilon^v : \mathbb{R}^7 \rightarrow \mathbb{R}^{64}$ and $\epsilon^e : \mathbb{R}^3 \rightarrow \mathbb{R}^{64}$ are MultiLayer Perceptrons (MLPs) that respectively project the particle attributes $\mathbf{x'}_i$ and the particle relations $\mathbf{r}_{i,j}$ to an embedding space of size 64. We therefore have $G^0 = Encoder(X)$.

As an inductive bias, see figure 3, positional information is not stored in $\mathbf{x'}_i$ to make sure the model stays spatially invariant. The normalization of the particle-particle and particle-boundary distances should help the model to generalize to different scene scales. As opposed to what has been done in Sanchez et al. [15], we try to reduce the memory footprint of the particle feature vector $\mathbf{x'}_i$ by only storing the current velocity instead of a sliding window containing the last 5 velocities. We also store the material type as a single scalar instead of using an embedding of size 16. Those simplifications allow us to compress the latent representation of the vertices and edges from 128 to 64 sized vectors which should accelerate both training and inference as well as make the model more scalable.

6.1.2 **The Processor.** From the initial graph constructed by the Encoder, we want the vertices to gather information about their local neighborhood to infer what the next state of the particle system should be. This is achieved through Message Passing (MP) [6] where both the vertices and edges representation will be updated by aggregating information using adjacent connections. Those updates are performed iteratively M times to progressively increase the receptive field of the graph convolution so each vertex can capture a more accurate representation of its surroundings such that $G^{m+1} = MP^{m+1}(G^m)$. Figure 2 provides a more visual interpretation of this process.

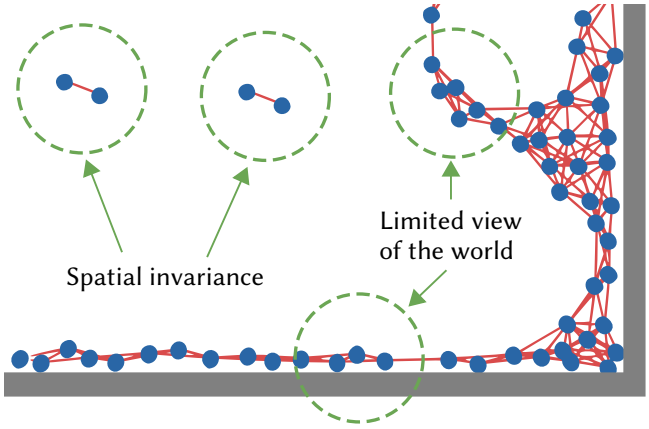

Figure 3: Inductive biases in the context of particle simulation. Multiple inductive biases are imposed on our data in the form of transformations to prevent our model from using correlations between variables that are fundamentally independent of each other. The search radius is used to define the limited view that each particle has on the overall simulation domain. The absolute position of the particles is not provided to our model since we want inference to be spatially invariant based on the theory from classical mechanics.

The first step is to construct the message $\mathbf{m}_{i,j}$ that will be propagated through the network. This message also happends to be the new latent representation of each edges for the next iteration such that

$$\mathbf{e}_{i,j} \leftarrow \mathbf{m}_{i,j} = \phi^e([\mathbf{e}_{i,j}, \mathbf{v}_i, \mathbf{v}_j]) \tag{2}$$

where $\phi^e : \mathbb{R}^{192} \rightarrow \mathbb{R}^{64}$ is a shallow MLP. This message is then aggregated for each vertex through summation which gives us the vertex update formulation

$$\mathbf{v}_i \leftarrow \mathbf{v}_i + \phi^v\left(\left[\sum_{\mathbf{v}_j \in \mathcal{N}(\mathbf{v}_i)} \mathbf{m}_{i,j}, \mathbf{v}_i\right]\right) \tag{3}$$

where $\phi^v : \mathbb{R}^{128} \rightarrow \mathbb{R}^{64}$ is also a shallow MLP. Note that, as a self connection, $\mathbf{v}_i$ is concatenated with the result of the aggregated messages before being passed to ϕ^v. A skip connection is also added to the result of the vertex latent representation update. This iterative process can then be summarized with $G^M = Processor(G^0)$ where $M = 5$ in our final achitecture.

6.1.3 **The Decoder.** After the message passing iterations, each vertex representation should contain meaningful informations about the physical state of the simulation in the local neighborhood of the particle it represents. The predicted normalized acceleration $\tilde{\ddot{\mathbf{p}}}$ is then extracted using $\mathbf{y}_i = \delta^v(\mathbf{v}_i^M)$ where $\delta^v : \mathbb{R}^{64} \rightarrow \mathbb{R}^2$ is yet another shallow MLP.

