UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Building a Simulation Platform for Embodied AI

Permalink

https://escholarship.org/uc/item/9p86p108

Author

Xiang, Fanbo

Publication Date

2024

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

Building a Simulation Platform for Embodied AI

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

in

Computer Science

by

Fanbo Xiang

Committee in charge:

Professor Hao Su, Chair
Professor Sicun Gao, Co-Chair
Professor Nikolay A Atanasov
Professor Laurel D Riek
Professor Michael Yip

2024

Copyright

Fanbo Xiang, 2024

All rights reserved.

The Dissertation of Fanbo Xiang is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2024

TABLE OF CONTENTS

Dissertation Approval Page iii

Table of Contents iv

List of Figures vii

List of Tables x

Acknowledgements xi

Vita xiii

Publications xiii

Abstract of the Dissertation xv

Chapter 1 Introduction 1
1.1 Embodied AI Background 1
1.1.1 Problem Formulation 1
1.1.2 Embodied AI Simulators 2
1.2 Challenges in Simulation Framework Construction 4
1.2.1 Engine Architecture 4
1.2.2 Physical Simulation 5
1.2.3 Rendering 6
1.2.4 Asset Acquisition 6
1.3 Overview of Techniques and Contributions 7

Chapter 2 Building a Simulator for Generalizable Articulated Object Manipulation . . 9
2.1 Introduction 9
2.2 Related Work 12
2.3 SAPIEN Simulation Environment 15
2.3.1 SAPIEN Engine 16
2.3.2 SAPIEN Asset 18
2.3.3 SAPIEN Renderer 19
2.3.4 Profiling Analysis 20
2.4 Tasks and Benchmarks 20
2.4.1 Robotic Perception 20
2.4.2 Robotic Interaction 24
2.5 PartNet-Mobility Annotation System 27
2.6 Movable Part Segmentation and Motion Recognition 30
2.7 Conclusion 33
2.8 Terminology 33

Chapter 3 Simulation Asset Acquisition by Neural Capturing 36
3.1 Introduction 36
3.2 Related Work 39
3.3 Neural Texture Mapping 42
3.3.1 Overview 42
3.3.2 Disentangled neural scene representation 43
3.3.3 Texture space and inverse texture mapping 45
3.3.4 Training neural texture mapping 45
3.3.5 Extension to reflectance fields 48
3.4 Implementation Details 48
3.4.1 Network details 48
3.4.2 Training details 49
3.5 Results 50
3.5.1 Configuration 50
3.5.2 View synthesis results on DTU scenes 52
3.5.3 Texture mapping and appearance editing 53
3.5.4 Extension to reflectance fields 55
3.6 Clarification on Cube map Visualization 56
3.7 Network Implementation Details 57
3.7.1 Network structure 57
3.7.2 Training details in initialization 57
3.8 Additional Results 58
3.8.1 Full quantitative comparison 58
3.8.2 Additional visual results 59
3.9 Conclusion 59

Chapter 4 Building a General Embodied AI Simulation Framework 61
4.1 Introduction 61
4.2 Engine Architecture 63
4.2.1 Entity-Component System 64
4.2.2 Builders and Asset Loaders 65
4.3 Multi-Physics Systems 66
4.3.1 PhysX Rigid Body Simulator 66
4.3.2 Deformable Body Simulators 71
4.4 Unified Rendering System 74
4.4.1 Visual Learning System Optimization: End-to-end GPU Execution 75
4.4.2 Ray-tracing and Bindless Rendering 80
4.4.3 Software Architecture 81
4.4.4 Active Stereo Sensor Simulation 84
4.4.5 LiDAR Simulation 85
4.5 Manipulation Environment Construction 90
4.5.1 Sample Environments 91
4.6 Software Package and Interfaces 94
4.6.1 SAPIEN Python library 94

4.6.2 Graphical User Interface 95
4.6.3 Virtual Reality 96
4.6.4 Extensions 97

Chapter 5 Conclusion 99

Bibliography 101

LIST OF FIGURES

Figure 2.1. Robot-object interaction in SAPIEN. 10

Figure 2.2. SAPIEN simulator overview. 15

Figure 2.3. Robotic interaction tasks enabled by SAPIEN 18

Figure 2.4. Movable part detection results. 21

Figure 2.5. Robotic interaction tasks supported by SAPIEN 24

Figure 2.6. Diverse manipulation tasks supported by SAPIEN. 28

Figure 2.7. PartNet-Mobility annotation interface. 29

Figure 2.8. Vision tasks supported by SAPIEN 33

Figure 3.1. NeuTex summary. 38

Figure 3.2. Overview of NeuTex architecture 40

Figure 3.3. Illustration of the effectivness of point cloud initialization and cycle loss . 46

Figure 3.4. Baseline comparisons on DTU scenes. 49

Figure 3.5. Visualization for the uniform texture space discovered by NeuTex 50

Figure 3.6. Texture editing results on DTU and our scenes. 51

Figure 3.7. NeuTex in reflectance fields setting. 53

Figure 3.8. Finetuning with cycle and render loss gives a smoother surface. 54

Figure 3.9. Illustration for the cube map projection. 56

Figure 3.10. Illustration for the cube map projection. 56

Figure 3.11. Network structure for the 4 networks in NeuTex 57

Figure 3.12. Comparison on the remaining DTU scenes. 59

Figure 3.13. Additional texture editing on DTU scenes. 59

Figure 4.1. Illustration for SAPIEN's Entity-Component System architecture to organize simulation objects. 63

Figure 4.2. Relations between SAPIEN's Scene, Entity, Component, and System. 64
Figure 4.3. Overview of PhysX rigid body features exposed by SAPIEN. 66
Figure 4.4. Illustration for SAPIEN's data-passing pipelines with NVIDIA PhysX. . . . 68
Figure 4.5. Global tensor API used by PhysX GPU system in SAPIEN. 70
Figure 4.6. Deformable manipulation environments supported by SAPIEN MPM system. 73
Figure 4.7. Two-way dynamic coupling between rigid and deformable physical systems 74
Figure 4.8. Comparison for common visual RL system setups. 76
Figure 4.9. Visualization for rendered RGB images (512×512) in Cartpole (left) and Replica (right). 77
Figure 4.10. Comparison for different render pipelines setups. 79
Figure 4.11. SAPIEN's Pose Synchronization Manager setup. 80
Figure 4.12. Resource layout and management for SAPIEN renderer. 82
Figure 4.13. Software architecture for SAPIEN renderer. 85
Figure 4.14. Comparison for our active-stereo depth simulation with rasterization and ray-tracing. 86
Figure 4.15. Common causes of LiDAR multi-path artifacts. 86
Figure 4.16. Visual comparison between real-captured and simulated point clouds. 88
Figure 4.17. Simulation and real-world setups for LiDAR camera policy sim-to-real . . . 90
Figure 4.18. Success rate and distribution for sim-to-real cube grasping. 90
Figure 4.19. Samples of ManiSkill v3 environments. 91
Figure 4.20. ManiSkill Cartpole peformance metrics. 93
Figure 4.21. ManiSkill AnymalC-Reach peformance metrics. 93
Figure 4.22. ManiSkill PickCube peformance metrics. 94
Figure 4.23. ManiSkill PickSingleYCB peformance metrics. 94

