Learning rigid-body simulators over implicit shapes for large-scale scenes and vision

Yulia Rubanova* **Tatiana Lopez-Guevara** **Kelsey R. Allen** **William F. Whitney**
Kimberly Stachenfeld† **Tobias Pfaff†**

Google Deepmind

Abstract

Simulating large scenes with many rigid objects is crucial for a variety of applications, such as robotics, engineering, film and video games. Rigid interactions are notoriously hard to model: small changes to the initial state or the simulation parameters can lead to large changes in the final state. Recently, *learned* simulators based on graph networks (GNNs) were developed as an alternative to hand-designed simulators like MuJoCo [36] and PyBullet [13]. They are able to accurately capture dynamics of real objects directly from real-world observations. However, current state-of-the-art learned simulators operate on meshes and scale poorly to scenes with many objects or detailed shapes. Here we present SDF-Sim, the first learned rigid-body simulator designed for scale. We use learned signed-distance functions (SDFs) to represent the object shapes and to speed up distance computation. We design the simulator to leverage SDFs and avoid the fundamental bottleneck of the previous simulators associated with collision detection. For the first time in literature, we demonstrate that we can scale the GNN-based simulators to scenes with hundreds of objects and up to 1.1 million nodes, where mesh-based approaches run out of memory. Finally, we show that SDF-Sim can be applied to real world scenes by extracting SDFs from multi-view images.

1 Introduction

Simulating real-world environments, such as a bookshelf with books and decorations or a dinner table with plates and glasses, presents a significant challenge for traditional physics simulators. These simulators require precise 3D shape, location, and corresponding physical parameters of each object, making the simulation of arbitrary scenes a difficult task.

Recently, *learned* simulators based on graph networks (GNNs) [4, 19, 29] have been introduced as a powerful alternative to traditional hand-designed simulators like MuJoCo [36] or PyBullet [13]. Graph networks can capture a range of complex physical dynamics, learn physical properties directly from real data and generalize to new scenes. However, current GNN-based methods do not scale well to large scenes. Similarly to the traditional simulators, they rely on 3D meshes to describe object shapes. In scenes with a large number of objects or objects with detailed meshes, these scenes might contain thousands of nodes and mesh triangles, making collision detection between objects extremely costly. In the context of graph networks, large meshes also lead to input graphs that might contain hundreds of thousands of nodes and edges, crippling runtime and memory performance.

To remedy the issue with expensive collision detection, a common approach in the classic simulation literature is to use *signed-distance functions* or *fields* (SDFs). SDFs implicitly represent the object

*Corresponding email: rubanova@google.com
†Co-senior authorship

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

 https://doi.org/10.52202/079017-3997

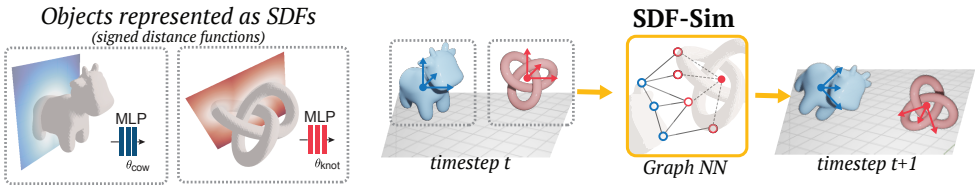


Figure 1: Overview of SDF-Sim pipeline. SDFs parameterized by MLPs are learned for each object to implicitly represent the object shape and the distance field. A GNN-based simulator uses learned SDFs to predict object dynamics for the next simulation step.

shapes and allow to find the distance and the closest point on the object surface from an arbitrary location in 3D space in constant-time. Since distance queries are a main driver of the compute cost in traditional rigid body simulation, they can be significantly sped up using SDFs. However, in practice, SDFs are frequently pre-computed from a mesh and stored as a 3D grid, which trades off the runtime efficiency for increased memory cost and limits their usefulness for large real scenes.

An orthogonal line of work started to explore SDFs *learned* from a set of images to reconstruct the 3D shape from vision. Those SDFs store the shape implicitly in the MLP weights and are fast to train and query. They are less memory-hungry compared to 3D grids, making them a perfect candidate for simulation. However, despite these advantages, learned SDFs were applied for dynamics scenes only in a limited context [14, 23].

We present SDF-Sim, a learned graph-network-based simulator for rigid interactions that uses learned SDFs to represent object shapes, Figure 1. Using a special design of the input graph, SDF-Sim allows us to substantially reduce the runtime and memory requirements compared to mesh-based learned simulators. This is the first demonstration of a learned simulator generalizable to extremely large scenes up to 1.1 million nodes in Figure 2, orders of magnitude larger than what have been shown in any previous work on learned rigid simulation. These simulations include concave shapes (shoes, a hanger) and thin structures (screwdriver, baking form). Finally, we show that SDF-Sim can directly work with SDFs obtained from multi-view RGB images, Figure 3, supporting rich 3D simulation of objects extracted from real scenes.

2 Background

Mesh-based simulation A simulation can be represented as a time series of system states $\mathcal{S}^1, \ldots, \mathcal{S}^T$. The goal is to learn a neural simulator that predicts the next state $\mathcal{S}^{t+1}$ given a history of previous states $\{\mathcal{S}^{t-h+1}, \ldots, \mathcal{S}^t\}$. Simulators based on graph networks (GNNs) [8, 29] encode the system state into a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with nodes $\mathcal{V}$ and edges $\mathcal{E}$. For rigid body simulation, this graph can be constructed from the triangle mesh of the individual objects: mesh vertices become the graph nodes, and mesh edges act as graph edges. The object motion is computed by message passing across the nodes and edges in the graph. Within individual objects, the position, rotation and velocity of the object can be computed by message passing through the nodes and edges of that object.

Why learned simulators? Analytical simulators like MuJoCo, PyBullet or Drake, are most commonly used for modelling rigid bodies. However, the traditional simulators rely on hard-coded approximations of physical interactions that might not match the properties of real objects, even with careful hyperparameter tuning[1]. The predictions from these simulators inevitably diverge from observations of real objects – a so-called sim-to-real gap [17].

Learned simulators have unique advantages that analytical simulators don't provide. First, learned simulators can be trained directly on real-world observations. They can track the real object trajectory better than analytical simulators, solving a well-known sim-to-real gap[17]. Another common issue is precisely estimating the initial states, which analytical simulators rely on – learned simulators can compensate for these inaccuracies [3]. Finally, learned simulators are differentiable and can be used for optimization and design[2]. At the same time, learned simulators are not optimized for runtime and memory, are slower and more memory-constrained than analytical simulators. In this work, we address that limitation of memory constraint and unlock the ability to run learned simulators on larger scenes in comparison to previous learned simulators.

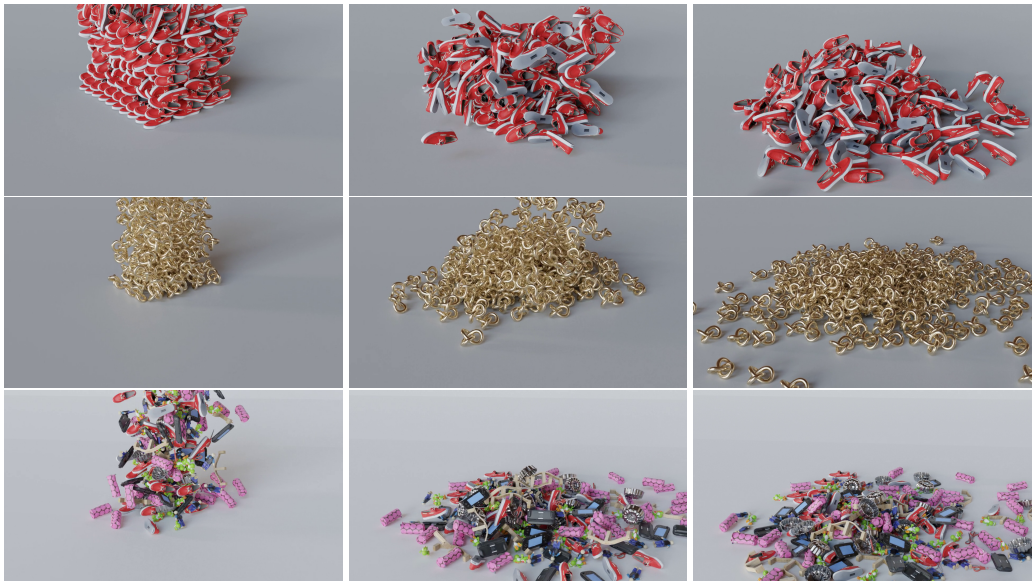


Figure 2: Example of rollouts from SDF-Sim scaled to large simulations, all simulated for 200 steps. (Top) 300 shoes (object from Movi-C), with 851k nodes, falling onto the floor (Middle) 270 knots from Movi-B, 384k nodes (Bottom) 380 objects from Movi-C, 1.1M nodes. See simulation videos on https://sites.google.com/view/sdf-sim.

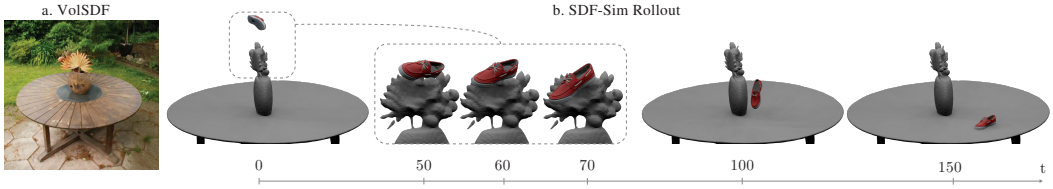


Figure 3: Simulating assets extracted from vision. (a) We extract the SDF from the images of a real-world scene with a garden table [7]. (b) We simulate a virtual shoe object falling onto a vase and a table using SDF-Sim. SDF-Sim is able to predict realistic dynamics, even for the collision of the shoe with the intricate shape of the vase (frames 50-70). See section 4.4 for details and the video on the website.

Modelling collisions in GNN simulators The most challenging component of the simulation is computation of collision impulses between objects. To do so, GNN simulators introduce collision edges $\mathcal{E}^{coll}$ between nodes [31] or triangles [4] on the mesh surface that belong to different objects and are within a predefined distance threshold. However, the amount of these edges is the main bottleneck of GNN-based simulators. Asymptotically, the number of potential collision edges grows quadratically with the number of simulated nodes, leading to prohibitive compute and memory costs. Another challenge is identifying which pairs of triangles/nodes to connect with collision edges in the first place, by computing the distance to the closest point on a mesh. Typically, this procedure is implemented with tree search methods over all mesh triangles in the scene, such as BVH [11], that are difficult to integrate into deep learning pipelines. In this work, we address both of these challenges by using SDFs.

Signed distance functions Signed-distance functions (SDFs) are widely used in computer graphics, game engines, and robotics for fast collision detection and computation of distances to an object [33, 45]. SDF defines a field $f(\mathbf{y}) : \mathbb{R}^3 \rightarrow \mathbb{R}$ that represents the signed distance from an arbitrary point $\mathbf{y}$ to the closest point on the surface of the object. The sign of the SDF determines whether a point is outside (positive) or inside (negative) of the object. The zero level of the SDF $\{\mathbf{y} \in \mathbb{R}^3 | f(\mathbf{y}) = 0\}$ implicitly represents the object surface. SDFs permit constant-time queries of the distance to an object surface, irrespective of the number of nodes/faces in the object mesh, which is an essential component of collision detection.

https://doi.org/10.52202/079017-3997

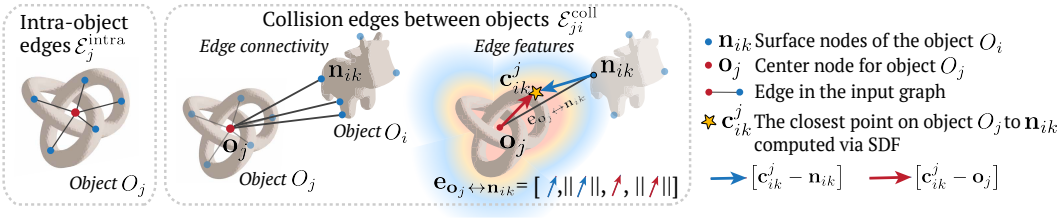


Figure 4: Construction of graph edges in SDF-Sim.