6.1.4 **Integrator.** At evaluation time, it is possible to generate a full animation sequence using a feedback loop based on a single frame X^{t_0} as the initial conditions. The predicted normalized

acceleration is first de-normalized with

$$\ddot{\mathbf{p}} = \tilde{\ddot{\mathbf{p}}}\sigma_{\ddot{\mathbf{p}}} + \mu_{\ddot{\mathbf{p}}} \tag{4}$$

where $\sigma_{\ddot{\mathbf{p}}}$ and $\mu_{\ddot{\mathbf{p}}}$ are respectivelly the standard deviation and mean of the training set acceleration. Then, the acceleration is integrated to update the velocity which is then itself integrated to update the position with

$$\begin{aligned} \dot{\mathbf{p}}^{k+1} &= \dot{\mathbf{p}}^{k} + \ddot{\mathbf{p}}^{k}\Delta t \\ \mathbf{p}^{k+1} &= \mathbf{p}^{k} + \dot{\mathbf{p}}^{k+1}\Delta t \end{aligned} \tag{5}$$

where, in our case, $\Delta t = 1$ and can therefore safely be ignored. Thus, we get the next state of the particle system at time $k+1$ using semi-implicit Euler integration such that $\tilde{X}^{t_{k+1}} = Integrate(\tilde{X}^{t_k}, d_\theta)$. This new state can directly be used as input to the Encoder to continue the feedback loop until the last frame of the sequence is generated.

6.1.5 **MLP Modules.** All MLPs of our simplified GNS model count 2 hidden layers of size 64 with only the first one being activated with a nonlinear ReLU activation function [13]. Apart from the Decoder, all MLPs are followed by layer normalization [1] to stabilize training.

6.2 Implementation Details

One goal of this project is to produce a simple and easy-to-understand implementation based on a single modern deep learning framework. At the time of this writing, PyG seems to be the best option to keep the dependencies to a minimum while leveraging the latest GPU optimizations offered by this fairly recent library [5].

In order to reduce computation as training is performed, we precompute the velocity and acceleration of each particle system X^{t_k} for each k using finite differences such that a single frame X^{t_k} contains all the information needed to perform training independently from the frames before and after.

$$\begin{aligned} \dot{p}^{k} &= \frac{p^{k} - p^{k-1}}{\Delta t} \\ \ddot{p}^{k} &= \frac{\dot{p}^{k+1} - \dot{p}^{k}}{\Delta t} = \frac{p^{k+1} - 2p^{k} + p^{k-1}}{\Delta t^2}. \end{aligned} \tag{6}$$

Note that our model considers $\Delta t = 1$ to slightly simplify things. To keep the training data footprint small both on disk and in memory, we compute many additional quantities on the fly right before embeddings the particle features and relations with the Encoder. Those operations are implemented as PyTorch transformation and are computed in this specific order: *Bound Distance*, *Radius Graph*, *Edge Distance*, *Noise Injection* and *Normalize Features*.

6.2.1 **Bound Distance.** For each particle we compute the normalized distance between itself and the 4 (in 2D) boundaries of the simulation domain $\hat{\mathbf{b}}_i$. We need to keep in mind that this distance is normalized by the predefined search radius of 0.02 in our final model.

6.2.2 **Radius Graph.** Next, the graph is built based on the proximity between particles. Fortunately for us, PyG provides a GPU accelerated method to very efficiently create a graph from point cloud data. As visualized in figure 4, the search radius is a very sensitive hyperparameter that can have a huge impact on the performance of this model.

Looking at the figure, on the first row we see the connections generated using the search radius defined above. A small search radius (a) produces a sparse set of connections that might miss important information to describe the neighborhood of each particle. On the other end of the spectrum, using a large search radius (c) will generate a very dense graph that will also be undesirable considering the large amount of irrelevant information that will be gathered and processed by each particle. We, therefore, aim for a balanced search radius (b) that will connect each particle to its relevant neighborhood only. In the second row, we have the same search radius applied to generate the whole graphs where the receptive field of the central particle after 10 iterations of message passing is displayed in red. The sparse graph (d), clearly shows the issue where two particles very close to each other can go through many message passing iterations completely unaware of each other which will dramatically decrease the overall accuracy of the model.

Note that, in PyG, to have particles that are 0.01 units away from each other to be connected, we need to define a search radius of twice that distance, so 0.02 in our case.