Figure 4.24. SAPIEN GUI setup example. 96

Figure 4.25. SAPIEN VR viewer. 97

LIST OF TABLES

Table 2.1. Comparison for SAPIEN to other simulation environments. 11

Table 2.2. Comparison of articulated part datasets. 14

Table 2.3. Statistics of PartNet-Mobility Dataset 18

Table 2.4. Movable part detection results. 20

Table 2.5. Motion recognition results. 22

Table 2.6. SAC results on door and drawer opening. 27

Table 2.7. Movable part segmentation results for all categories 31

Table 3.1. Average PSNR/SSIM for novel view synthesis on 4 held-out views on 5 DTU scenes. 50

Table 3.2. PSNR/SSIM for novel view synthesis quality on 4 held-out views on 5 DTU scenes. 60

Table 4.1. Simulation parameters supported by SAPIEN IPC system. 72

Table 4.2. Simulation parameters supported by SAPIEN MPM system. 73

Table 4.3. Environment speedups from rendering techniques. 77

Table 4.4. Our LiDAR simulation method demonstrate smaller discrepancies between simulated depth map and real-captured ones compared to clean depth maps. 88

Table 4.5. Platform compatibility for SAPIEN's features. 95

ACKNOWLEDGEMENTS

First and foremost, I am sincerely grateful to my PhD advisor, Prof. Hao Su, for the unwavering guidance, support, and encouragement, since my master's studies through my whole PhD career. Your vision into the future and belief in my potential have transformed this dissertation from an academic exercise into an impactful contribution to the field.

I am deeply grateful to my PhD committee members, Prof. Sicun Gao, Prof. Nikolay Atanasov, Prof. Laurel Riek and Prof. Michael Yip, for the insightful suggestions, which greatly help me refine this work.

I also wish to thank my collaborators, including colleagues and friends at universities and corporations. I especially want to acknowledge Jiayuan Gu, Tongzhou Mu, Xinyue Wei, Xiaodi Yuan, and Stone Tao, with whom I have closely collaborated in research projects. I would also like to thank my labmates Yuzhe Qin, Zhan Ling, Zhiao Huang, Ruoxi Shi, Yulin Liu, Xiaoshuai Zhang, and Linghao Chen from whom I obtained helpful insights for my research. I want to express my gratitude to Miles Macklin, Angel Chang, Kaichun Mo, Zexiang Xu, and Rui Chen, who provided valuable mentorship beyond UC San Diego which greatly contributes to my success.

Finally, I would like to express my deepest gratitude to my mother and father. Thank you for your unconditional love and endless support. Your love and belief in me have been my constant source of strength.

Chapter 2, in full, is a reprint of the material published in the 2020 Conference on Computer Vision and Pattern Recognition (CVPR): "SAPIEN: A Simulated Part-based Interactive Environment" (Fanbo Xiang, Yuzhe Qin, Kaichun Mo, Yikuan Xia, Hao Zhu, Fangchen Liu, Minghua Liu, Hanxiao Jiang, Yifu Yuan, He Wang, Li Yi, Angel X. Chang, Leonidas J. Guibas, Hao Su). The dissertation author was the primary investigator and author of this paper.

Chapter 3, in full, is a reprint of the material published in the 2021 Conference on Computer Vision and Pattern Recognition (CVPR): "NeuTex: Neural Texture Mapping for Volumetric Neural Rendering " (Fanbo Xiang, Zexiang Xu, Miloš Hašan, Yannick Hold-Geoffroy,

Kalyan Sunkavalli, Hao Su). The dissertation author was the primary investigator and author of this paper.

Chapter 4, in full, is a coauthored/unpublished technical report: "Building a General Embodied AI Simulation Framework" (Fanbo Xiang, Stone Tao, Xiaodi Yuan, Ang Li, Yulin Liu, Zhiao Huang, and Hao Su). The dissertation author was the primary investigator and author of this material.

VITA

2014-2018 Bachelor of Science, University of Illinois Urbana-Champaign

2018-2020 Master of Science, University of California San Diego

2020-2024 Doctor of Philosophy, University of California San Diego

PUBLICATIONS

[1] Weihang Chen, Jing Xu, Fanbo Xiang, Xiaodi Yuan, Hao Su, and Rui Chen. General-purpose sim2real protocol for learning contact-rich manipulation with marker-based visuotactile sensors. *IEEE Transactions on Robotics*, 40:1509–1526, 2024.

[2] Jiayuan Gu, Fanbo Xiang, Xuanlin Li, Zhan Ling, Xiqiang Liu, Tongzhou Mu, Yihe Tang, Stone Tao, Xinyue Wei, Yunchao Yao, Xiaodi Yuan, Pengwei Xie, Zhiao Huang, Rui Chen, and Hao Su. Maniskill2: A unified benchmark for generalizable manipulation skills. In *The Eleventh International Conference on Learning Representations*, 2023.

[3] Xiaoshuai Zhang, Rui Chen, Ang Li, Fanbo Xiang, Yuzhe Qin, Jiayuan Gu, Zhan Ling, Minghua Liu, Peiyu Zeng, Songfang Han, Zhiao Huang, Tongzhou Mu, Jing Xu, and Hao Su. Close the optical sensing domain gap by physics-grounded active stereo sensor simulation. *IEEE Transactions on Robotics*, 39(3):2429–2447, 2023.

[4] Pengwei Xie, Rui Chen, Siang Chen, Yuzhe Qin, Fanbo Xiang, Tianyu Sun, Jing Xu, Guijin Wang, and Hao Su. Part-guided 3d rl for sim2real articulated object manipulation. *IEEE Robotics and Automation Letters*, 8(11):7178–7185, 2023.

[5] Ziyuan Liu, Wei Liu, Yuzhe Qin, Fanbo Xiang, Minghao Gou, Songyan Xin, Maximo A. Roa, Berk Calli, Hao Su, Yu Sun, and Ping Tan. Ocrtoc: A cloud-based competition and benchmark for robotic grasping and manipulation. *IEEE Robotics and Automation Letters*, 7(1):486–493, 2022.

[6] Kaichun Mo, Yuzhe Qin, Fanbo Xiang, Hao Su, and Leonidas Guibas. O2o-afford: Annotation-free large-scale object-object affordance learning. In Aleksandra Faust, David Hsu, and Gerhard Neumann, editors, *Proceedings of the 5th Conference on Robot Learning*, volume 164 of *Proceedings of Machine Learning Research*, pages 1666–1677. PMLR, 08–11 Nov 2022.

[7] Tongzhou Mu, Zhan Ling, Fanbo Xiang, Derek Yang, Xuanlin Li, Stone Tao, Zhiao Huang, Zhiwei Jia, and Hao Su. Maniskill: Generalizable manipulation skill benchmark with large-scale demonstrations. In *NeurIPS Datasets and Benchmarks*, 2021.

[8] Fanbo Xiang, Zexiang Xu, Milos Hasan, Yannick Hold-Geoffroy, Kalyan Sunkavalli, and Hao Su. Neutex: Neural texture mapping for volumetric neural rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7119–7128, June 2021.

[9] Anpei Chen, Zexiang Xu, Fuqiang Zhao, Xiaoshuai Zhang, Fanbo Xiang, Jingyi Yu, and Hao Su. Mvsnerf: Fast generalizable radiance field reconstruction from multi-view stereo. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 14124–14133, October 2021.