Constructing an SDF In the computer graphics and simulation literature, SDF is often pre-computed as a high-resolution 3D grid containing signed distances from the points on the grid to the object surface [26, 37]. The grid SDF allows to speed up distance queries by trading off memory: for example, 512x512x512 grid would take ≈134M voxels (0.5Gb of memory) for a single object.

Learned SDFs started to gain popularity for reconstruction of water-tight 3D shapes from images. They approximate the continuous distance field $f(\mathbf{y}; \theta)$ with an MLP parameterized by θ [23, 28, 43]. Unlike 3D-grid-SDFs, learned SDFs are not tied to a fixed grid resolution and can represent detailed shapes using a small set of parameters θ. Despite these advantages, *learned* SDFs have not been sufficiently explored to speed up physical simulations. Limited available works combine learned SDFs with classic physics solvers [5, 33]. They demonstrate that learned SDFs can massively reduce the distance query time thanks to parallelization on a GPU compared to mesh-based computation, while taking ∼32x less memory than traditional 3D-voxel-grid SDFs.

Computing closest points For any point in 3D space $\mathbf{y}$ an SDF f_θ allows us to easily compute the closest point $\mathbf{y}^*$ on the object surface that it represents, as:

$$\mathbf{y}^* = \mathbf{y} - f_\theta(\mathbf{y}) \nabla f_\theta(\mathbf{y}), \tag{1}$$

where, by definition, $f_\theta(\mathbf{y})$ is the distance between $\mathbf{y}$ and $\mathbf{y}^*$, and the gradient $\nabla f_\theta(\mathbf{y})$ points in the opposite direction of the shortest path from $\mathbf{y}$ to the surface of the object and is unit-norm. If $f_\theta(\mathbf{y})$ is parameterized as an MLP, this calculation requires only one forward and one backward pass of the network. This provides an efficient way to calculate the closest points for collision resolution in the simulation. In this work, for the first time in the literature, we use learned SDFs to accelerate distance computation in SOTA graph-network simulators.

3 SDF-Sim

We introduce SDF-Sim, a graph-network-based simulator for rigid objects that uses learned SDFs to represent object shapes and to perform fast and memory-efficient distance computation between the objects. By leveraging SDF properties, we propose a new way to construct the input graph for the graph network, allowing us to use a smaller graph size and to get an order-of-magnitude reduction in runtime and memory on large simulations.

3.1 Training SDF functions per object

We represent SDF $f_\theta(\mathbf{y})$ as an MLP that takes in a 3D point $\mathbf{y} \in \mathbb{R}^3$ and outputs a scalar SDF value. Learned SDFs are pre-trained separately for each object and remain fixed throughout the simulation.

Meshes To compare to the existing mesh-based baselines, we apply SDF-Sim on benchmarks where meshes are available and train the learned SDF $f_\theta(\mathbf{y})$ from a mesh. To train an SDF, we sample points in 3D space and compute the ground-truth signed distances from these points to the mesh surfaces using a classic BVH [11] algorithm. Finally, we train an MLP $f_\theta(\mathbf{y})$ to fit these distances with supervision. See more details in section C. We use the same architecture and the model size for each object shape of 8 MLP layers with 128 units each, unless otherwise stated.

Vision We use VolSDF [43] to distill an SDF from a set of images representing a 360-degree view of the outdoor garden scene first described in [7] with camera poses estimated via COLMAP [32]. See visualisation in Figure 3 and section C.2 for details.

3.2 Learned simulator

Object representation We represent the shape of the object O_i in the reference pose (centered at zero) as a learned SDF function f_{θ_i}. At a simulation step t, a rigid transformation $\mathcal{T}_i^t = (\mathbf{p}_i^t, \mathbf{R}_i^t)$ transforms the object from reference pose to the current pose in the simulation. Here, $\mathbf{p}_i^t$ is an object translation, corresponding to the object's center of mass at timestep t; $\mathbf{R}_i^t$ is a rotation. The task of the learned simulator is predicting the next-step transformation $\mathcal{T}_i^{t+1}$ for each object. In the following text, we omit the time index t for brevity.

To represent I objects in the input graph, we introduce *object* nodes $\mathcal{V}_O = \{\mathbf{o}_i | i = 1..I\}$, located at position $\mathbf{p}_i$, and a set of *surface* nodes $\mathcal{V}_S = \{\mathbf{n}_{ik} \in O_i | i = 1..I, k = 1..K_i\}$ on the surface of the objects, where the number of nodes K_i may differ for each object. These surface nodes $\{\mathbf{n}_{ik}\}$ move with their corresponding object according to the transformation $\mathcal{T}_i^t$. With a slight abuse of notation, we will refer to $\{\mathbf{n}_{ik}\}$ both as the node entities and their 3D position in the simulation space.

Nodes and edges within the object To construct the graph connectivity *within* an object, we follow an established line of work on learned simulators [4, 19, 29]. We connect surface nodes $\{\mathbf{n}_{ik}\}$ to their corresponding object node $\mathbf{o}_i$ using edges $\mathcal{E}_i^{\text{intra}} = \{\mathbf{e}_{\mathbf{o}_i \leftrightarrow \mathbf{n}_{ik}} | \mathbf{n}_{ik} \in O_i\}$. Thus, all the information about object motion, e.g., impulses from collision events, is propagated between the nodes via the object node $\mathbf{o}_i$. As shown by [19] we can omit the surface edges between the nodes $\{\mathbf{n}_{ik}\}$ without loss of accuracy.

In graph networks, nodes and edges are associated with feature vectors that can hold the information about the motion and the relation between the nodes. We follow the approach of [4, 19, 29] to construct a set of nodes and edge features that are necessary for simulation: To construct the node features for surface nodes $\mathbf{n}_{ik}$, we compute the finite-difference velocity estimates in the simulation space using a history of the latest three timesteps $\mathbf{v}_{ik} = (\mathbf{n}_{ik}^t - \mathbf{n}_{ik}^{t-1}, \mathbf{n}_{ik}^{t-1} - \mathbf{n}_{ik}^{t-2})$. We set node features to be $[\mathbf{v}_{ik}, ||\mathbf{v}_{ik}||, \phi_i]$ where ϕ_i are the constant object parameters: mass, friction and restitution. We use the same procedure for the object nodes $\{\mathbf{o}_i\}$ using their positions $\mathbf{p}_i$. For intra-object edges $\mathcal{E}_i^{\text{intra}}$, we use displacement vector between the surface node position and the object center as the edge feature $\mathbf{e}_{\mathbf{o}_i \leftrightarrow \mathbf{n}_{ik}} = [\mathbf{o}_i - \mathbf{n}_{ik}, ||\mathbf{o}_i - \mathbf{n}_{ik}||]$.

SDF-based inter-object edges. Here we introduce a new way to construct collision edges between the objects by leveraging their SDF representations (Figure 4). We design these edges such that they contain sufficient information to detect collisions, while their number remains linear in the number of nodes. This is unlike quadratic number of collision edges in mesh-based simulators.

Consider two objects O_i and O_j. For a node $\mathbf{n}_{ik}$ on O_i, we directly query the SDF f_{θ_j} of object O_j to get the distance d_{ik}^j from $\mathbf{n}_{ik}$ to the closest point on O_j. We note that unlike mesh-based approaches, this is a single test, and we do not need to calculate the distance from $\mathbf{n}_{ik}$ to *each* node/triangle on O_j. Then, if this distance d_{ik}^j is within a predefined distance threshold $\mathcal{D}$, we connect the surface node $\mathbf{n}_{ik}$ directly to the opposing *object* node $\mathbf{o}_j$ and refer to this edge as $\mathbf{e}_{\mathbf{o}_j \leftrightarrow \mathbf{n}_{ik}}$. Thus, we define the set of inter-object edges as $\mathcal{E}_{ji}^{\text{coll}} = \{\mathbf{e}_{\mathbf{o}_j \leftrightarrow \mathbf{n}_{ik}} | \forall \mathbf{n}_{ik} \in O_i : f_j(\mathbf{n}_{ik}) \leq \mathcal{D}\}$. This approach is different from mesh-based simulators [4, 19, 29] that connect pairs of nodes or triangles on the two surfaces.

We construct the features for collision edges $\mathbf{e}_{\mathbf{o}_j \leftrightarrow \mathbf{n}_{ik}}$ such that they contain information about potential points of collision. First, we compute the closest point $\mathbf{c}_{ik}^j$ from $\mathbf{n}_{ik} \in O_i$ to the surface of object O_j. To do so, we transform the position of $\mathbf{n}_{ik}$ into the reference space of O_j using $\mathcal{T}_j^{-1}(\mathbf{n}_{ik})$. We call an SDF function to get the distance from the node $\mathbf{n}_{ik}$ to the closest point on O_j as $d_{ik}^j = f_{\theta_j}(\mathcal{T}_j^{-1}(\mathbf{n}_{ik}))$. The closest point on the surface of O_j is then computed similarly to Eq. 1:

$$\mathbf{c}_{ik}^j = \mathbf{n}_{ik} - d_{ik}^j \mathcal{T}_j \Big(\nabla f_{\theta_j} \left(\mathcal{T}_j^{-1}(\mathbf{n}_{ik}) \right) \Big) \tag{2}$$

Note that $\mathbf{c}_{ik}^j$ lies on the surface defined by f_{θ_j}, but does not have to coincide with any surface node and is not part of node set $\mathcal{V}$. Thus, SDFs allow us to test contact between two objects at higher fidelity, without relying on the density of the surface nodes.

Finally, we construct the features for the collision edges as follows: $\mathbf{e}_{\mathbf{o}_j \leftrightarrow \mathbf{n}_k} = [\mathbf{c}_{ik}^j - \mathbf{n}_{ik}, \mathbf{c}_{ik}^j - \mathbf{o}_j, ||\mathbf{c}_{ik}^j - \mathbf{n}_{ik}||, ||\mathbf{c}_{ik}^j - \mathbf{o}_j||]$. These features provide information about the relative position of the closest point (a potential collision point) within the object O_j as well as its relative location

 https://doi.org/10.52202/079017-3997

to the node $\mathbf{n}_{ik}$ on the opposing object. This information is sufficient for the neural network to resolve collisions. Through message-passing over such input graph, the object node $\mathbf{o}_j$ can collect the information from all the nodes that are within a collision radius relative to the object O_j. The model also has access to the velocity and rotation history for both objects through node features, so it is able to infer how fast the node $\mathbf{n}_{ik}$ is changing its position w.r.t. O_j.

With such construction, SDF-Sim requires significantly fewer edges than mesh-based methods, because a single collision edge can test a node against an entire object surface. Asymptotically, SDF-Sim has $\mathcal{O}(I \cdot K)$ edges in the worst case, with the number of objects I being magnitudes smaller than the total number of surface nodes $K = \sum_i K_i$, instead of $\mathcal{O}(K^2)$ for mesh-based simulators. As we demonstrate below, this choice of input graph and simulator unlocks the ability to scale to very large scenes that has not been previously shown in the literature.

Graph network simulator We encode the input graph using MLPs for each node and edge type. We process the graph using 10 message passing steps as in [4, 29]. We decode the processed surface node features into an acceleration estimate $\mathbf{a}_{ik}$ for each surface node $\mathbf{n}_{ik}$, and compute a per-node position update $\hat{\mathbf{n}}_{ik}^{t+1}$ using Euler integration. Finally, next-step rigid transformations $\{\hat{\mathcal{T}}_i^{t+1}\}$ are computed from $\hat{\mathbf{n}}_{ik}^{t+1}$ using shape matching [25], following [4]. The simulator is trained on a single-step prediction task using a per-node L2 loss on the acceleration estimate $\mathbf{a}_{ik}$.

4 Results

We start by evaluating the accuracy and efficiency of SDF-Sim on small-scale simulation benchmarks, where evaluation of the baselines is feasible as well. Subsequently, we demonstrate the scaling properties of the models on scenes with an increasing number of objects. We show that SDF-Sim produces realistic rollouts of extremely large scenes with up to 1.1M nodes, which was not possible with the previous generation of learned simulators. We emphasize that the benefits of SDF-Sim are not specific to simulations with similar object shapes, where separate SDFs are used for each object. Finally, we present ablations for how the quality of the learned SDFs affects the simulation. All simulation videos are available on the website https://sites.google.com/view/sdf-sim.

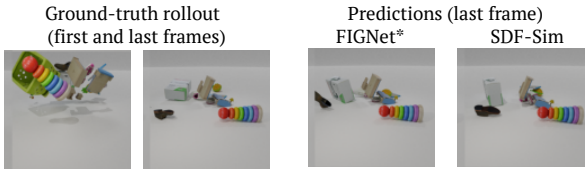


Figure 5: Comparison of the last frames of rollouts predicted on Movi-C. See more frames in Figure S9 and on the website.