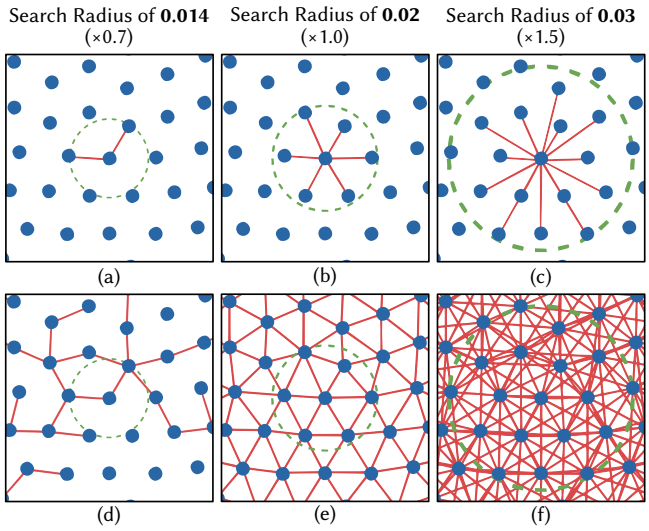

Figure 4: Search radius and its impact on the generated graph. In the first row, we have the connections generated using the search radius defined above. A small search radius (a) produces a sparse set of connections while a large search radius (c) will generate a very dense graphs. In (b) we have a balanced search radius that will connect each particle to its relevant neighborhood. In the second row, we see the same search radius applied to generate the whole graphs where the receptive field of the central particle after 10 iterations of message passing is displayed in red.

6.2.3 **Edge Distance.** This transformation simply measures the length of the created edges. This distance is also normalized by the search radius the same way *Bound Distance* is.

6.2.4 **Noise Injection.** One key component of this architecture is the injection of noise in the training data to synthesize the inaccuracies of the model's predictions making it slowly drift away

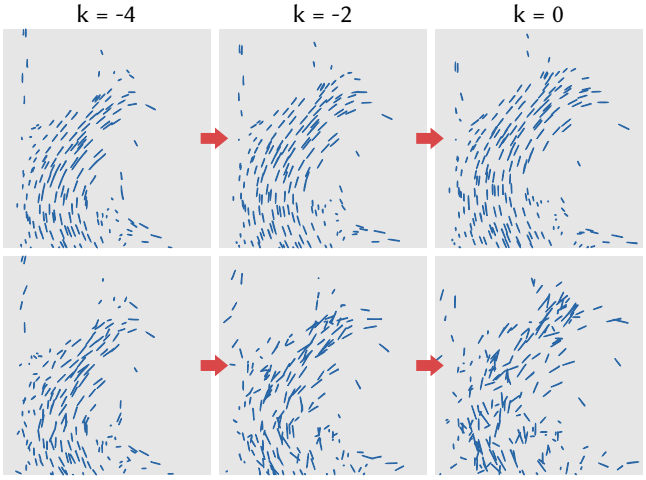

Figure 5: Noise is added to the training data to synthesize the inaccuracies of the predictions. The first row represents the training data with no noise added to it while the second row contains the accumulated noise affecting both the velocities and positions from left to right. We train the model to correct for this drift, which allows it to correct itself when predicting a sequence of frames.

from the ground truth. Noise sampled from a gaussian distribution of mean 0 and variance 0.0003 [15] is progressively accumulated across the history of velocities defining the current state of the particle system.

In figure 5, the first row represents the training data with no noise added to it while the second row contains the accumulated noise affecting both the velocities and positions from left to right. Instead of predicting the ground truth acceleration, our model will be trained to predict the corrected acceleration needed to compensate for the drift injected in the training data, therefore, allowing it to correct itself at inference time. Note that figure 5 shows the configuration used in the original GNS where a velocity history of 5 frames is provided to describe the state of the particle system. In our simplified architecture, we have empirically determined that this velocity history was not required to get good results. We therefore only add noise to the current velocity, using a single frame, and train to correct for it, thus no longer treating it as a random walk.

While this noise greatly improves the global trajectories of the particle, it can sometimes affect the particle-particle interaction accuracy leading to volume loss. To improve on that we propose to decay the injected noise amplitude as we decay the learning rate. Experimenting on this idea has yielded improved results when reducing the noise amplitude by a factor of 0.4 to end the training with noise with a standard deviation of 0.00012. The result section 8 provides a link to a video containing animated results using this technique.

6.2.5 **Normalize Features.** This final transformation uses the pre-computed statistics (mean and standard deviation) of the whole training set to normalize both the velocity and acceleration of the particles, so each dimension is centered at 0 with a standard deviation of 1. Those statistics are stored on disk as training metadata in a JSON file.

These on-the-fly transforms allow us to fit the whole training set in memory and therefore avoid the bottlenecks of having to load mini-batches from disk as training is performed. One caveat to this is that PyTorch requires a considerable amount of memory to produce the file that will be loaded in memory at training time. For example, a generated file of 26GB on disk takes about 100GB of memory to process. This problem only arises at creation time and does not affect the memory footprint when training.

6.2.6 **Edge Latent Representation Update.** The message passing workflow in PyG is very well adapted to update the latent representation of the vertices $\mathbf{v}_i$ while keeping the edges representation $\mathbf{e}_{i,j}$ unchanged. It is therefore important to store the generated messages $\mathbf{m}_{i,j}$ as a member of the message passing object in order to return it as a tuple along with the updated vertex latent representation $(\mathbf{v}_i, \mathbf{m}_{i,j})$.

7 EXPERIMENTS

This section provides important details about the training procedure as well as the different metrics used to evaluate the performance and quality of the predicted simulations.