[10] Fanbo Xiang, Yuzhe Qin, Kaichun Mo, Yikuan Xia, Hao Zhu, Fangchen Liu, Minghua Liu, Hanxiao Jiang, Yifu Yuan, He Wang, Li Yi, Angel X. Chang, Leonidas J. Guibas, and Hao Su. Sapien: A simulated part-based interactive environment. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

ABSTRACT OF THE DISSERTATION

Building a Simulation Platform for Embodied AI

by

Fanbo Xiang

Doctor of Philosophy in Computer Science

University of California San Diego, 2024

Professor Hao Su, Chair
Professor Sicun Gao, Co-Chair

Embodied AI, the study of intelligent agents that interact with their surroundings to solve challenging tasks, is a crucial step toward building robotic agents that match or surpass human capabilities. While simulators offer scalable, reproducible, and safe methods for studying embodied AI, existing simulation frameworks often fall short in terms of features, extensibility, and the availability of simulation assets. In this dissertation, I present our efforts to advance simulation frameworks that improve performance and reduce sim-to-real domain gaps, as well as our methods for collecting 3D assets through annotation and 3D capture.

One of the most significant challenges in developing simulation environments for embod-

ied AI and manipulation tasks is the scarcity of high-quality simulation assets. To this end, we create a large-scale articulated object dataset, PartNet-Mobility, and develop an accompanying simulator, SAPIEN, to support the simulation and rendering for articulated body manipulation environments. PartNet-Mobility and SAPIEN have enabled research on various problems, including part motion prediction and generalizable articulation manipulation.

The lack of content in embodied AI simulations has motivated our exploration of 3D capture techniques to obtain assets by capturing real-world objects. We have developed the first neural capture technique capable of disentangling 2D appearance maps from 3D geometry, bringing neural capture closer to producing content directly usable in embodied AI simulators. In addition, our method can be extended to capture spatially-varying material parameters under controlled lighting conditions, an essential feature for accurately rendering captured objects.

Finally, we provide significant enhancements to the SAPIEN simulator, transforming it into a comprehensive embodied AI simulation platform. This platform now includes a many-world rendering architecture that achieves state-of-the-art performance, heterogeneous parallel simulation to support the study of generalizable manipulation with fast GPU simulation, physics-grounded simulation for depth sensors with small sim-to-real domain gaps, and various usability features such as graphical interfaces and virtual reality support.

Chapter 1

Introduction

Embodied Artificial Intelligence (Embodied AI) studies intelligent agents with a "body". The goal is to create agents capable of solving challenging tasks that require interaction with the physical world. This field represents a crucial step toward developing general intelligent agents that can match or surpass human abilities, an ultimate aspiration of artificial intelligence. While this goal seems overly far-reaching and all-encompassing at first, rapid advancements in computer vision, natural language processing, reinforcement learning, and other fields have demonstrated superhuman performance on crucial tasks foundational to embodied AI, such as object recognition, language understanding, and playing interactive games. These breakthroughs have propelled embodied AI to the forefront of AI research.

In this chapter, I first introduce the formulation of embodied AI and the motivation behind developing embodied AI simulators and assets. I then outline the key challenges in creating simulation frameworks, collecting assets, and constructing task environments. Finally, I summarize the work that will be discussed in the subsequent chapters.

1.1 Embodied AI Background

1.1.1 Problem Formulation

In the field of embodied AI, researchers commonly model *tasks* with the Markov decision process (MDP), popularized by reinforcement learning and control theories. A discrete

Markov decision process is a 4-tuple $(\mathcal{S}, \mathcal{A}, T, R)$ where $\mathcal{S}$ is the state space; $\mathcal{A}$ is the action space; $T : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ is the probabilistic transition function where $\Delta(\mathcal{X})$ denotes the set of all probability distributions over set $\mathcal{X}$. $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function. In practice, agents cannot observe the full state space $\mathcal{S}$, and thus the observation space $\mathcal{O}$ and conditional observation function $O : s \to \Delta(\mathcal{O})$ are introduced, forming the partially-observable MDP (POMDP). Additionally, discount factor γ is introduced to favor immediate rewards; time horizon H is introduced to limit the maximum episode length; and $\rho \in \Delta(\mathcal{S})$ represents the initial state distribution.

A *policy* for a POMDP is a probabilistic function $\pi : \mathcal{O} \to \Delta(\mathcal{A})$. Starting from the initial state $s_0 \sim \rho$, we obtain a trajectory following environment transition and the policy: $o_0 \sim O(s_0), a_0 \sim \pi(o), s_1 \sim T(s_0, a_0), o_1 \sim O(s_1), \ldots, s_H \sim T(s_{H-1}, a_{H-1})$, known as a *rollout* of the policy π. The goal of solving a POMDP is to find the optimal policy that maximizes the expected return $\mathbb{E}_\pi \left[\sum_{t=0}^{H-1} \gamma^t R(s_t, a_t, s_{t+1})\right]$. The reward function implicitly defines the goal of the embodied AI task.

1.1.2 Embodied AI Simulators

Simulators and virtual environments have played a central role in the development of embodied AI and relevant reinforcement learning algorithms. The starting point of deep reinforcement learning, Deep Q-Network [91], is developed based on Atari game environments. Other influential deep RL algorithms, including Proximal Policy Optimization (PPO) [125], Deep Deterministic Policy Gradient (DDPG) [74], and Soft Actor-Critic (SAC) [45], are all developed and benchmarked with Gym [10] environments based on the rigid body simulator MuJoCo [144].

An embodied AI simulator for robotics tasks typically consists of 2 core components, a physical engine and a rendering engine. The physical engine models the environment dynamics (generating next state s_{t+1} based on current state s_t and action a_t) and provides sensory observations (o_t from s_t) that can be easily computed from the simulation state, such as robot joint

positions, force/torque readings, and contact points. The rendering engine generates observations from image sensors such as RGB cameras and LiDARs, which are complex to compute and require specialized rendering software and hardware.

Simulators present several clear advantages over real-world robotic systems. First, training agents in the real world demands substantial physical resources, including robot hardware and experimental spaces, as well as expert human operators to continuously fix hardware issues and repeatedly reset the environment to an initial state. Next, simulators are not limited by the speed of real-world physics and could significantly accelerate policy training with modern computing hardware. Moreover, simulators provide privileged ground-truth information useful for model training and evaluation unobtainable in the real world, such as exact object poses and ground-truth depth images.

Simulator usage in the field of embodied AI can be classified into three 3 categories. Firstly, they serve as testbeds for developing and benchmarking learning algorithms, as demonstrated by the examples discussed above. Most Gym environments used to train RL algorithms may not closely resemble real-world robotic setups, but they address general challenges in MDPs, thereby facilitating the development of general RL algorithms. Since the trained agents are not intended for deployment in the real world, these simulations should prioritize performance over realism while maintaining the key challenges of the problem setups. This approach allows for faster training and quicker iterations on algorithm design while avoiding oversimplifying the problem. Secondly, simulators are used as a model, or *digital twin* of the real world environment in many recent works in locomotion [106, 138] and manipulation [103, 19]. In these works, a replica of the real-world environment is created in virtual space, and agents trained in simulation are then deployed in the real world. Since it is infeasible to perfectly model the real world in a simulation, the training and deployment process of agents must be adapted to minimize the impact of the domain gap—a process known as sim-to-real. Common sim-to-real approaches include domain randomization where simulation parameters are randomized to train a more robust policy, system identification where simulation parameters are identified with real data,

and fine-tuning in the real world. In this application, the realism of the simulation is crucial, as it directly influences the domain gap and the difficulty of the sim-to-real process. Finally, some simulators such as Gazebo [64] are specifically designed to replicate real-world robotic setups, allowing for safe testing of algorithm deployments without using real hardware, and for reverse-engineering black-box systems.