Kubric datasets We evaluate SDF-Sim on the benchmark Kubric datasets [16] (shared with Apache 2.0 license), consisting of simulated trajectories of rigid objects thrown towards the center of the scene. We evaluate on two difficulty tiers: Movi-B and Movi-C. Movi-B simulations contain eleven synthetic shapes (e.g., cube, torus, cow) with up to 1142 nodes each. Movi-C comprises 930 scanned real-world household objects, including hollow, flat, or otherwise non-trivial surfaces with thousands of triangles. Both Movi B/C contain only 3-10 objects per simulation. We provide more details about mesh sizes in Movi-B/C in supplement A.1.

We perform many of our quantitative comparisons on Movi-B/C datasets with small-scale scenes because these domains are small enough for us to run also baseline methods to quantify efficiency and accuracy. On Movi-C we compare only to FIGNet* [19], because other baselines run out of memory during training due to the large number of collision edges, as reported by Allen et al. [4].

Baseline models We compare SDF-Sim to the existing state-of-the-art learned simulators for rigid-body interactions: FIGNet [4] and FIGNet* [19]. Both methods are based on graph networks. Unlike SDF-Sim, they operate directly over object meshes and use a special type of graph edges between mesh triangles. Their runtime and memory costs grow with the number and size of the object meshes used for the simulation. FIGNet* is a memory-efficient version of FIGNet that omits edges between the surface nodes of each object. In their original publication, FIGNet was demonstrated to scale up to 10 objects with a few thousands nodes each [4], while FIGNet* was tested up to a larger table scene with 40k triangles, but with only one moving object [19].

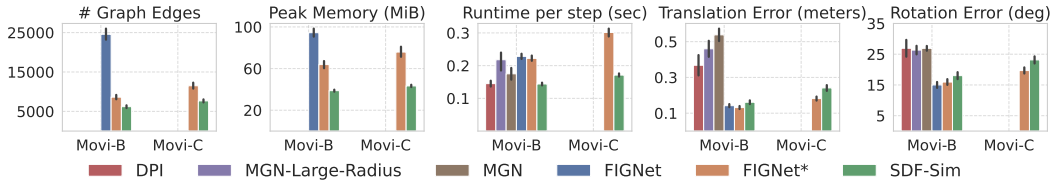


Figure 6: Accuracy, memory and runtime comparisons between the SDF-Sim model and the mesh-based baselines on the Movi-B/C benchmarks. On Movi-C, most baselines except FIGNet* run out of memory and are not shown. As "Peak Memory" we report the peak memory used by the model per single step of the simulation. DPI, MGN-Large-Radius and MGN results were reported by [4]. See Tables 3 and 4 for the exact numbers.

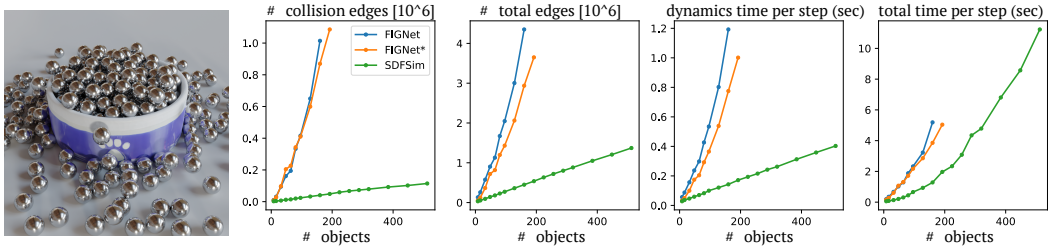


Figure 7: Large-scale simulation of **Spheres-in-Bowl**, simulated for 200 timesteps. Left: the final step of SDF-Sim rollout on the scene with 512 spheres. Right: number of edges and runtime w.r.t. the number of spheres in the simulation (max 512). In complex simulations with lots of contacts, FIGNet and FIGNet* generate an excessive number of collision edges, quickly exceeding GPU memory (end of the orange and blue lines). SDF-Sim generates an order-of-magnitude fewer collision edges, and can easily simulate scenes with 100s of objects without running out of memory.

We additionally include previously reported results from Allen et al. [4] on Movi-B for the following models: DPI [18] that represents the objects as a set of disjoint particles; as well as MeshGraphNets (MGN) and MGN-Large-Radius [29] that use inter-object edges between mesh surface nodes. As reported by Allen et al. [4], DPI, MGN and MGN-Large-Radius baselines suffer from a prohibitively large size of the inputs graph and can only run on small Movi-B simulations.

4.1 Baseline comparison on datasets of small scenes.

First, we evaluate SDF-Sim on Movi-B/C datasets with up to 10 objects. As shown in Figure 6, our SDF-Sim uses substantially sparser graphs to represent the scenes, with 28% fewer graph edges compared to FIGNet* on Movi-B and 33% fewer on Movi-C. This translates into reduction of peak memory required to execute one step of the simulation; 39% reduction on Movi-B and 42.8% on Movi-C. The reduction in the number of edges in SDF-Sim is enabled by object-to-node collision edges that scale linearly with the number of nodes, as opposed to quadratic number of face-face collision edges in FIGNet and FIGNet* (in the worst case, see section 2). Sparser graphs in SDF-Sim also lead to lower average runtime per step of a rollout, by 36% in Movi-B and 43% in Movi-C, because the graph network performs edge updates for/over fewer edges even though the number of nodes in the graph remains the same. Figure S2(a) shows that the runtime of SDF-Sim is consistently lower than that of FIGNet* for varying sizes of the input graph. In the next section we show how these efficiency improvements translate into order-or-magnitude gains on large-scale simulations.

In terms of simulation accuracy, measured as object translation and rotation RMSE errors, SDF-Sim has substantially lower errors than previous baselines DPI, MGN and MGN-Large-Radius on Movi-B. However, in comparison to SOTA models FIGNet and FIGNet*, SDF-Sim has a slightly higher error. Note that the errors of SDF-Sim are already very low: the translation error of SDF-Sim is 0.24 meters, which is only 4.9% of the average object travel distance of 4.92 meters in Movi-C dataset (see tables Tables 3 and 4 for exact numbers). In rigid-body systems, even small discrepancies in predicted positions/rotations can lead to a drastically different object trajectory after a collision, and we consider 4.9% deviation to be a good result. Finally, in the next section we will demonstrate that *on large scenes* SDF-Sim is actually *more accurate* in comparison to FIGNet*.

https://doi.org/10.52202/079017-3997

4.2 SDF-Sim scales to scenes with up to 512 objects

To study the scaling properties of the model, we created **Spheres-in-Bowl**: a set of simulations with a variable number of spheres, ranging from 1 to 512, being dropped into a bowl (Figure 7). We create the ground-truth for these scenes using PyBullet, the same simulator as used for Kubric Movi-B/C. We evaluate a set of learned simulators: SDF-Sim and FIGNet* models trained on Movi-C, as well as FIGNet trained on Movi-B. Note that the largest version, with 512 spheres, has over 50 times more objects than in Movi-B/C datasets used for training.

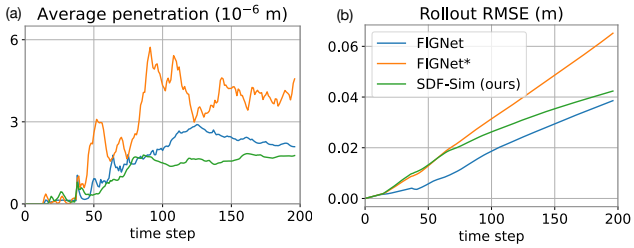


Figure 8: Accuracy metrics w.r.t. simulation time step for **Spheres-in-Bowl** simulation shown in Figure 7. (a) Penetration metrics. (b) Rollout RMSE. Both metrics are averaged over simulation runs with up to 140 spheres, the maximum for which all baselines could be run.

As shown in Figure 7, SDF-Sim has an order-of-magnitude fewer collision edges, compared to FIGNet* and FIGNet. This difference is substantial, as the number of edges dominates the memory cost. FIGNet and FIGNet* run out of memory for simulations with >140 and >160 objects, respectively, whereas SDF-Sim can simulate the entire set of 512 interacting objects using the same Nvidia V100 GPU. In terms of total runtime, SDF-Sim is up to 5 times faster than FIGNet*, even including the time for querying learned SDFs. To the best of our knowledge, this is the first demonstration of a learned simulator successfully scaling to scenes with hundreds of objects and thousands of collisions, despite being trained only on ten objects per scene.

Next, we evaluate the accuracy of these simulations. SDF-Sim has the lowest penetration distance throughout the simulation, as shown in Figure 8(a). Figure 8(b) shows the rollout errors across different steps of the simulation. Notably, SDF-Sim generalizes to these larger scenes better than FIGNet* and has lower error, despite slightly lower accuracy on datasets with smaller scenes (section 4.1). FIGNet has the lowest error out of learned models, likely because it was trained on Movi-B, which contains a sphere object, while Movi-C dataset, used for training Fignet* and SDF-Sim, does not. Despite not having seen a sphere object in training, SDF-Sim performs very well on both penetration and rollout metrics, indicating that the SDF representation is not only efficient, but also shows good generalization over geometry.

4.3 Scaling to extra-large scenes with up to 1.1 million nodes

Next, we provide a further qualitative demonstration that SDF-Sim can scale to extra-large scenes and produce realistic rollouts. We design three scenes with falling stacks of shoes, metal knots, and mixed objects taken from the Movi-C dataset, shown in Figure 2. Such contact-rich simulations are challenging even for analytical simulators and are classic test examples in the computer graphics literature. These simulations consist of: 300 shoes with 851k nodes, 270 knots with 384k nodes, and 380 various Movi-C objects with 1.1M nodes, respectively. All of these scenes are orders of magnitude larger than those in Movi-C dataset used for SDF-Sim training.

Simulations in Figure 2 show that SDF-Sim can scale to these massive scenes without running out of memory and produce qualitatively realistic rollouts of 200 steps; see videos. Note that we cannot run any other learned baseline on these scenes, as due to the large number of potential contacts, these models produce a vast number of collision edges and exceed the GPU memory.

4.4 Simulating real-world scenes from vision

Simulating large real-world scenes is the primary area where the ability to generalize to new, larger scenes is crucial, e.g., in virtual reality applications. Here we present a proof-of-concept that SDF-Sim can successfully handle real scenes, despite being trained only on small simulated assets. We take a Garden scene [7], represented by a sequence of RGB images and use VolSDF [43] to distill a 3D SDF representation of the table and the vase from this scene. SDF-Sim allows to cleanly integrate this 3D representation into a simulation, where it would interact with another object. Finally, we add a shoe object from Movi-C to this scene. Figure 3 shows a rollout from the SDF-Sim model of a shoe

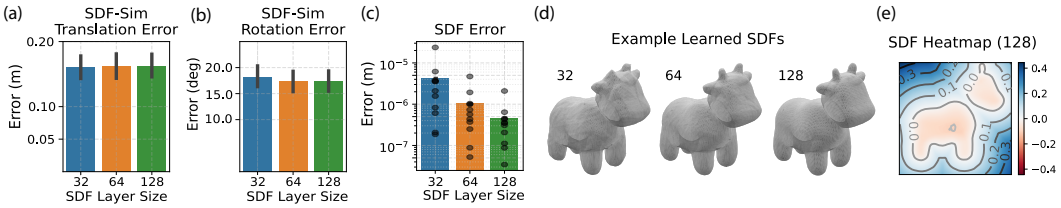


Figure 9: Ablation on learned SDF model sizes: 8 layer MLPs with layer sizes of 32, 64, or 128 hidden units. (a, b) Translation and rotation error for SDF-Sim trained on Movi-B with different SDF sizes. (c) Mean squared error of the predicted SDF estimates near the surface. (d) Visualisations of the cow shape from Movi-B with different SDF sizes. (e) A cross-section of the learned SDF field for the Movi-B cow shape.

falling on top of the vase and the table SDF extracted by VolSDF. With its over 80k nodes, this mesh exceeds the limits of FIGNet baseline that runs out of memory. In contrast, SDF-Sim can capture the nuanced interactions between the objects, which is particularly evident in frames 50 to 70; see the video on the website. This experiment demonstrates that SDF-Sim can generalize to new scenes, despite being trained on synthetic data of small scenes.