7.1 Training

Our model is trained in a supervised learning setting where for each example X^t of the training set our model tries to accurately predicts the current normalized acceleration. We use the per-frame MSE loss function to train the model by comparing the predicted values $\tilde{\hat{\mathbf{p}}}_i$ against the ground truth $\hat{\mathbf{p}}_i$ of the N particles in the mini-batch such that

$$\frac{1}{N}\sum_{i=1}^{N} ||\tilde{\hat{\mathbf{p}}}_i - \hat{\mathbf{p}}_i||_2^2. \tag{7}$$

Only the kinematic particles are considered when computing the MSE loss using a binary mask based on the material type attribute.

As the training is performed, the first frame X^{t_0} of a simulation in the validation set is used to compute a full simulation sequence every 10000 gradient update step. We use this as cross-validation using MSE across the whole sequence of frames to make sure we are not overfitting the training data. The full state of the trained model is being written to disk every 50000 gradient update steps (permanent models) as well as every 1000 gradient update steps where only the 5 most recent models are kept on disk (running models). The models were trained on both *Nvidia 1080 Ti* and *Nvidia RTX 8000*. The rate of convergence is similar with both GPUs, but thanks to its large amount of memory, the number of graphs computed in parallel can be increased to 24 on the *RTX 8000* instead of the maximum of 6 on the *1080 Ti*. Bigger batch sizes helped at stabilizing the training procedure, but globally the model was taking longer to converge, so we ended up using a batch size of 2 regardless of the hardware used. The MSE on the training set seems to plateau between 0.1 - 0.2 (depending on the amount of noise injected) after 12h - 36h of training (depending on the architecture used). As seen in figure 6, the training is usually early-stopped at around 1.6 million gradient update steps which approximatively represent

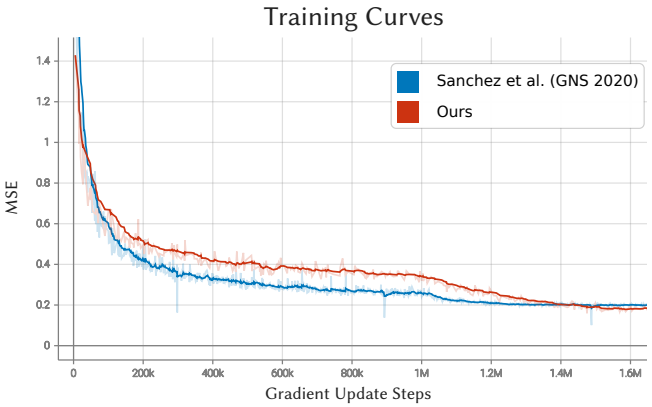

Figure 6: Training curves of the original GNS model from Sanchez et al. compared against our simplified model. The training takes roughly 1.6 million gradient update steps to converge and the learning rate is decayed from 1e-4 to 1e-6 between 1 and 1.5 million gradient update steps. Our model is able to further reduce the error on the training set as we are also decreasing the noise injected into the training set as the learning rate is reduced.

5.38 epochs. To reduce large oscillations in the learning curves close to the end of the training we schedule the learning rate to be exponentially decayed from $1e-4$ to $1e-6$ between 1 and 1.5 million gradient update steps such that

$$\eta \leftarrow \eta \left(\frac{1e-6}{1e-4} \right)^{\frac{1}{1500000-1000000}} \tag{8}$$

if the gradient update step count is in the range $[1000000, 1500000]$. Figure 7 shows that the MSE error on the validation set can be further reduced with the original GNS. Note that their model takes 36 hours to converge while our model reaches the same point within 12 hours of training.

7.2 Evaluation

As mentioned in the previous sections, the quality of the predictions is evaluated using per-frame MSE on the normalized acceleration. Although this is a good metric to train the model on isolated simulation frames, this metric can become misleading in the context of frame sequences because of the accumulation of errors on each frame. As opposed to other metrics based on optimal transport, this one is not permutation invariant which can further bias the calculated error. That being said, we observed a strong correlation between this MSE on normalized acceleration and the perceptual believability of the generated simulations on the validation set. We, therefore, kept this metric as our main quantitative measure because of its efficiency.

Even though more advanced metrics such as Earth Mover's distance [19] should be investigated, there is, unfortunately, no perfect metric that truly captures how real a simulation appears to the human eye. A simulation that is slightly delayed in time could look very realistic while performing badly for both MSE and Earth-Mover distance. Since perceptual believability is still our main target, simulation sequences are frequently generated as training is performed to keep track of the actual progress being made in that regard.

In this work, we use the original GNS from Sanchez et al. [15] as our baseline for both accuracy and speed. Our goal is to find design modifications that will increase the efficiency of the architecture without compromising the perceptual believability of the results.