1.2 Challenges in Simulation Framework Construction

The construction of simulation frameworks faces a wide range of challenges, as it involves integrating and unifying knowledge from physical simulation, rendering, 3D asset acquisition, and machine learning to achieve goals such as simplifying the design of learning environments, accelerating algorithm prototyping, scaling up interactive data collection, and minimizing sim-to-real domain gaps. In this section, I detail major challenges in simulation framework construction from the following aspects: engine architecture, physical simulation, rendering, and asset acquisition.

1.2.1 Engine Architecture

Simulation framework is the key software that implements low-level physics and rendering engine and connects them to high-level embodied AI environment. The design of the simulation framework directly affects features, client APIs, and modeling procedures accessible by developers for high-level learning environments. The RLBench [58] environment relies on the CoppeliaSim [117] framework, which supports Bullet [28] and ODE [132] as physical simulators and implements OpenGL-based renderer. CoppeliaSim, and consequently RLBench, provides an excellent interactive UI for creating and exploring environments but requires extensions to be implemented in Lua and C++, which can make customization challenging for the largely Python-based learning community. ThreeDWorld [37], AI2-THOR [65] and VirtualHome [109] rely on the Unity game engine for rendering, using Unity's built-in renderer, and for physical simulation through NVIDIA PhysX [100] and various community plugins. Unity offers con-

venient scene authoring tools and a large creator community, but it is designed primarily for game creation rather than embodied AI, missing many crucial performance optimizations, such as GPU simulation and batched rendering. Habitat [121] does not rely on an external simulation framework and directly uses Bullet physics and an in-house OpenGL renderer. This design offers high performance but is more challenging to extend and maintain since the low-level system is tailored to a specific application. Recently, NVIDIA offered the GPU simulation framework IsaacSim used by IsaacLab [90] and BEHAVIOR-1K [70]. However, the system integrates with a closed ecosystem, restricts modifications to the simulation and rendering engines, and supports a limited range of hardware.

Analysis of existing systems reveals several important dimensions to consider in the design of a simulation framework for embodied AI. First, the framework needs to support easy modifications to adapt to the rapidly growing field. Second, it should provide user-friendly graphical and programming interfaces accepted by the learning community. Third, it should be designed and optimized specifically for the field of embodied AI. Finally, it is essential to select physical and rendering engines whose features align with intended use cases.

1.2.2 Physical Simulation

Physical simulation in embodied AI typically refers to collision checking, kinematics computation, and dynamics computation. In general, simulating physical phenomena of higher fidelity or complexity requires more computation, and thus designers of simulation frameworks and environments need to consider what level of physics to simulate. For example, Habitat [121] is designed for training agents to navigate static scenes, and only collision checking is implemented. ALFRED [127] introduces high-level instruction following into navigation and supports interacting with objects with discrete commands such as pick and place, where objects are teleported to and from the agent. Habitat 2 [137] and ManipulaTHOR [33] add dynamic simulation to robots and objects but do not simulate complex contact forces such as friction. IsaacGym [81] and robosuite [171] are designed to study dynamic locomotion and manipulation and support

full rigid body dynamics. SoftGym [75] and PlasticineLab [54] enable the study of deformable material and fluids by bringing particle dynamics into embodied AI simulators.

1.2.3 Rendering

Rendering is the process of synthesizing images from synthetic models, materials, and lighting. Different embodied AI applications demand varying rendering workloads. For example, environments designed to validate the feasibility of algorithms with visual inputs favor speed over photorealism. However, when deploying visual policies in the real world, the sensor simulation should be as realistic as possible. Therefore, an embodied AI simulation framework should support a wide range of rendering modes, including both fast rasterization and realistic ray tracing. While mature rendering solutions exist in the game and movie industries, these solutions are optimized for rendering a single scene. Embodied AI environments are commonly used in batches with hundreds or thousands of scenes, necessitating a redesign of the rendering system to achieve good performance. Finally, versatility is another important consideration for renderers, as 3D object representation and capturing continuously evolve, the renderer also needs to be updated to support these new techniques.

1.2.4 Asset Acquisition

Obtaining simulation assets, including 3D object models, visual materials, scene layouts, and part semantics, is one of the most significant challenges in embodied AI simulation frameworks. Even with highly advanced physical and rendering technology, the tasks supported by the simulator are ultimately limited by what asset is available. While there exists large-scale 3D object datasets such as ShapeNet [18] and Objaverse [31], the models vary greatly in geometry and visual quality. These models also lack physical material and part-motion information, which are necessary for realistic physical simulation.

An alternative approach to simulation asset acquisition is by capturing real-world objects. This direction is especially promising given the rapid advancements in neural capturing

techniques [89, 97, 63] and the increasing availability of 3D scanning software and hardware on consumer devices. However, applying 3D captures in embodied AI simulators faces several challenges. First, many capture methods utilize non-conventional 3D representations that do not directly map to existing simulation and rendering software. Methods for converting these representations into conventional ones with minimal loss of quality, or novel simulation and rendering methods that can directly accept these representations, need to be researched. Second, 3D capturing methods may only capture the color or lighting of the scene without disentangling visual material from lighting, which is necessary for rendering them under different lighting conditions.

1.3 Overview of Techniques and Contributions

To address the challenges discussed above, I introduce three works in this dissertation, covering 3D dataset construction, asset capture, and simulation framework construction to create a fully-featured simulation platform to support current and future research in embodied AI.

The first work (Chapter 2) details our effort to create the first embodied AI simulation framework to support generalizable manipulation of articulated objects. Most importantly, we address the lack-of-content problem by collecting the first large-scale simulation-ready articulated object dataset PartNet-Mobility, including part annotation, visual textures, motion axis, and motion ranges for more than 2k objects. The dataset has made many research problems feasible, including motion axis prediction and generalizable part manipulation. Accompanying the dataset, we developed the first prototype of our simulation framework, SAPIEN, where we integrated CPU-based rigid and articulated object physical simulation, implemented rasterized and ray-traced rendering with limited customization support, and provided a robotic interface through the Robot Operating System (ROS).

The second work (Chapter 3) handles 3D asset acquisition through neural capturing. We were motivated by the fact that the popular neural capture method NeRF [89] relies on 3D

volumetric representations, while most simulation and rendering techniques use 2D surface representations. In this work named NeuTex, we took the first step towards bridging 3D and 2D representations by explicitly disentangling object geometry represented as 3D volume from object appearance represented as a 2D texture map. This work is the first to recover surface-aware texture parametrization and enables 2D appearance editing. In addition, we provide an extension to capture 2D visual material parameters, such as albedo and roughness, under controlled lighting conditions. This allows the captured objects to respond to different lighting conditions.

The third work (Chapter 4) is a technical report for the latest progress on our simulation platform SAPIEN after years of continuous development and maintenance. This report covers techniques developed to support multiple published research and new systems that achieve state-of-the-art simulation and rendering performance and sim-to-real capabilities. In particular, we discuss various new features that significantly improve the performance, realism, or usability of the current SAPIEN system, including the heterogeneous parallel simulation where we support efficient simulation of significantly different environments in parallel using GPU-based physical simulator, a novel many-world rendering architecture that achieves the highest rendering throughput, deformable body simulators based on material-point method and the finite-element method to support more manipulable material types, physics-grounded active stereo and LiDAR sensor simulations that show minimal sim-to-real domain gaps, cross-platform software support, and convenient user interfaces including GUIs and virtual reality (VR). Our platform could bring boosts in performance, simplification in architecture, and improvements in sim-to-real for embodied AI tasks in general.