4.5 Ablation of learned SDF quality

We investigate the quality of the estimated SDF values and their impact on simulator performance. We train SDF using MLPs with 32, 64 and 128 units per layer, and 8 layers total. Figure 9(c) shows that larger SDF models help to improve the quality of estimated SDFs. Overall, the learned SDF models of all three layer sizes can reproduce the shape, although the reconstruction from a smallest 32-unit model is more coarse (Figure 9(d), see Figure S8 for more examples). However, Figure 9(a,b) shows that SDF-Sim accuracy is similar across different SDF MLP sizes, suggesting that the learned simulator can learn to compensate for noise in the estimated SDFs. We provide additional metrics on SDF accuracy, SDF gradients and SDF projected closest points in section D.2. We also provide the comparison of memory footprint of SDF versus storing meshes in section D.5.

5 Related Work

Dynamics over learned SDFs Overall, using implicit shapes, e.g., learned SDFs, for simulation has been explored only in limited settings. DiffSDFSim [35] and DANO [12] use learned object shapes (SDFs and NeRFs respectively) in combination with an analytic contact model on a scene with a single object. Argila et al. [5] and Qiao et al. [30] model the interaction of a learned SDF with a cloth via an analytic simulator. In the space of learned simulators, FIGNet* [19] uses a GNN-based model on a 3D scene reconstructed via NeRF and converted into a mesh. Driess et al. [15] learn latent dynamics over NeRFs. Whitney et al. [41] convert a NeRF 3D scene into a set of particles and use them in a GNN simulator. Unlike these pipelines, SDF-Sim avoids the costly simulation of large meshes or sets of particles and operates directly over the SDFs. Mani et al. [23] use learned SDFs to represent large rigid scenes for the sake of efficiency and use the SDF values directly in the GNN simulator. However, this model is designed to use only a *single* SDF per scene. In contrast, we focus on more challenging interactions of many rigid objects, that is much more prevalent in real applications.

Learning SDFs from vision Learned SDFs started to gain popularity for reconstructing an 3D surfaces from a sequence of images. Unlike NeRFs [7], SDFs are more likely to represent closed, watertight, smooth surfaces. Wang et al. [39], Yariv et al. [43], Yu et al. [44] train an MLP to represent an SDF of a single object. Other works train a single generative model for many SDF shapes, allowing to amortise the training cost, generate new shapes and perform shape completion from partial inputs [9, 10, 28, 34]. Recent works also offer ways to use SDFs for deformable objects [42], articulated objects [24], or to edit SDFs [38].

One can also reconstruct SDFs from other modalities, e.g., [24, 40] demonstrate how to reconstruct a new unseen object as an SDF using an RGB-D video from a single camera view. SDFs can be

https://doi.org/10.52202/079017-3997

recovered for large room-scale scenes [6] or even in real time as the camera moves through the space [27], or from noisy point clouds, e.g., from Lidar [21, 22]. We note that although we can leverage the SDFs extracted by these methods, these works pursue a different goal and focus only on accurate reconstruction of the 3D surface and do not aim to use these SDFs for simulation.

6 Discussion and Limitations

We introduce a rigid-body simulator, SDF-Sim, that uses learned SDFs to represent the object shapes and employs a graph network to learn dynamics over these objects. We re-design the construction of the input graph for the simulator to make use of SDFs and reduce the number of inter-object edges from quadratic to linear, unlocking the ability to scale to large scenes. We provide detailed study on both small-scale benchmarks and large scenes with up to 50x more objects than seen in the training. Additionally, we investigate the impact of SDF quality on the simulation results. Our work demonstrates, for the first time in the learning simulation literature, that we can use these simulators on scenes with hundreds of objects and up to 1.1 million nodes and produce realistic rollouts, potentially unlocking applications to virtual reality, film, robotics, and more.

Limitations One limitation of SDF-Sim is that it requires training an SDF MLP for every new shape we want to include in a simulation. A promising future direction is to train one model on a dataset of many shapes, amortising the cost of training SDFs, similarly to [28, 34]. We also note that SDF-Sim has slightly higher error on small-scale datasets (section 4.1), although SDF-Sim is closer to the ground-truth on large scenes than FIGNet* (section 4.2). We believe it is justifiable to trade a slight increase in model error in exchange for the ability to run on larger scenes than previously possible. We believe that SDF-Sim is therefore best suited for the applications that favour scaling over physical accuracy, such as animation and film. Finally, in this paper we only focus on rigid-rigid interactions, but SDFs can be extended to incorporate deformable or articulated objects [24, 42], fluids [23], and even a mix of SDFs, meshes, and particles.

Broader impact We see our work as a step in making applications that require simulation, such as augmented reality or animation, more accessible to everyday consumers. As such, our work allows us to directly use 3D assets filmed on a mobile phone, and simulate large scenes using a commodity hardware with a single GPU, without requiring a specialized skill of creating meshes appropriate for simulation. Overall, we see SDF-Sim as an important step towards learning physically realistic dynamics models from real-world data, and for enabling physical reasoning over complex, large-scale physical scenes.

Acknowledgements

The authors would like to thank Ladislav Rampášek for immense help on proof-reading the paper and revising the notation for the final paper submission.

References

[1] Brian Acosta, William Yang, and Michael Posa. Validating robotics simulators on real-world impacts. *IEEE Robotics and Automation Letters*, 7:6471–6478, 2021. URL https://api.semanticscholar.org/CorpusID:238253129. 2

[2] Kelsey Allen, Tatiana Lopez-Guevara, Kimberly L Stachenfeld, Alvaro Sanchez Gonzalez, Peter Battaglia, Jessica B Hamrick, and Tobias Pfaff. Inverse design for fluid-structure interactions using graph network simulators. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 13759–13774. Curran Associates, Inc., 2022. 2

[3] Kelsey R Allen, Tatiana Lopez Guevara, Yulia Rubanova, Kim Stachenfeld, Alvaro Sanchez-Gonzalez, Peter Battaglia, and Tobias Pfaff. Graph network simulators can learn discontinuous, rigid contact dynamics. In Karen Liu, Dana Kulic, and Jeff Ichnowski, editors, *Proceedings of The 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pages 1157–1167. PMLR, 14–18 Dec 2023. URL https://proceedings.mlr.press/v205/allen23a.html. 2

[4] Kelsey R. Allen, Yulia Rubanova, Tatiana Lopez-Guevara, William Whitney, Alvaro Sanchez-Gonzalez, Peter Battaglia, and Tobias Pfaff. Learning rigid dynamics with face interaction graph networks. In *International Conference on Learning Representations*, 2023. 1, 3, 5, 6, 7, 14, 15, 18

[5] Hugo Bertiche Argila, Meysam Madadi, and Sergio Escalera. Neural implicit surfaces for efficient and accurate collisions in physically based simulations. *ArXiv*, abs/2110.01614, 2021. URL https://api.semanticscholar.org/CorpusID:238354360. 4, 9

[6] Dejan Azinović, Ricardo Martin-Brualla, Dan B Goldman, Matthias Nießner, and Justus Thies. Neural rgb-d surface reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6290–6301, June 2022. 10

[7] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Mip-NeRF 360: Unbounded anti-aliased neural radiance fields. *CVPR*, 2022. 3, 4, 8, 9, 16

[8] 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. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018. 2

[9] Yen-Chi Cheng, Hsin-Ying Lee, Sergey Tulyakov, Alexander G Schwing, and Liang-Yan Gui. SDFusion: Multimodal 3d shape completion, reconstruction, and generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4456–4465, 2023. 9

[10] Gene Chou, Yuval Bahat, and Felix Heide. Diffusion-sdf: Conditional generative modeling of signed distance functions. 2023. 9

[11] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, oct 1976. ISSN 0001-0782. doi: 10.1145/360349.360354. URL https://doi.org/10.1145/360349.360354. 3, 4, 16

[12] Simon Le Cleac'h, Hong Yu, Michelle Guo, Taylor A. Howell, Ruohan Gao, Jiajun Wu, Zachary Manchester, and Mac Schwager. Differentiable physics simulation of dynamics-augmented neural objects. *IEEE Robotics and Automation Letters*, 8:2780–2787, 2022. URL https://api.semanticscholar.org/CorpusID:252967901. 9

[13] Erwin Coumans. Bullet physics simulation. In *ACM SIGGRAPH 2015 Courses*, page 7, 2015. 1

[14] Danny Driess, Jung-Su Ha, Marc Toussaint, and Russ Tedrake. Learning models as functionals of signed-distance fields for manipulation planning. In *5th Annual Conference on Robot Learning*, 2021. URL https://openreview.net/forum?id=FS30JeiGG3h. 2

[15] Danny Driess, Zhiao Huang, Yunzhu Li, Russ Tedrake, and Marc Toussaint. Learning multi-object dynamics with compositional neural radiance fields. In *6th Annual Conference on Robot Learning*, 2022. URL https://openreview.net/forum?id=qUvTmyGpnm7. 9

[16] Klaus Greff et al. Kubric: a scalable dataset generator. 2022. 6, 14

[17] Peide Huang, Xilun Zhang, Ziang Cao, Shiqi Liu, Mengdi Xu, Wenhao Ding, Jonathan Francis, Bingqing Chen, and Ding Zhao. What went wrong? closing the sim-to-real gap via differentiable causal discovery. In Jie Tan, Marc Toussaint, and Kourosh Darvish, editors, *Proceedings of The 7th Conference on Robot Learning*, volume 229 of *Proceedings of Machine Learning Research*, pages 734–760. PMLR, 06–09 Nov 2023. URL https://proceedings.mlr.press/v229/huang23c.html. 2

[18] Yunzhu Li, Jiajun Wu, Jun-Yan Zhu, Joshua B Tenenbaum, Antonio Torralba, and Russ Tedrake. Propagation networks for model-based control under partial observation. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 1205–1211. IEEE, 2019. 7

[19] Tatiana Lopez-Guevara, Yulia Rubanova, William F. Whitney, Tobias Pfaff, Kimberly Stachenfeld, and Kelsey R. Allen. Scaling face interaction graph networks to real world scenes, 2024. 1, 5, 6, 9, 14

 https://doi.org/10.52202/079017-3997

[20] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Seminal graphics: pioneering efforts that shaped the field*, pages 347–353. 1998. 17

[21] Baorui Ma, Zhizhong Han, Yu-Shen Liu, and Matthias Zwicker. Neural-pull: Learning signed distance function from point clouds by learning to pull space onto surface. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 7246–7257. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/ma21b.html. 10

[22] Baorui Ma, Yu-Shen Liu, and Zhizhong Han. Learning signed distance functions from noisy 3d point clouds via noise to noise mapping. In *International Conference on Machine Learning (ICML)*, 2023. 10

[23] Arjun Mani, Ishaan Preetam Chandratreya, Elliot Creager, Carl Vondrick, and Richard Zemel. Surfsup: Learning fluid simulation for novel surfaces. *arXiv preprint arXiv:2303.08128*, 2023. 2, 4, 9, 10

[24] Jiteng Mu, Weichao Qiu, Adam Kortylewski, Alan Yuille, Nuno Vasconcelos, and Xiaolong Wang. A-sdf: Learning disentangled signed distance functions for articulated shape representation. pages 12981–12991, 2021. 9, 10

[25] Matthias Müller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. Meshless deformations based on shape matching. *ACM transactions on graphics (TOG)*, 24(3):471–478, 2005. 6

[26] Adithyavairavan Murali, Arsalan Mousavian, Clemens Eppner, Adam Fishman, and Dieter Fox. CabiNet: Scaling neural collision detection for object rearrangement with procedural scene generation. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, May 2023. URL https://arxiv.org/abs/2304.09302. 4

[27] Joseph Ortiz, Alexander Clegg, Jing Dong, Edgar Sucar, David Novotny, Michael Zollhoefer, and Mustafa Mukadam. isdf: Real-time neural signed distance fields for robot perception. In *Robotics: Science and Systems*, 2022. 10

[28] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019. 4, 9, 10

[29] Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter Battaglia. Learning mesh-based simulation with graph networks. In *International Conference on Learning Representations*, 2021. 1, 2, 5, 6, 7

[30] Yi-Ling Qiao, Alexander Gao, Yiran Xu, Yue Feng, Jia-Bin Huang, and Ming C. Lin. Dynamic mesh-aware radiance fields. *ICCV*, 2023. 9

[31] Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning*, pages 8459–8468. PMLR, 2020. 3