To validate our claims regarding perceptual believability we will conduct a study where many participants are asked to identify the most realistic simulation from pairs generated with the reference solver (Taichi-MPM) and our simplified GNS model. If the two simulations in the pair are equally realistic, we expect the results to be very close to 50/50 between the two methods, meaning that people are equally likely to pick both methods since they are impossible to differentiate in terms of perceptual believability. On the other hand, if the simulations produced by our model are clearly inferior, we expect the split to be closer to 100/0 in favor of Taichi-MPM.

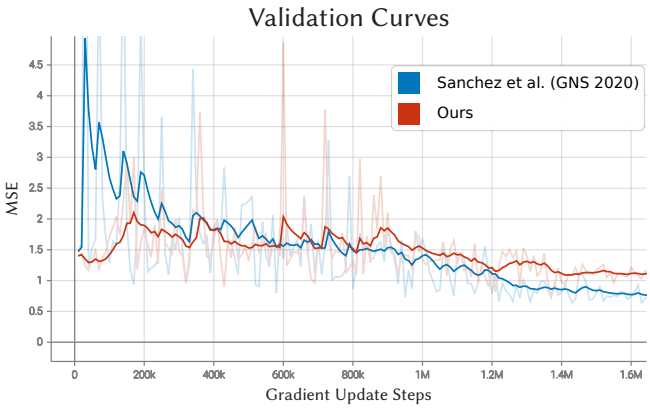

Figure 7: Validation curves of the original GNS model compared against our simplified model. The original GNS is able to further reduce the MSE error on the validation set when compare against our simpler alternative.

8 RESULTS

Considering the large design space of the GNS architecture, we only report on a fraction of the many model configurations that were tested in this work. Figure 1 outlines 7 frames of 3 simulations generated with our simplified GNS model. The color of the particles is derived from their speed where white and dark blue respectively represent fast and slow-moving particles. The gray particles represent static colliders that are not updated by our model. In order to properly evaluate the quality of the generated simulations, we invite the reader to watch the video [1] provided as supplemental material. Figure 8 shows a comparison of a ground truth simulation generated with Taichi-MPM 2D as well a the predicted animations inferred with both the original GNS model by Sanchez et al. and our simplified version. Due to its much larger capacity, the original GNS is able to much more faithfully replicate the ground truth. That being said, our model is not too far off and we argue that

[1]Video featuring most of the simulations presented as figures in this work: https://youtu.be/7jW1nPhBwKQ

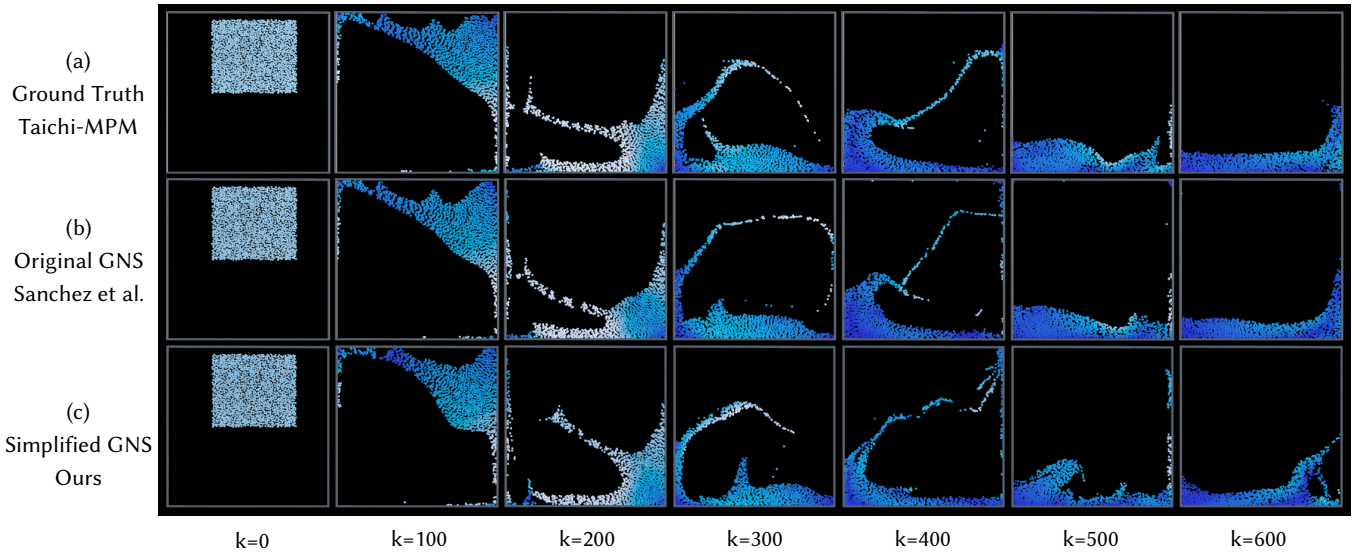

Figure 8: Simulations of 601 frames shown at every 100 frames. Each model generates a full simulation sequence starting with the initial conditions defined at time k = 0 for both the ground truth (a) and our model (c) but at time k = 4 for the original GNS model (b) which requires a velocity history of 5 frames to describe the state of a particle system. The ground truth simulation (a) generated using Taichi-MPM 2D [8]. The original GNS model from Sanchez et al. (b) re-implemented in PyG. Our simplified GNS model (c) as described in this work.

in the absence of the ground truth as a reference, both of those predictions would appear equally believable. The discussion in the next section outlines the differences in computational complexity and perceptual believability between those two models.