Chapter 2

Building a Simulator for Generalizable Articulated Object Manipulation

Building home assistant robots has long been a goal for vision and robotics researchers. To achieve this task, a simulated environment with physically realistic simulation, sufficient articulated objects, and transferability to the real robot is indispensable. Existing environments achieve these requirements for robotics simulation with different levels of simplification and focus. We take one step further in constructing an environment that supports household tasks for training robot learning algorithm. Our work, SAPIEN, is a realistic and physics-rich simulated environment that hosts a large-scale set of articulated objects. SAPIEN enables various robotic vision and interaction tasks that require detailed part-level understanding.We evaluate state-of-the-art vision algorithms for part detection and motion attribute recognition as well as demonstrate robotic interaction tasks using heuristic approaches and reinforcement learning algorithms. We hope that SAPIEN will open research directions yet to be explored, including learning cognition through interaction, part motion discovery, and construction of robotics-ready simulated game environment.

2.1 Introduction

To achieve human-level perception and interaction with the 3D world, home-assistant robots must have the capability to use perception to interact with 3D objects [29, 164, 114]. For

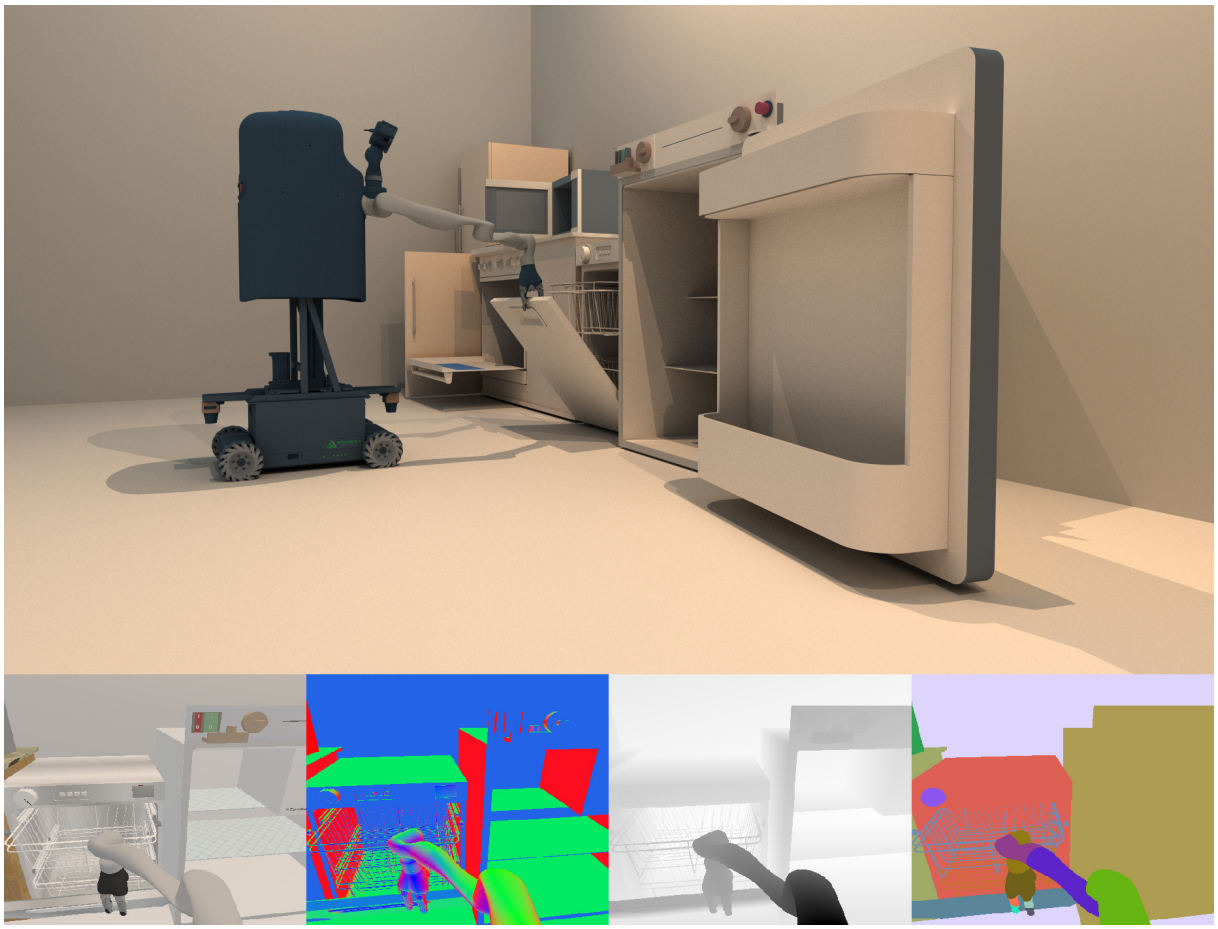


Figure 2.1. Robot-object Interaction in SAPIEN. We show the ray-traced scene (top) and robot camera views (bottom): RGB image, surface normals, depth and semantic segmentation of motion parts, while a robot is learning to operate a dishwasher.

a robot to help put away groceries, it must be able to open the refrigerator by locating the door handle, pulling the door and fetching the target objects.

One direct way to address the problem is to train robots by interacting with the real environment [83, 13, 69]. However, training robots in the real world could be very time consuming, costly, unstable, and potentially unsafe. Moreover, a slight perturbation in hardware or environment setup can result in different outcomes in the real world, thus inhibiting reproducible research. Researchers, therefore, have long used simulated environments for tasks such as navigation [120, 157, 121, 4, 11, 154, 44, 157] and control [64, 117, 144, 28].

Constructing simulated environments for robot learning with transferability to the real

Table 2.1. Comparison to other Simulation Environments. Habitat [121] is a representative for navigation environments, which include Gibson [157, 156], Minos [120] and others; they primarily use static physics but are starting to add interactions very recently. AI2-THOR [65] is a representative for game-like interactive environments; these environments usually support navigation with limited low-level object interactions. OpenAI Gym [10] and RLBench [58] provide interactive environments, but the use of commercial software limits their customizability.

Environment	Level	Physics	Rendering	Tasks	Interface
Habitat [121]*	Scene	Static+	Real Photo	Navigation, Vision	Python, C++
AI2-THOR [65]*	Scene-Object	Dynamic	Unity	Navigation+, Vision	Python, Unity
OpenAI Gym MuJoCo [10]	Scene-Object	Dynamic	OpenGL(fixed)	Learning, Robotics	Python
RLBench[58]	Scene-Object	Dynamic	V-REP, PyRep	Learning, Vision, Robotics	Python, V-REP
SAPIEN	Scene-Object-Part	Dynamic	Customizable	Learning, Vision, Robotics	Python, C++

world is a non-trivial task. It faces challenges from four main aspects: 1) The environment needs to reproduce the real-world physics to some level. As it is still infeasible to simulate real-world physics exactly, any physical simulator needs to decide the level-of-details and accuracy it operates on. Some approximate physics by simulating rigid bodies and joints[100, 144, 28]; some handle soft deformable objects [144, 28]; and others simulate fluid [144, 122]. 2) The environment should incorporate the simulation of real robots, including the behaviors of real robotics manipulators, sensors and controllers [95]. Only this can enable seamless transfer to the real-world after training. 3) The environment needs to produce physically accurate renderings to mitigate the visual domain gap. 4) Most importantly, the environment requires sufficient content, scenes and objects for the robot to interact with, since data diversity is critical for training and evaluating learning-based algorithms. The content also determines how much we shall address challenges in the previous tasks: data with soft objects such as cloth requires deformable body simulation; translucent objects require special rendering techniques, and specific robot requires a specific interface.