[32] Johannes L Schonberger and Jan-Michael Frahm. Structure-from-motion revisited. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4104–4113, 2016. 4

[33] Nicholas Sharp and Alec Jacobson. Spelunking the deep: Guaranteed queries on general neural implicit surfaces via range analysis. *ACM Trans. Graph.*, 41(4), jul 2022. ISSN 0730-0301. doi: 10.1145/3528223.3530155. URL https://doi.org/10.1145/3528223.3530155. 3, 4

[34] Jaehyeok Shim, Changwoo Kang, and Kyungdon Joo. Diffusion-based signed distance fields for 3d shape generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 20887–20897, June 2023. 9, 10

[35] Michael Strecke and Joerg Stueckler. Diffsdfsim: Differentiable rigid-body dynamics with implicit shapes. In *3DV*, pages 96–105, 2021. URL https://doi.org/10.1109/3DV53792.2021.00020. 9

[36] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE, 2012. 1

[37] Dylan Turpin, Tao Zhong, Shutong Zhang, Guanglei Zhu, Eric Heiden, Miles Macklin, Stavros Tsogkas, Sven Dickinson, and Animesh Garg. Fast-grasp'd: Dexterous multi-finger grasp generation through differentiable simulation. In *ICRA*, 2023. 4

[38] Petros Tzathas, Petros Maragos, and Anastasios Roussos. 3d neural sculpting (3dns): Editing neural signed distance functions, 2023. 9

[39] Peng Wang, Lingjie Liu, Yuan Liu, Christian Theobalt, Taku Komura, and Wenping Wang. Neus: Learning neural implicit surfaces by volume rendering for multi-view reconstruction. *NeurIPS*, 2021. 9

[40] Bowen Wen, Jonathan Tremblay, Valts Blukis, Stephen Tyree, Thomas Muller, Alex Evans, Dieter Fox, Jan Kautz, and Stan Birchfield. BundleSDF: Neural 6-DoF tracking and 3D reconstruction of unknown objects. *CVPR*, 2023. 9

[41] William F. Whitney, Tatiana Lopez-Guevara, Tobias Pfaff, Yulia Rubanova, Thomas Kipf, Kimberly Stachenfeld, and Kelsey R. Allen. Learning 3d particle-based simulators from rgb-d videos, 2023. 9

[42] Youngsun Wi, Peter R. Florence, Andy Zeng, and Nima Fazeli. Virdo: Visio-tactile implicit representations of deformable objects. *2022 International Conference on Robotics and Automation (ICRA)*, pages 3583–3590, 2022. URL https://api.semanticscholar.org/CorpusID:246473027. 9, 10

[43] Lior Yariv, Jiatao Gu, Yoni Kasten, and Yaron Lipman. Volume rendering of neural implicit surfaces. In *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021. 4, 8, 9, 16

[44] Zehao Yu, Songyou Peng, Michael Niemeyer, Torsten Sattler, and Andreas Geiger. Monosdf: Exploring monocular geometric cues for neural implicit surface reconstruction. *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. 9

[45] Meng Zhang, Duygu Ceylan, Tuanfeng Y. Wang, and Niloy Jyoti Mitra. Dynamic neural garments. *ACM Transactions on Graphics (TOG)*, 40:1 – 15, 2021. URL https://api.semanticscholar.org/CorpusID:232013515. 3

[46] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018. 22

 https://doi.org/10.52202/079017-3997

A Datasets and custom simulations

A.1 Kubric dataset and evaluations [16]

We use Movi-B and Movi-C datasets from Kubric [16] for small-scale evaluation and fair comparison to baselines [4] and [19]. Simulations in Movi-B and Movi-C datasets contain between 3 and 10 objects per scene. In all simulations objects are thrown on the floor towards the center of the scene. The simulations consist of 100 time steps, or 2 seconds of the simulation time. See the visualisations of Movi-B shapes in Figure S8 and selected shapes from Movi-C in Figure S7.

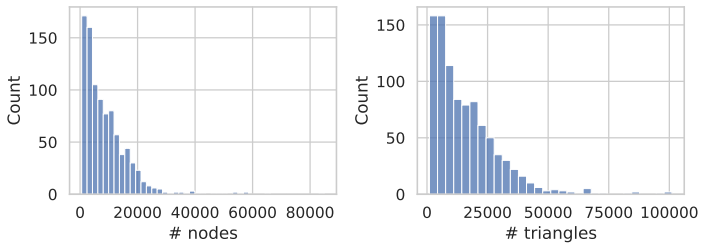
In both Movi-B and Movi-C we use 1500 trajectories for training, 100 for validation and 100 for test, as provided in the original Kubric dataset. Kubric dataset is distributed according to the Apache 2.0 license*.

	# object types	# nodes per object		# triangles per object	
		mean	max	mean	max
Movi-B	11	3,483	13,952	6,938	28,224
Movi-C	930	8,982	84,950	15,955	101,142

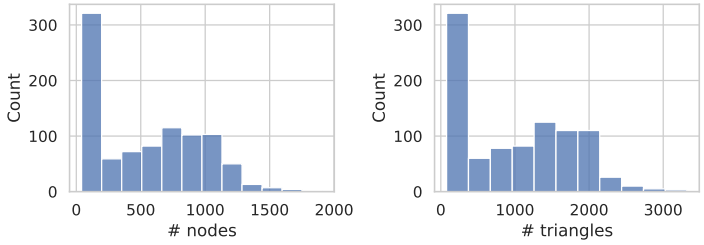
Table 1: Sizes of the **detailed meshes** used to train SDFs.

	# object types	# nodes per object		# triangles per object	
		mean	max	mean	max
Movi-B	11	453	1,142	841	2,160
Movi-C	930	543	1,908	1,002	3,324

Table 2: Sizes of the **collision meshes** used for simulation.



(a) Detailed meshes used for SDF training



(b) Collision meshes used for simulation

Figure 10: Distribution of mesh sizes used in Movi-C (930 objects).

A.2 Large scenes and vision

For evaluations on large-scale scenes, we create several custom scenes in Blender with large number of objects. For each simulation, we arrange the objects in a stacked formation above the floor, to

*https://github.com/google-research/kubric/blob/main/LICENSE

create a large number of collisions. Note that modelling a falling stack of objects is an extremely hard task even for classic simulators due to a large number of collisions.

Spheres-in-Bowl. For evaluations in section 4.2, we create a scene with variable number of spheres (from 1 to 512) falling into a bowl. Spheres are objects from Movi-B, with 64 nodes each, resized to 0.52 meters in diameter. A bowl is an object from Movi-C.

Falling shoes (Figure 2 top). We use 300 identical shoe objects from Movi-C. This simulation contains 851k nodes and 200 frames

Falling knots (Figure 2 middle). We use 270 torus-knots from Movi-B. The resulting simulation has 384k nodes and 200 frames.

Heaps of Stuff (Figure 2 bottom). We use 380 various objects from Movi-C . The scene consists of 1.1M nodes and is simulated for 400 time steps.

B SDF-Sim simulator architecture and training

Architecture Following [4], we use a graph network with node and edge updates without global pooling. For node and edge updates, we use MLPs with 2 hidden layers and 128 units each with LayerNorm. GNN consists of 10 message-passing steps.

We add a small amount gaussian noise to the inputs in $\mathcal{N}(0, 0.003)$ to help the network to correct inaccuracies from predictions in the previous steps and guarantee stable rollouts.

We normalise both inputs and outputs to zero-mean and unit-variance. The input graph is constructed using a history of positions for each node for the three latest time steps $[\mathbf{n}_k^{t-2}, \mathbf{n}_k^{t-1}, \mathbf{n}_k^t]$ that we convert to a history of two previous velocities $[\mathbf{n}_k^{t-1} - \mathbf{n}_k^{t-2}, \mathbf{n}_k^t - \mathbf{n}_k^{t-1}]$. The model predicts the acceleration for each node $\hat{\mathbf{a}}_k^{t+1}$.

Loss We train the model on 1-step prediction task. To construct a loss, we estimate ground-truth accelerations via finite difference: $\mathbf{a}_k^{t+1} = (\mathbf{n}_k^{t+1} - \mathbf{n}_k^t) - (\mathbf{n}_k^t - \mathbf{n}_k^{t-1}) = \mathbf{n}_k^{t+1} - 2\mathbf{n}_k^t + \mathbf{n}_k^{t-1}$. We use mean-squared error loss on the accelerations per node (normalised as mentioned above) to train the model. Note that the loss is computed per-node rather than per object.

Generating a rollout To generate a rollout of multiple steps, we convert the predicted the per-node accelerations $\hat{\mathbf{a}}_k^{t+1\dagger}$ into the next simulation state (positions and translations of the objects). To do so, we estimate the positions of each node at the next time point as $\hat{\mathbf{n}_k}^{t+1} = \hat{\mathbf{a}}_k^{t+1} + 2\mathbf{n}_k^t - \mathbf{n}_k^{t-1}$. Finally, we use shape matching, similarly to [4] to estimate the positions and rotation of the objects from the node positions.

Metrics Following [4], we report translation and rotation error between the predicted and ground-truth rollouts of 50 steps. For translation, we report root mean-squared error over the object translation $\mathbf{o}_i^t$. For rotation, we report the mean angle between the predicted and ground-truth object rotations. Translation vector and rotation quaternions per object are computed from predicted node positions via shape matching, as described above. As in prior work [4], translation and rotation accuracy is reported on rollouts of 50 time steps, while memory and runtime metrics are reported on the full rollouts of 100 steps.

Training time and hardware We train the network for 1M steps on 8 A100 GPUs devices with a batch size of 8. We use Adam optimizer, and an an exponential learning rate decay from 1e-3 to 1e-4. The approximate training time is 5 days.

[†]Here we use a "hat" $\hat{\cdot}$ notation to denote the quantities that are estimated or predicted as opposed to the ground-truth values.

https://doi.org/10.52202/079017-3997

C Learning SDFs

C.1 Training SDFs from meshes

For our experiments, we pre-train the SDFs on each object that will be used in the simulation. For SDF training, we use the meshes in the canonical pose centered around the [0,0,0] point, rescaled so that the maximum vertex coordinate is 0.5. Thus, the entire object mesh is located within [-0.5, 0.5] interval for each axis.

Sampling query points We sample query points $\mathbf{y}$ on and near the surface of the mesh to use for SDF training. First, we take the mesh nodes $\mathcal{V}_S$ and sample more points on the surface of the mesh uniformly using `sample_points_uniformly` function in open3D library. We sample 10,000 points per sq. meter of the object surface area. Then, we add gaussian noise with $\sigma = 0.1$ to the surface points. We use both surface points and noisy points as query points for training. We re-sample the noisy points every 100 steps of training.

Computing ground-truth SDF We compute the ground-truth SDF d to the object mesh for all query points. First, for each query point $\mathbf{y}$, we use a classic Boundary Vector Hierarchy (BVH) [11] method to find the closest triangle on the object mesh and then calculate the closest point $\mathbf{y}^*$ on the triangle. The ground-truth distance from the point $\mathbf{y}$ to the mesh is $||\mathbf{y} - \mathbf{y}^*||$. Finally, we use the normal $\mathbf{N}_f$ of the closest triangle (pointing outside of the object) to estimate the SDF sign. We compute the dot product $\mathbf{N}_f \cdot (\mathbf{y} - \mathbf{y}^*)$. If the dot product is positive, the point is outside the object and the SDF should have a positive sign, and if it is negative, it is inside the object and its SDF should have a negative sign. The ground-truth SDF d is therefore $||\mathbf{y} - \mathbf{y}^*||$ multiplied by the sign.

Training SDF We train the SDF function $f_\theta(\mathbf{y})$ by supervising it using the ground-truth SDF d. We use the L2 loss on the signed distances $(d - f_\theta(\mathbf{y}))^2$, as well as the L2 loss on the absolute distances $(|d| - |f_\theta(\mathbf{y})|)^2$. The reason is that we found that sign estimation can be unstable if vector $(\mathbf{y} - \mathbf{y}^*)$ is close to being orthogonal to the triangle normal vector $\mathbf{n}_f$. Therefore, we use the loss on *unsigned* distances on all query points, but the loss on the *signed* distances is only used on the points where the sign can be reliably estimated. Specifically, we use *signed*-distance loss on the points where the angle between $(\mathbf{y} - \mathbf{y}^*)$ and $\mathbf{n}_f$ is less than 75 degrees or more than 105 degrees.