A hyperparameter grid search is shown, figure 9, where both the message passing iteration count (x-axis) and the embedding sizes (y-axis) were varied while keeping all the other hyperparameters fixed. On the right of each generated simulation, you can find the per-frame inference time in milliseconds as well as the memory requirements to generate the simulation. Refer to the provided video to see the simulations in motion.

In addition to the many visual supports, we conducted a survey, figure 10, where 72 participants were asked to choose between two simulations generated using the same initial conditions but using 2 different methods to produce 24 pairs of simulations. The participants were invited to base their choice on the realism of each simulation or to simply pick their favorite in cases where both simulations appear equally realistic. The 24 pairs were divided into 3 groups of 8 pairs: Taichi-MPM vs Original GNS, Taichi-MPM vs Our simplified GNS, and Original GNS vs Our simplified GNS. The 24 questions were shuffled for each participant and the position of each option (left / right) was also randomized to prevent any kind of bias. The participants were unaware of the 3 groups and had no idea of how many solvers/models were tested.

9 DISCUSSION

As shown in figure 9, perceptual believability does not improve linearly with respect to the increase in inference time and memory requirement. Starting from the bottom right of the figure and moving toward the top left we clearly see the rapid improvement in the overall appearance of the simulation at little cost, but as we get closer to the top-left corner small improvements made in perceptual believability start to inquire huge cost in performance.

Models using embeddings of size 16 or models constraint to a single message passing iteration clearly lack the capacity to properly capture the complexity of the function we are trying to learn. On the other hand, a model using embeddings of size 128 and 10 iterations of message passing will have no problem learning the function behind our data, but this model is both slow to train/evaluate and requires a ton of memory. The model highlighted in green in figure 9 offers an interesting balance between perceptual believability and inference costs. More specifically, this simpler model, using embeddings of size 64 and performing only 5 iterations of message passing, is 1.8 times faster and requires 3.7 times less memory than the model top-left.

While de design space of the GNS is quite large, many hyperparameters such as layer normalization, self-connections, maximum connection count, edge embedding update, and velocity history frame count have very little impact on the quality of the generated simulations. On the other hand, hyperparameters such as message passing iteration count, data normalization, MLP depth, connection search radius, and embedding sizes can dramatically influence the behavior of our model. In the following sub-sections, we further discuss the specifics of how those hyperparameters influence the quality of the predictions.

9.0.1 **Connection Search Radius.** Increasing the connection radius from 0.014 to 0.020 gives each particle a wider receptive field of its local neighborhood which makes the generated graph denser and therefore helps the global behavior of the fluid by providing a more descriptive context to each particle. Radius larger than 0.020