Existing environments achieve these requirements for robotics simulation with different levels of simplification and focus. For example, OpenAI Gym [10] provides an interactive and easy-to-use interface; Gibson [157] and AI Habitat [121] use photorealistic rendering for semantic navigation tasks. A more detailed discussion of popular environments can be found in

Sec 2.2. These environments can support the benchmarking and training of down-stream tasks such as navigation, low-level control, and grasping. However, from the perspective of tasks, there still lacks environments that target at object manipulation of daily objects, a basic skill of household robots. In a household environment, a great portion of daily objects are articulated and require manipulation: bottles with caps, ovens with doors, electronics with switches and buttons. Notably, RLBench [58] provides well-defined robotics tasks and realistic controller interface with detailed manipulation demonstration, but it lacks diversity in its simulated scenarios.

We take one step further in constructing an environment that supports the manipulation of diverse articulated objects. Our system, SAPIEN, is a realistic and physics-rich simulated environment that hosts a large set of articulated objects. At the core of SAPIEN are three main components: 1) SAPIEN Engine, an interaction-rich and physics-realistic simulation environment integrating PhysX physical engine and ROS control interface; this engine supports accurate simulation of rigid body and joint constraints for simulation of articulated objects. 2) SAPIEN Asset, including PartNet-Mobility dataset, which contains 14K movable parts over 2,346 3D articulated models from 46 common indoor object categories, richly annotated with kinematic part motions and dynamic interactive attributes; 3) SAPIEN Renderer, with both fast-frame-rate OpenGL rasterizer and more photorealistic ray-tracing options. We demonstrate that SAPIEN enables a large variety of robotic perception and interaction tasks by benchmarking state-of-the-art vision algorithms for part detection and motion attribute recognition. We also show a variety of robotic interaction tasks that SAPIEN supports by demonstrating heuristic approaches and reinforcement learning algorithms.

2.2 Related Work

Simulation Environments.

In recent years, there has been a proliferation of indoor simulation environments primarily designed for navigation, visual recognition and reasoning [120, 154, 157, 121, 4]. Static environments, based on synthetic scenes [154] or real-world RGB-D scans [4] and reconstruc-

tions [120, 157, 121], are able to provide images that closely resembles reality, minimizing the domain gap in the visual aspect. However, they usually offer very limited or no object interactions, failing to capture the dynamic and interactive nature of the real world.

To allow for more interactive features in environments, researchers leverage partial functionalities of game engines or physics engine to provide photorealistic rendering together with interactions [115, 94, 65, 109, 11, 161, 38, 67]. When agents interact with objects in these environments, it is via high level state changes triggered by explicit commands (e.g. "open refrigerator"), or proximity (e.g. refrigerator door opens when the robot or robot arm is next to the trigger region). In addition, the underlying physics is often over-simplified, for instance, to directly exert force and torque. While they allow for research on high-level object interactions, they cannot close the gap between high-level instructions and low-level dynamics since they fail to include accurate simulation of articulated robots and objects. This limits the use of such simulators for learning of detailed low-level robot-object interactions.

Finally, there are environments that integrate full-featured physics engines. These environments are favored in continuous control and reinforcement learning tasks. OpenAI Gym [10], RLLAB [32], DeepMind Control Suite [140] and DoorGym [146] integrate MuJoCo physical engine to provide RL environments. Arena [135], a platform that supports multi-agent environments, is built on top of Unity [60]. PyBullet [28], a real-time physics engine with Python interface, powers a series of projects focusing on robotics tasks [167, 80]. Gazebo [64], a high-level visualization and modeling package, is widely used in robotics community [88, 55]. Recently, RLBench [58], a benchmark and physical environment for robot learning, uses V-REP [117] as the backend to provide diverse tasks for robot manipulation. Our environment, SAPIEN engine, is directly based on the open-source Nvidia PhysX API [100], which has comparable performance and interface with PyBullet, and avoids the unnecessary complication introduced by game engine infrastructures, or any barriers from commercial software such as MuJoCo and V-REP. Table 2.1 provides a brief summary of several representative environments.

One bottleneck of these robotic simulators is their limited rendering capability, which

Table 2.2. Comparison of Articulated Part Datasets. *RBO is collected in real-world with long video sequences.

Dataset	#Categories	#Models	#Motion Parts
Shape2Motion[151]	45	2,440	6,762
RPM-Net[163]	43	949	1,420
Hu et. al.@ [50]	-	368	368
RBO*[84]	14	14	21
Ours	46	2,346	**14,068**

creates a gap between simulation and the real world. Another limitation of many of these environments, including RLBench [58] and DoorGym [146], is that they are very task-centric, designed to work for only a few predefined tasks. The SAPIEN simulator, equipped with 2,346 3D interactive models from 46 object categories and flexible rendering pipelines, provides robot agents a virtual environment for learning a large set of complex, diverse and customizable robotic interaction tasks.

Simulation Content.

Navigation environments typically use datasets providing real-world RGB-D scans [157, 17, 136], and/or high-quality synthetic scenes [134]. Simulation environments that leverage game engines [115, 94, 11, 38, 65] come with manually designed or procedurally generated game scenes. For environments with detailed physics and reinforcement learning support [10, 140, 32], they usually support very few scenarios with simple objects and robot agents. Notably, RLBench [58] provides a relatively large robot learning dataset with varied tasks. To address the lack-of-content problem, our work provides a large-scale simulation-ready dataset, PartNet-Mobility dataset, that is constructed from 3D model datasets including PartNet [92] and ShapeNet [18].

There are also shape part datasets with part articulation annotations. Table 2.2 summarizes recent part mobility datasets. The RBO dataset [84] is a collection of 358 RGB-D video sequences of humans manipulating 14 objects which are reconstructed as articulated 3D meshes. The meshes have detailed part motion parameters and have realistic textures. Other datasets

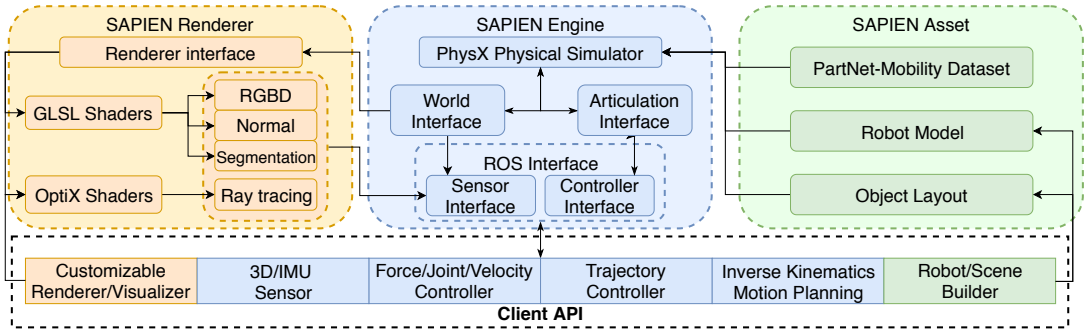


Figure 2.2. SAPIEN Simulator Overview. The left box shows SAPIEN Renderer, which takes custom shaders and scene information to produce images such as RGB-D and segmentation. The middle box shows SAPIEN Engine, which integrates PhysX simulator and ROS control interface that enables various robot actions and utilities. The right box shows SAPIEN Asset, which contains the large-scale PartNet-Mobility dataset that provides simulatable models with part-level mobility.

annotate 3D synthetic CAD models with articulation information. Hu et. al.@ [50] introduced a dataset of 368 mobility part articulations with diverse types. RPM-Net [163] provides another dataset with 969 objects and 1,420 mobility units. Shape2Motion [151] provides a dataset of 2,440 objects and 6,762 movable parts for mobility analysis, but it does not provide RGB textures and motion limits that hinders physical simulation. Compared to these datasets, our dataset contains comparable number of objects (2,346), but with much more movable part annotations (14,068). Besides, our models have textures and motion range limits, which are crucial for the dataset to be simulatable.