SDF architecture Unless otherwise stated, the learned SDFs consisted of MLPs with 8 layers and 128 hidden units. The models were trained for 400K steps and a learning rate of 10^-5 for Kubric Meshes. For the Garden Vase scene, MLPs with 8 layers and 256 units were used, but training details were otherwise the same. For quantitative results in Figure 6, we run the all the model with 3 seeds and report the mean and the 95% confidence interval.

SDF training time and hardware For all meshes in Movi-C, we trained the MLPs for 400k iterations, which requires 18 hours of training time on average on a single V100 GPU. It is likely that this training time could be reduced, as we did not aim to minimize the computational cost of the SDFs in this work. Instead, we chose to train the SDF to maximize the quality for complex Movi-C meshes that have 8,982 nodes and 15,955 triangles on average.

Other approaches tested In NeRF literature a common regularisation to train SDFs is the Eikonal regularisation that enforces unit-norm of the SDF gradient. However, we found that Eikonal regularisation leads to over-smoothed sphere-like shapes and did not use it in our training. Furthermore, we find that our approach of directly fitting the distances already results in unit-norm-gradient and does not require an additional regularisation term. We also tested another common technique of periodic encodings as the MLP inputs, however, we found that it leads to checkerboard-like artifacts for points outside of the object and did not use this method.

C.2 Training SDFs from vision

For an experiment in section 4.4, we take a Garden Scene from [7], represented as a set of 2D images with a 360-degree view of the scene. We train a VolSDF model [43] to distill an SDF from this scene. VolSDF leverages a volumetric representation that explicitly encodes the volume density as a

transformed SDF. This explicit formulation provides an inductive bias over the geometry of an object and a built-in preference for capturing smooth and clean shapes.

VolSDF already represents a learned SDF and, in principle, can be directly used in SDF-Sim. However, to simplify the pipeline and have a consistent SDF architecture for all objects, we train another learned SDF, as described in Section C. To do so, we extract a high quality mesh (with over 80k vertices) of the table and the vase from VolSDF using the Marching Cubes algorithm [20]. Then we train a learned SDF with the architecture described in section C and use it in the SDF-Sim.

 https://doi.org/10.52202/079017-3997

D Additional results

D.1 Quantitative Results on small benchmark datasets.

Movi-B

Model	Translation RMSE (m)	Rotation Err (deg)
DPI	0.368 ± 0.057	26.928 ± 2.740
MGN-LargeRadius	0.460 ± 0.045	26.342 ± 1.397
MGN	0.538 ± 0.035	26.914 ± 0.783
FIGNet	0.14 ± 0.0087	**15.0** ± 0.675
FIGNet*	**0.13** ± 0.008	15.94 ± 0.8849
SDF-Sim (ours)	0.16 ± 0.0124	18.03 ± 0.9433

Model	# Collision Edges	# Graph Edges	Peak Memory (MiB)	Runtime per step (ms)
DPI	2250.688 ± 65	-	-	0.145 ± 0.009
MGN-LargeRadius	1797.985 ± 60	-	-	0.218 ± 0.033
MGN	34.367 ± 4	-	-	0.175 ± 0.018
FIGNet	1360.99 ± 196	24569.59 ± 938	94.45 ± 2.9698	0.23 ± 0.0044
FIGNet*	1409.87 ± 267	8613.05 ± 512	63.83 ± 3.7489	0.22 ± 0.0077
SDF-Sim (ours)	**201.14** ± 10	**6195.58** ± 193	**38.8** ± 0.5711	**0.14** ± 0.004

Table 3: Quantitative comparison of SDF-Sim and the baselines on a **Movi-B** dataset. This table presents the numbers for the Figure 6 in the main text. The error bars show the 95% confidence interval from running all models with 3 random seeds. DPI, MGN-LargeRadius and MGN results are reported by [4]. Results marked as '-' were not reported by the previous work [4]

.

Movi-C

Model	Translation RMSE (m)	Rotation Err (deg)
DPI-Reimplemented*	OOM	
MGN-LargeRadius*	OOM	
MGN*	OOM	
FIGNet	OOM	
FIGNet*	**0.18** ± 0.0097	**19.68** ± 0.6483
SDF-Sim (ours)	0.24 ± 0.0151	23.15 ± 0.668

Model	# Collision Edges	# Graph Edges	Peak Memory (MiB)	Runtime per step (ms)
DPI-Reimplemented*	OOM			
MGN-LargeRadius*	OOM			
MGN*	OOM			
FIGNet	OOM			
FIGNet*	2220.96 ± 469	11502.18 ± 944	75.87 ± 9.587	0.30 ± 0.0163
SDF-Sim (ours)	**300.46** ± 34	**7661.17** ± 266	**43.35** ± 0.7466	**0.17** ± 0.0038

Table 4: Quantitative comparison of SDF-Sim and the baselines on a **Movi-C** dataset. This table presents the numbers for the Figure 6 in the main text. The error bars show the 95% confidence interval from running all models with 3 random seeds. DPI, MGN-LargeRadius and MGN and FIGNet baselines run OOM on the Movi-C simulations and are not reported.

.

D.2 Additional metrics on SDF quality

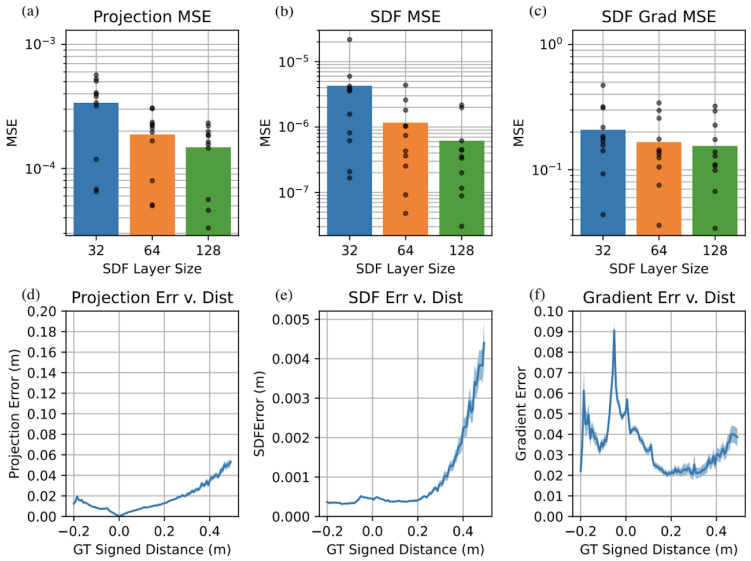


Figure S1: **Additional metrics on SDF quality**
(a-c) Metrics computed across learned SDF models of different sizes for different objects in Movi-B. Each black marker denotes a different Movi-B shape. For each metric, loss decreases as we increase the size of the model.
(a) MSE between the projected surface points predicted from the learned SDFs and the true closest point on the surface.
(b) MSE between the signed distance outputted by the learned SDF and the true signed distance.
(c) MSE between the gradient of the learned SDF and the gradient of the ground truth SDF function.

(d-f) SDF evaluation metrics as a function of distance from the object surface. SDF error, projection error and the gradient errors remain constant up to distance 0.2 from the object surface and start to increase afterwards. Generally all errors remain low in comparison to the object size of 0.5 meter.
(d) Error as distance between the learned SDF projected surface point and the true closest surface point, plotted as a function of signed distance from the surface.
(e) Error as the absolute difference in magnitude between the learned SDF prediction and ground truth SDF, plotted as a function of signed distance from the surface.
(f) Error as the cosine distance between the gradient of learned SDF and the ground truth SDF gradient.

https://doi.org/10.52202/079017-3997

D.3 Runtime versus number of nodes in simulation graph.

We measure the time to execute one step of simulation as a function of the input graph on different scenes in Movi-C dataset. For both SDF-Sim and FIGNet*, the runtime grows linearly with the number of the nodes in the graph. The runtime of SDF-Sim is consistently lower than FIGNet*. On large meshes with ∼8000 nodes and ∼18000 peak edges, SDF-Sim has a 52% smaller runtime compared to the FIGNet* baseline.

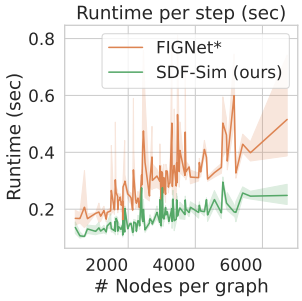


Figure S2: **Runtime per simulation step** w.r.t. the total number of nodes in the scene. The runtime evaluations were performed on the Movi-C test set on Nvidia A100 GPU. FIGNet baseline runs OOM on Movi-C and is not shown here.

D.4 Runtime comparison to Bullet simulator on Spheres-in-Bowl.

We compare the runtime of SDF-Sim, FIGNet and FIGNet* to the analytical simulator Bullet. SDF-Sim is substantially faster in comparison to FIGNet and FIGNet* and is similar to Bullet up to scenes with around 30 objects. Note that Bullet simulator is highly optimized for efficiency, and we do not expect learned simulators to match it in terms of runtime.

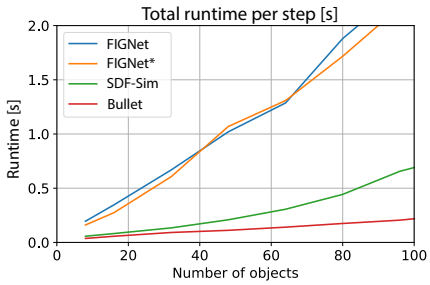


Figure S3: **Runtime comparisons**, measured per-step on Spheres-in-Bowl. X axis shows the number of spheres in Spheres-in-Bowl. SDF-Sim has similar runtime to Bullet on scenes up to 30 objects.

D.5 Memory of the SDF versus a mesh

A key advantage of learned SDFs is that they provide a compact representation of complex object shapes compared to meshes. In Figure S4, we compare the sizes of meshes from Movi-C dataset (red) to the sizes of the SDF model parameters (purple) in MiB, assuming both are stored as pickle files on disk. While the size of the meshes grows with the number of mesh nodes, memory requirements of the SDFs do not depend on the complexity of the object, as both simple and intricate shapes are modeled with the same size MLP. We see that our MLP with 8 layers of 128 units has similar memory demands to meshes with 15,000 nodes, which is 4x less memory than the largest meshes in Movi-C. Thus, SDF-Sim can potentially use less memory to store the object shapes: while FIGNet and FIGNet* require to store the entire object mesh, SDF-Sim needs only a compact SDF and a set surface nodes, which, as we will show in section D.7, can be relatively small.

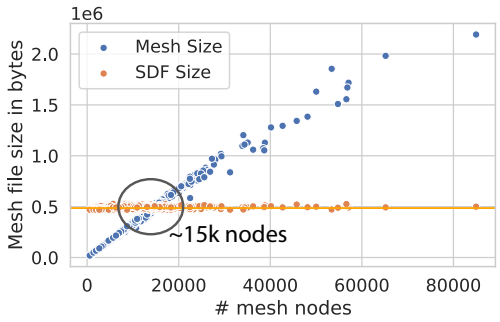


Figure S4: **Memory footprint of storing the mesh versus storing the SDF weights** w.r.t. the number of mesh nodes, evaluated on 930 objects from Movi-C.

D.6 Penetration and error comparison to Bullet

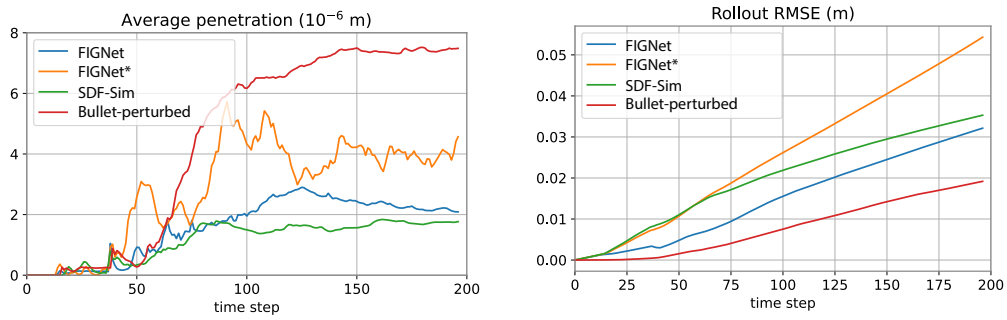


Figure S5: Penetration and Rollout RMSE on Spheres-In-Bow simulation. Bullet simulation is used as a ground-truth. This plot extends Figure 8 with an additional Bullet-perturbed baseline, where we add a tiny amount of gaussian noise to the initial states in Bullet simulation. **(Left)** SDF-Sim has *smaller* penetrations than Bullet, presumably because Bullet is optimized for speed instead. **(Right)** Both SDF-Sim and Bullet-perturbed eventually diverge from the original simulation, and the error of SDF-Sim is higher than Bullet-perturbed (as expected), but lower than FIGNet*.

https://doi.org/10.52202/079017-3997

D.7 Re-sampling object surface nodes using an SDF

Simulation accuracy heavily depends on the quality of the object shapes, typically a mesh. However, not all meshes are suitable for simulation. Object meshes, either created by artists or obtained by scanning a real-world object, often consist of thousands of nodes, and the nodes are concentrated around on the fine details of the shape (an example shown in Figure S6(a)). To be able to simulate the objects, the shapes are often approximated by hand-crafted convex *collision meshes* (Figure S6(b)). But such meshes might not always exist, and might not be optimal for GNN-based simulators like FIGNet. However, mesh-based simulators like FIGNet, typically use these collision meshes, as changing a mesh while retaining a connected surface is not easy.