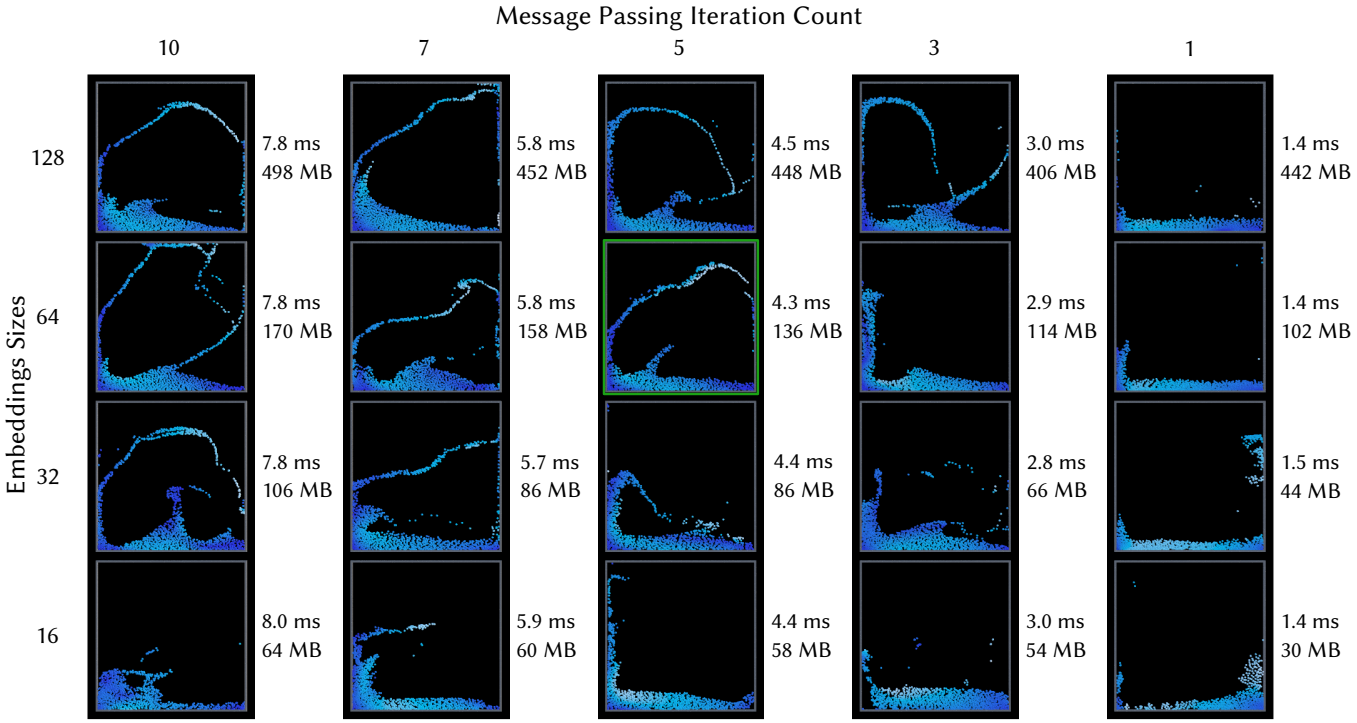

Figure 9: Hyperparameter grid search with message passing iteration count on the x-axis and embedding sizes on the y axis. Model capacity decreases from top left to bottom right. Per-frame inference time and memory requirements are provided to the right of each simulation as a reference of the cost of each design. Performance was measured on an Nvidia 1080 Ti. The model highlighted in green is the one described as our simplified GNS in this work.

tends to confuse the model by aggregating too much information leading to poor particle-particle interaction and volume loss. A large search radius also dramatically slows down the model. In order to reduce the design space and keep the model as efficient as possible, we fix the search radius to an appropriate value based on what is discussed in figure 4 and rely on message passing iterations to vary the receptive field of each particle.

9.0.2 **MLP Depth and Nonlinearities.** Removing the ReLU activation functions and collapsing the MLP network to a single linear transformation prevents our model from learning anything meaningful. This variation clearly lacks the capacity to approximate the target function. For a visual reference, the simulation generated with this configuration is very similar to the outputs obtained with the model with a single message passing iteration and using embeddings of size 16 in figure 9. Changing the MLP depth from 2 to 3 layers (1 and 2 activated layers respectively) slightly reduces the error on the validation set at the cost of making the model noticeably slower. For this reason, we have decided to stick with the MLP of size 2 with a single activated layer for our final design.

9.0.3 **Message Passing Iteration Count.** Given the fixed search radius, low message passing iteration counts such as 1 or 3 can be too restrictive in terms of the receptive field generated as seen in figure 9. Iteration counts between 5 and 10 give good results in general, but the computational complexity scales linearly with respect to the number of iterations. Our final design computes 5 iterations of message passing as it seems to offer the best balance between speed and perceptual believability.

9.0.4 **Graph Latent Representation Size.** While embeddings sizes of 128 seem to be needed in order to perfectly replicate the ground truth, it is possible to obtain very good results with embeddings of sizes 64 and 32 as seen in figure 9. Embeddings of size 16 do not provide enough capacity for our model to capture the subtleties of the underlying function. Thus, our simplified GNS uses embeddings of size 64.

9.0.5 **Data Normalization.** As seen in the last clip of the provided video. Removing velocity and acceleration normalization prevents our model from learning the desired function producing incoherent results. This is an important limitation of the GNS architecture, as it means that in order to use this model for a specific domain, we first need to have access to statistics such as the mean and standard deviation for both the velocity and acceleration. These statistics are computed from the training set which implies that a training set specific to this domain must exist for our model to be usable. This represents an important limitation for generalization to unseen domains. This will most likely be an important challenge for future work.

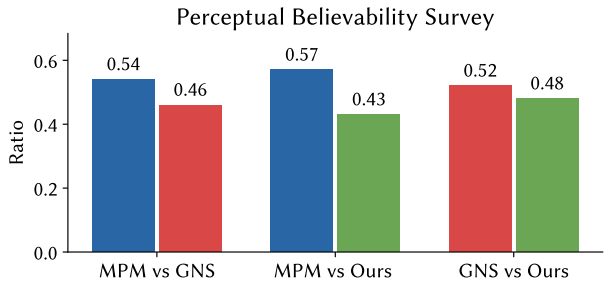

Figure 10: Survey conducted with 72 participants asked to pick the most realistic animated particle simulation out of 24 pairs. Each bar pair (3 in total) is therefore based on a sample of 576 answers. "MPM" represents simulation generated with Taichi-MPM 2D, "GNS" is the original GNS model from Sanchez et al., and "Ours" corresponds to our simplified GNS model. The resulting ratio represents the proportion of picks from each method when presented with a shuffled pair of animation generated using the different methods.