2.3 SAPIEN Simulation Environment

SAPIEN aims to integrate state-of-the-art physical simulators, modern graphics rendering engines, and user-friendly robotic interfaces into a unified framework (Figure 2.2), to support a diverse set of robotic perception and interaction tasks. We develop the environment with C++ for efficiency and provide Python wrapper API for ease-of-use at the user end. Below we detailedly introduce the three main components: SAPIEN engine, SAPIEN asset and SAPIEN renderer.

2.3.1 SAPIEN Engine

We use the open-source Nvidia PhysX physical engine to provide detailed robot-object interaction simulation. The system provides Robot Operating System (ROS) supports that are easy-to-use for end-stream robotic research. We provide both synchronous and asynchronous modes of simulation to support reinforcement learning training and robotics tasks.

Physical Simulation.

We choose PhysX 4.1 [100] to provide rigid body kinematics and dynamics simulation, since it is open-source, simplistic, and provides functionalities designed for robotics. To simulate articulated bodies, we provide 3 different body-joint systems: kinematic joint system, dynamic joint system, and PhysX articulation. The kinematic joint system provides kinematic objects with parent-child relations, suitable for simulating very heavy objects that are not affected by small forces. Dynamic joint systems use PhysX joints to drive rigid bodies towards constraints, suitable for simulating complicated objects that do not require accurate control. PhysX articulation is a system specifically designed for robot simulation. It natively supports accurate force control, P-D control and inverse dynamics with the cost of relatively low speed.

ROS Interfaces.

Robot Operating System (ROS) [116] is a generic and widely-used framework for building robot applications. Our ROS interface bridges the gap between ROS and physical simulator, as well as providing a set of high-level APIs for interacting with robots using physics. It supports three levels of abstractions: direct force control, ROS controllers and motion planning interface.

At the lowest level, forces and torques are directly applied on joints, similar to OpenAI Gym [10]. This control method is simple and intuitive, but challenging to transfer to real environments due to the differences between real-world dynamics and simulation, and between the continuous nature of real-robots and the discretized approximation in simulations.

For high-level control, we provide joint space and Cartesian coordinate space control

APIs. We build controllers (Figure 2.2) based upon [26] and implement the standard ROS control interface. Researchers often want to specify movement at a higher level of abstraction, for example, to move the robot arm to a desired 6-DoF pose with specific path constraints. Thus, at the highest level, we provide motion planning support based on the popular MoveIt framework [27], which can generate motion plans that effectively move the robot around without collision.

Synchronous and Asynchronous Modes.

The SAPIEN Engine (see Figure 2.2 middle) can support both synchronous and asynchronous simulation modes. In synchronous mode, the simulation step is controlled by the client, which is common in training reinforcement learning algorithm [10]. For example, the agent receives observations from simulated environments and uses a customized policy model, often a neural network, to generate the corresponding action. Then the simulation runs forward for a step. In this synchronous mode, the simulation and client algorithms are integrated together.

However, for real-world robotics, the simulation and client response need to be asynchronous [64] and separated. The simulation should run independently, like the real world, while the client uses the same API as a real robot to interact with the simulation backend. To build such a framework, we create multiple sensors and controllers following the ROS API. After simulation starts, the client receives information from sensors and uses the controller interface (see Figure 2.2) to command robots via TCP/IP communication. The timestamp is synchronized from simulation to the client side, acting as a proxy for the real-world clock time. Under the framework, the simulated robots can use the same code as their real counterparts because most real robot controllers and sensors have exactly the same interface as our simulator API. This provides one important advantage: it enables robot researchers to migrate between simulated robots and real robots without any extra setup.

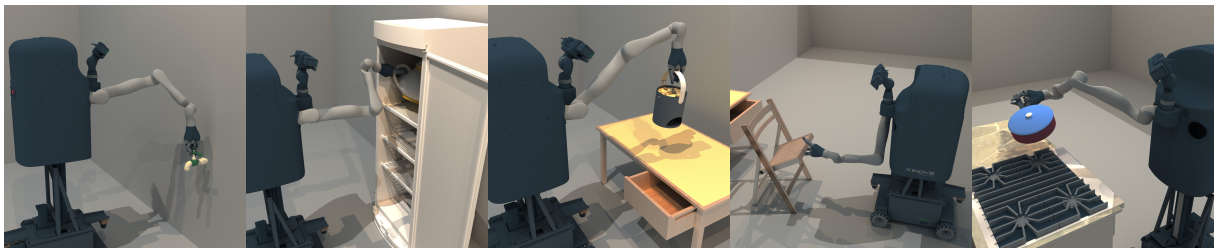


Figure 2.3. SAPIEN Enables Many Robotic Interaction Tasks. From left to right, we show five examples: faucet manipulation, object fetching, object lifting, chair folding, and object placing.

Table 2.3. Statistics of PartNet-Mobility Dataset. #M and #P shows the number of models and movable parts respectively.

	All	Bottle	Box	Bucket	Cabinet	Camera	Cart	Chair	Clock	Coffee	DishWsh.	Dispenser	Door	Eyegls	Fan	Faucet
#M	**2,346**	57	28	36	345	37	61	80	31	55	48	57	36	65	81	84
#P	**14,068**	114	94	74	1,174	341	232	1,235	106	374	112	162	103	195	172	228

	Chair	Fridge	Globe	Kettle	Keybrd	Knife	Lamp	Laptop	Lighter	MicWav	Monitor	Mouse	Oven	Pen	Phone	Pliers
#M	26	44	61	29	37	44	45	56	28	16	37	14	30	48	17	25
#P	58	118	130	66	3,593	149	165	112	86	85	93	61	214	97	271	59

	Pot	Printer	Remote	Safe	Scissors	Stapler	Stcase	Switch	Table	Toaster	Toilet	TrashCan	USB	Washer	Window
#M	25	29	49	30	47	23	24	70	101	25	69	70	51	17	58
#P	53	376	1,490	202	94	69	101	195	420	116	229	208	103	144	195

2.3.2 SAPIEN Asset

SAPIEN Asset is our simulation content, shown in the right box in Figure 2.2. It contains the large-scale ready-to-simulate PartNet-Mobility dataset, the simulated robot models and scene layouts.

PartNet-Mobility Dataset.