In contrast, SDF-Sim does not require a connected mesh; it relies only on surface nodes and the learned SDF to perform a simulation step. This gives us full flexibility to choose the number and the location of graph nodes. Varying the number of surface nodes gives us a tighter handle controlling the tradeoff between accuracy, simulation speed and memory.

To create a new set of surface nodes, we can *sample* the surface nodes from a learned SDF. We initialize $K = N^3$ equally-spaced points in the volume of a cube which encompasses the object. We project these points to the object surface using Eq. 1, with minor filtering (more details on this sampling procedure described below). With this strategy, we get points that evenly cover the entire object surface (Figure S6(c)). Finally, we take the SDF-Sim simulator trained on Movi-C and replace each object's mesh nodes with the ones sampled from an SDF *at test time*. Figure 6 shows that with this sampling strategy, we can maintain the simulation accuracy using 73.5% fewer nodes (hence smaller memory) compared to SDF-Sim with Movi-C collision meshes, used elsewhere in the paper. Specifically, in Figure S6(d) the models with $K = 7^3$ and $K = 8^3$ (dark brown bars) have similar translation error (0.23) compared to the model with original node distribution (green bar, 0.24). However, in Figure S6(e) the model $K = 7^3$ requires only 934.19 nodes per scene on average (fourth orange column), while the model with the original Movi-C nodes needs 3530.13 nodes (green column) to perform the same simulation.

Here we perform this experiment at test time only, but it would be straightforward to also *train* the SDF-Sim with the nodes sampled from an SDF, and experiment with sampling strategies for optimal prediction performance. This creates the opportunity to use a learned SDF as the sole representation of shape without ever requiring a mesh.

Details on sampling the surface nodes from an SDF We start with a uniform 3D grid of NxNxN points, located within an interval $[-0.7, 0.7]$, to ensure that the grid is larger than the object (recall that the object meshes were rescaled to $[-0.5, 0.5]$ before training an SDF). Placing the query points on a grid ensures that the corresponding projected points would cover the entire object surface, including concave parts like inner side of a shoe or a knot. Then, we project these points on the surface: $\mathbf{y}^* = \mathbf{y} - f_\theta(\mathbf{y})\nabla f_\theta(\mathbf{y})$. We generally find that the trained SDF functions are fairly accurate and the projected points are close to the surface. Nevertheless, we additionally filter the points $f_\theta(\mathbf{y}^*) < 0.01$ to ensure that $\mathbf{y}^*$ are exactly on the surface as it otherwise would impact the accuracy of the simulation. Finally, some areas of the shape might have clusters of points. We additionally downsample projected points using `voxel_down_sample` function from open3D library [46], such that at most one point is present per voxel of size 0.025 to further reduce the number of points.

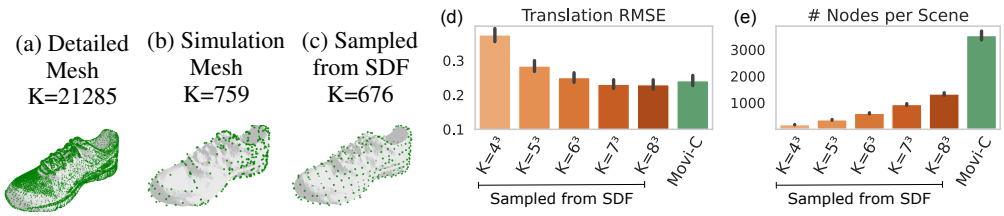


Figure S6: Shoe object from Movi-C, with vertices shown in green: (a) the original high-resolution mesh for rendering, (b) collision mesh for simulation, (c) nodes sampled on a grid and projected to the learned SDF). We can achieve favorable tradeoffs between error (d) and node count (e) by varying the dimensions K of the sample grid (orange bars). The green bar represents SDF-Sim that uses the nodes from the original Movi-C collision mesh, as used elsewhere in the paper.

D.8 Quantifying the source of SDF-Sim error

A natural question to ask is, where does the increased error (in comparison to FIGNet*) of SDF-Sim on Movi B/C comes from. The distances predicted by learned SDFs may not be perfectly accurate, because they come from a learned model. To quantify this, we compute the distances using a brute-force distance computation between query points and each triangle in the mesh. Then we train the simulator using these distances instead of learned SDFs. It is tricky to estimate the SDF sign using this approach (whether the point is inside/outside of the mesh), so we use the unsigned distances.

We ran an experiment where we train SDF-Sim on Movi-C with estimates of the *unsigned* distance computed from the mesh via BVH. We found that using learned SDFs versus pre-computed distances makes little difference in practice. We achieve similar Translation RMSE on Movi-C (0.23 with mesh-based UDFs versus 0.24 with learned SDF). While it is not a one-to-one comparison, this result does indicate that our method is relatively robust to SDF accuracy, and the learned simulator can learn to correct for minor inaccuracies.

https://doi.org/10.52202/079017-3997

E Learned SDF visualisations

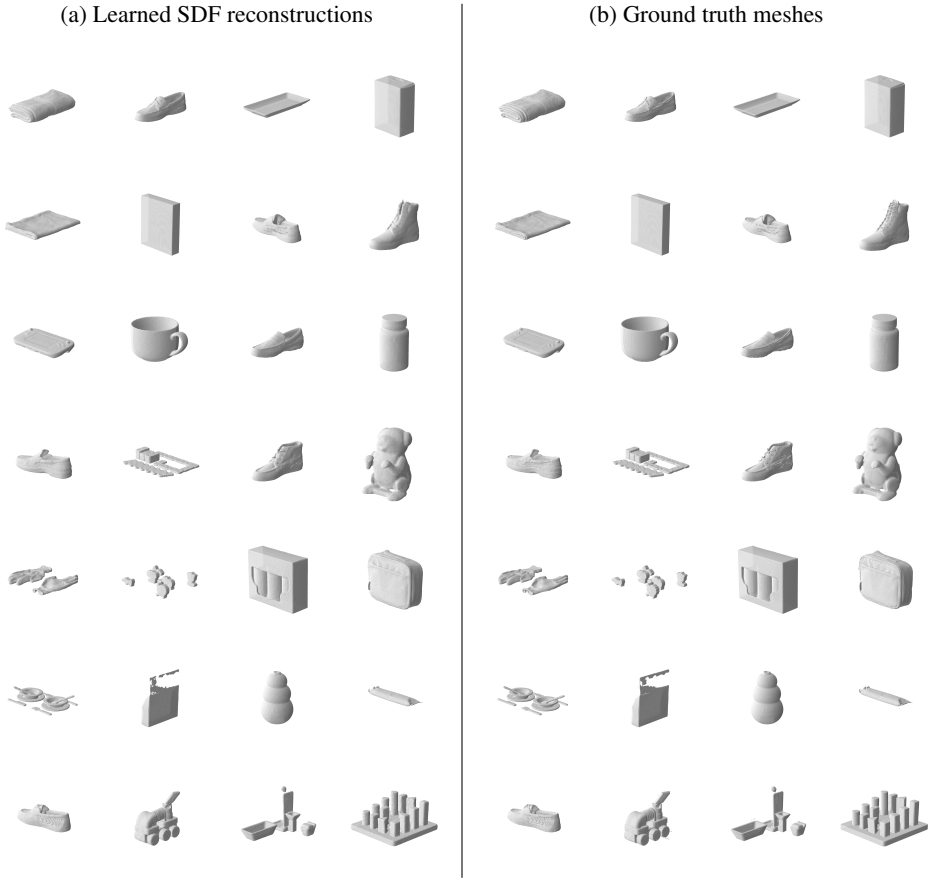


Figure S7: (a) Learned SDF reconstructions for randomly chosen Movi-C shapes. For visualisation purposes, we convert the object surface defined by a learned SDF: $\{\mathbf{y} : f_\theta(\mathbf{y}) = 0\}$ into a mesh using Marching Cubes and show this mesh here. (b) Corresponding ground-truth meshes.

https://doi.org/10.52202/079017-3997

E.1 SDF shapes through training

As the SDF network has to encode the distances only for a single object, we find that a simple MLP is sufficient to encode complex shapes to the necessary extent for the simulation. Figure S8(a) demonstrates that the most accurate shapes are at 400k iterations of training, however already after 4000 training steps the network is able to capture the outlines of most shapes. For simple shapes like cones or cubes, a model learns an accurate shape already by 8000 training iterations. Shapes with holes (like torus) can also be learned within the first 8000 iterations. Although in this paper we chose to train the SDFs for 400k iterations with a small learning rate for best results, these visualisations suggest that faster training for fewer iterations might be sufficient to use for simulation.

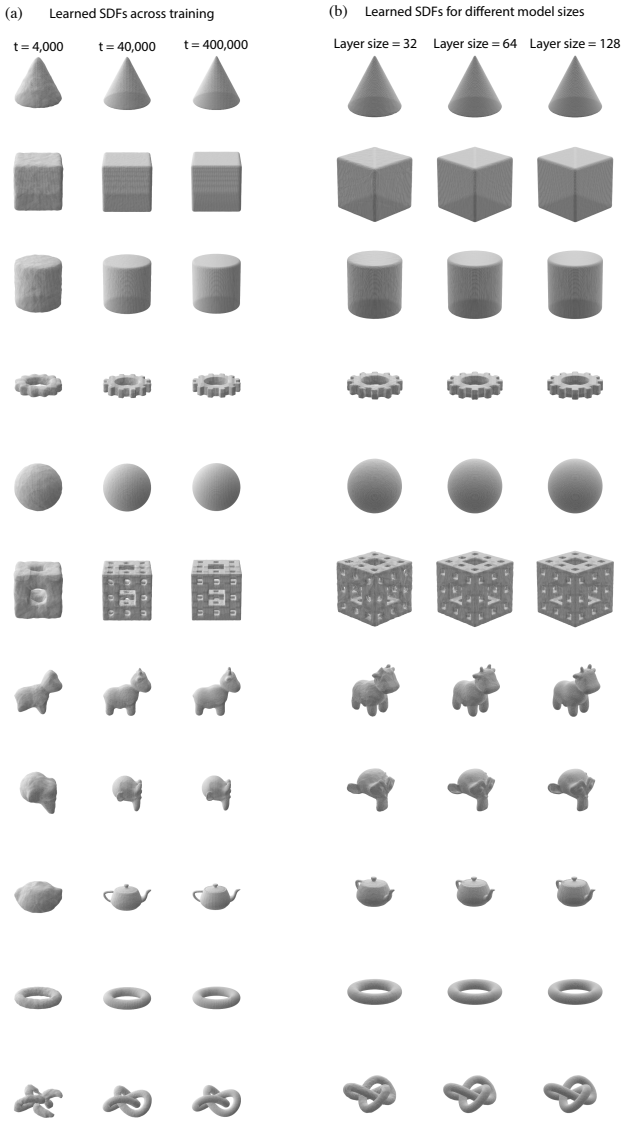


Figure S8: (a) Learned SDFs for Movi-B visualized at different points during training. (b) Learned SDFs for Movi-B visualized for different MLP sizes (different number of units per layer, all MLPs have 8 layers). For visualisation purposes in both figures, we convert the object surface defined by a learned SDF: $\{\mathbf{y} : f_\theta(\mathbf{y}) = 0\}$ into a mesh using Marching Cubes.