9.1 Optimal Design

To summarize, our simplified GNS uses 64 floats for the embeddings of both the edges and the vertices instead of the 128 floats of the original GNS model. The message passing count was reduced to 5 iterations instead of 10. Our model is no longer using a history of 5 frames of velocity to define each state of the particle system. Perceptually comparable results are obtained using a single frame of velocity which also makes the definition of new initial conditions much more practical. This is also in agreement with how solvers in physics and engineering are usually designed. We have also been able to reduce the depth of all MLPs of the architecture from 3 to 2 layers without serious degradation in perceptual quality. In practice, these simplifications have sped up the training process from 36 hours to only 12 hours. The inference is performed in 4.0 ms/frame instead of 9.1 ms/frame. Our model also requires less than a third of the original memory requirements to evaluate the model for a total of 142 MB instead of 468 MB.

9.2 Survey based on Perceptual Believability

To evaluate the impact of those simplifications on perceptual believability we have conducted the previously introduced survey, figure 10. Thanks to its greater expressivity, the original GNS yields superior results in the survey, since when compared against the ground truth, 46% of respondents are picking the original GNS over the ground truth. This is 3% higher than our simplified GNS at 43%. The direct comparison between the two GNS also exposes a difference in perceptual believability in favor of the original GNS.

That being said, keeping in mind that ratios of 50/50 mean that the 2 methods are indistinguishable from each other and that 100/0 means that the first method is clearly superior we argue that those differences in ratios are negligible compared to the sheer increase in performance offered by the simplified GNS model, as described in the section above.

10 CONCLUSION

We present a visually rich exploration of the design space of the GNS architecture guided by perceptual believability. This work provides a new set of hyperparameters to make the GNS architecture scalable and therefore more suitable for production in both visual effects and games. Although there is still optimization and exploration to be done in order to make this model competitive with current methods, it is a step in the right direction. Our exploration of the design space also shows that this architecture will remain stable even under extreme capacity constraints which is not true for many algorithms used in physics simulation. As demonstrated in this work, using perceptual believability to evaluate the quality of simulation allows simpler and faster models to truly shine in situations where physical accuracy is not the ultimate goal.

REFERENCES

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[2] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261* (2018).

[3] Peter W Battaglia, Razvan Pascanu, Matthew Lai, Danilo Rezende, and Koray Kavukcuoglu. 2016. Interaction networks for learning about objects, relations and physics. *arXiv preprint arXiv:1612.00222* (2016).

[4] Jie Chen, Haw-ren Fang, and Yousef Saad. 2009. Fast Approximate kNN Graph Construction for High Dimensional Data via Recursive Lanczos Bisection. *Journal of Machine Learning Research* 10, 9 (2009).

[5] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[6] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *International conference on machine learning*. PMLR, 1263–1272.

[7] Donghui Han, Shu-wei Hsu, Ann McNamara, and John Keyser. 2013. Believability in simplifications of large scale physically based simulation. In *Proceedings of the ACM Symposium on Applied Perception*. 99–106.

[8] Yuanming Hu, Yu Fang, Ziheng Ge, Ziyin Qu, Yixin Zhu, Andre Pradhana, and Chenfanfu Jiang. 2018. A Moving Least Squares Material Point Method with Displacement Discontinuity and Two-Way Rigid Body Coupling. *ACM Transactions on Graphics* 37, 4 (2018), 150.

[9] L'ubor Ladickỳ, SoHyeon Jeong, Barbara Solenthaler, Marc Pollefeys, and Markus Gross. 2015. Data-driven fluid simulations using regression forests. *ACM Transactions on Graphics (TOG)* 34, 6 (2015), 1–9.

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

[11] Joe J Monaghan. 1992. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics* 30, 1 (1992), 543–574.

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

[13] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Icml*.

[14] Carol O'Sullivan and John Dingliana. 2001. Collisions and perception. *ACM Transactions on Graphics (TOG)* 20, 3 (2001), 151–168.

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

[16] Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. 2018. Graph networks as learnable physics engines for inference and control. In *International Conference on Machine Learning*. PMLR, 4470–4479.

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

[18] Benjamin Ummenhofer, Lukas Prantl, Nils Thuerey, and Vladlen Koltun. 2019. Lagrangian fluid simulation with continuous convolutions. In *International Conference on Learning Representations*.

[19] Cédric Villani. 2021. *Topics in optimal transportation*. Vol. 58. American Mathematical Soc.

[20] Thomas Y Yeh, Glenn Reinman, Sanjay J Patel, and Petros Faloutsos. 2009. Fool me twice: Exploring and exploiting error tolerance in physics-based animation. *ACM Transactions on Graphics (TOG)* 29, 1 (2009), 1–11.