https://doi.org/10.52202/079017-3997

F Rollout Examples

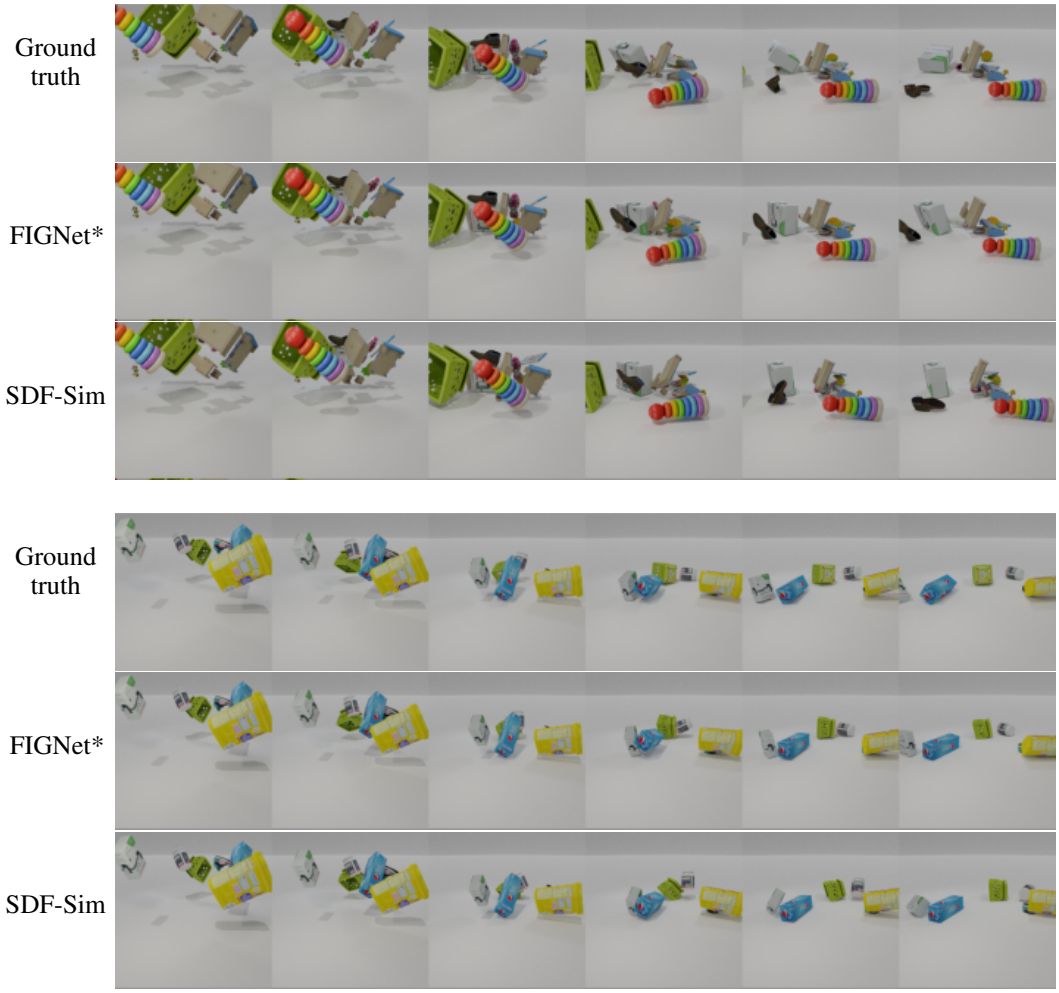


Figure S9: Rollout comparisons between baselines and SDF-Sim on Kubric Movi-C.

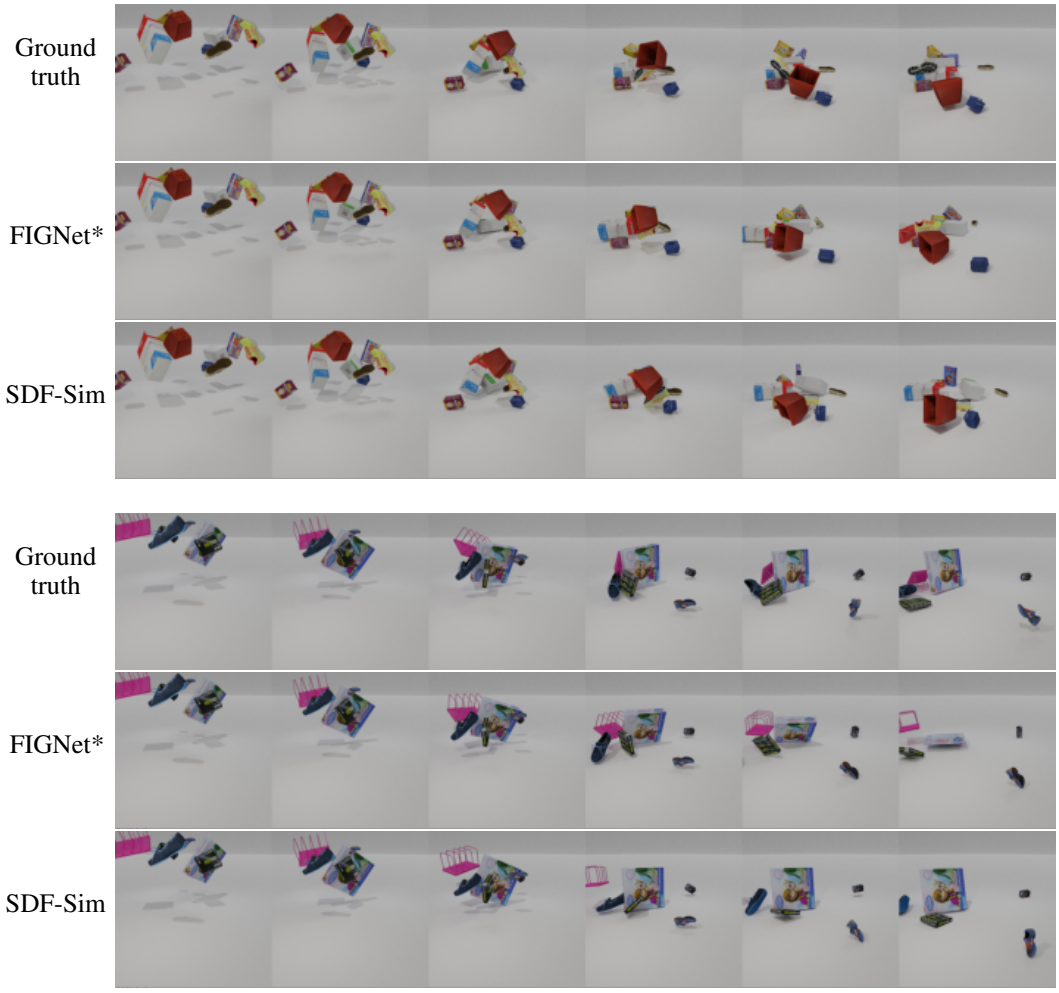


Figure S10: Rollout comparisons between baselines and SDF-Sim on Kubric Movi-C.

https://doi.org/10.52202/079017-3997

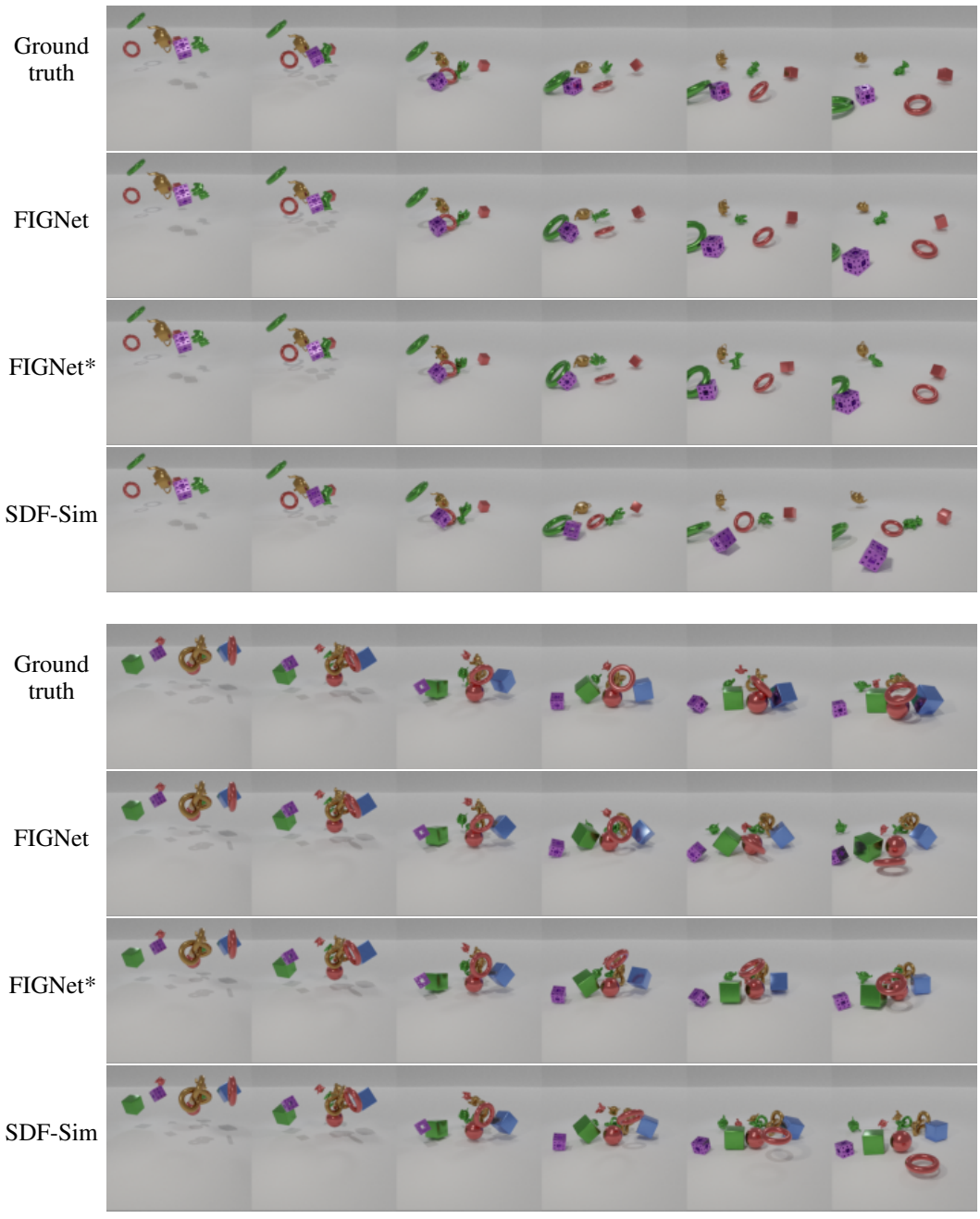


Figure S11: Rollout comparisons between baselines and SDF-Sim on Kubric Movi-B.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The main claims in the abstract and introduction reflect the scope of the paper and are supported by the experimental results.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The paper includes an explicit Limitations sections and states the assumptions (e.g. access to 3D meshes). The paper provides detailed investigations on scaling and efficiency.

3. **Theory Assumptions and Proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: The paper does not contain theoretical results.

4. **Experimental Result Reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper introduces a new model architecture. We provide the detailed description of the architecture, sufficient to reproduce the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: We provide a detailed description of the method sufficient to reproduce it. Open-source release of the code is not possible for this work.

6. **Experimental Setting/Details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: We provide the training details, data splits, type of optimizer, etc.

7. **Experiment Statistical Significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: We report the confidence intervals over multiple random seeds for our experiments.

8. **Experiments Compute Resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

https://doi.org/10.52202/079017-3997

Answer: [Yes]

Justification: We provide the information about the memory, runtime and hardware for our experiments.

9. **Code Of Ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics https://neurips.cc/public/EthicsGuidelines?

   Answer: [Yes]

   Justification: The authors reviewed the Code of Ethics and confirm that this work follows the code.

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: We provide an overview of the potential societal impact, although we do not foresee our work on physical simulation to have adversarial consequences.

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: To our best judgement, the presented approach (scalable 3D simulation) presents no risks for misuse.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Yes, the datasets and the models used as baselines are properly credited and the terms of use respected.

13. **New Assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [NA]

    Justification: The paper does not release new assets.

14. **Crowdsourcing and Research with Human Subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: Current paper does not use human participants and does not require an approval.

15. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: Current paper does not use human participants and does not require an approval.