We propose a large-scale 3D interactive model dataset that contains over 14K articulated parts over **2,346** object models from **46** common indoor object categories. All models are collected from 3D Warehouse* and organized as in ShapeNet [18] and PartNet [92]. We annotate 3 types of motions: hinge, slider, and screw, where hinge indicates rotation around an axis (e.g. doors); slider indicates translation along an axis (e.g. drawers), and screw indicates a combined hinge and slider (e.g. bottle caps, swivel chairs). For hinge and slider joints, we annotate the motion limit (i.e. angles, lengths). For screw, we annotate the motion limits and whether the

*https://3dwarehouse.sketchup.com/

2 degrees of freedom are coupled. Each joint has a parent and a child, and the collection of connected bodies and joints is called an articulation. We require the joints of an articulation to follow a tree structure with a single root, since most physical simulator handles tree-structured joint system well. Next, for each movable part, we assign a category-specific semantic label. Table 2.3 summarizes the dataset statistics.

SAPIEN Asset Loader

Unified Robot Description Format (URDF) is a common format for representing a physical model. For each object in the SAPIEN Asset, including PartNet-Mobility models and robot models, we provide an associated URDF file, which can be loaded in simulation. For accurate simulation of contact, we decompose meshes into convex parts [82, 52]. We randomize or manually set the physical properties, e.g. friction, damping, density, to appropriate ranges. For robot models, we also provide C++/Python APIs to create a robot piece by piece to avoid complications introduced by URDF.

2.3.3 SAPIEN Renderer

SAPIEN Renderer, shown in the left box of Figure 2.2, renders simulated scenes with OpenGL 4.5 and GLSL shaders, which are exposed to the client application for maximal customizability. By default, the rendering module uses a deferred lighting pipeline to provide RGB, albedo, normal, depth, and segmentation from camera space, where lighting is computed with Oren–Nayar diffuse model [153] and GGX specular model [148]. Our customizable rendering interface can suit special rendering needs, and even allow completely different rendering pipelines. We demonstrate this by replacing the fast OpenGL framework with our ray tracer coded with Nvidia OptiX [104] to produce physically accurate images at the cost of rendering time (see Figure 3.1).

Table 2.4. Movable Part Detection Results. (AP% with IoU threshold 0.5) 2D and PC denote 2D images and point clouds as different input modalities for the two algorithms. We show the detailed results for four objects categories and summarize the mAP over all categories.

		Cabinet				Table					Faucet			Fan		**All**
Algorithm	Inputs	rot. door	body	drawer	trans. door	drawer	body	wheel	door	caster	switch	base	spout	rotor	frame	**mAP**
Mask-RCNN [46]	2D (RGB)	62.0	94.2	66.4	27.7	54.3	88.0	3.4	6.3	0.0	52.5	47.9	99.7	54.4	67.5	**53.0**
	2D (RGB-D)	61.7	93.0	63.0	26.3	58.6	89.9	1.4	13.2	0.0	52.1	55.8	98.9	39.4	67.4	**52.8**
PartNet InsSeg [92]	PC (XYZ)	20.6	65.9	35.1	9.8	15.7	71.3	1.7	1.0	0.0	34.4	55.9	64.2	50.9	74.8	**36.1**
	PC (XYZRGB)	17.4	64.3	23.6	5.0	16.4	81.8	1.3	2.0	1.0	29.9	64.1	78.0	42.0	63.5	**37.1**

2.3.4 Profiling Analysis

The SAPIEN engine can run at about 5000Hz on the manipulation task we will describe in Sec. 2.4.2 and can render at about 700Hz with OpenGL mode. Tests were performed on a laptop with Ubuntu 18.04, on 2.2 GHz Intel i7-8750 CPU and an Nvidia GeForce RTX 2070 GPU.

2.4 Tasks and Benchmarks

We demonstrate the versatile abilities of our simulator by demonstrating robotic perception and interaction tasks.

2.4.1 Robotic Perception

SAPIEN simulator, equipped with the PartNet-Mobility dataset, provides a platform for several robotic perception tasks. In this paper, we study the tasks of movable part detection and part motion estimation, which are two important vision tasks supporting downstream robotic interaction.

Movable Part Detection

Before interacting with objects by parts, robotic agents need to first detect the parts of interest. Therefore, we define the task of movable part detection as follows. Given a single 2D image snapshot or 3D RGB-D partial scan of an object as input, an algorithm should produce

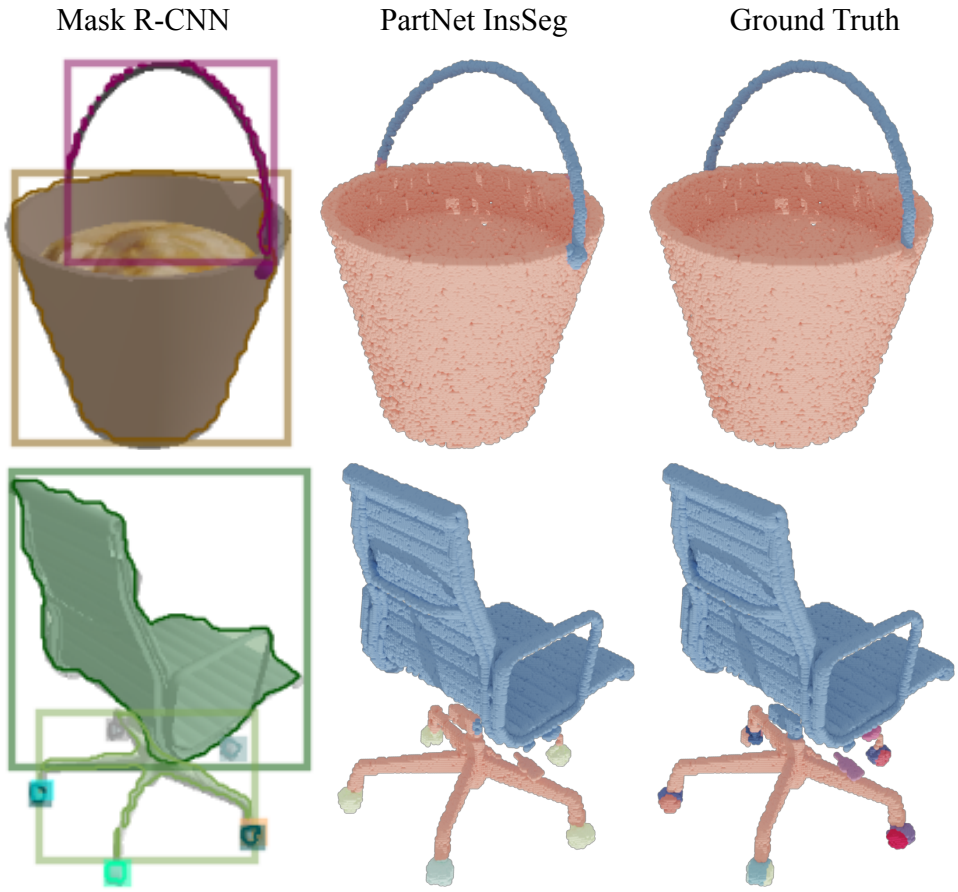


Figure 2.4. Movable Part Detection Results. The left column shows the results of Mask R-CNN [46], where each bounding box indicates a detected movable part. The middle and the right columns show the results of PartNet InsSeg [92] and the ground truth point clouds respectively, where different parts are in different color.

several disjoint part masks associated with their semantic labels, each of which corresponds to an individual movable part of the object.

Leveraging the rich assets from the PartNet-Mobility dataset and the SAPIEN rendering pipeline, we evaluate two state-of-the-art perception algorithms for object or part detection. Mask R-CNN [46] takes a 2D image as input and uses a region proposal network to detect a set of 2D part mask candidates. PartNet-InsSeg [92] is a 3D shape part instance segmentation approach that uses PointNet++ [112] to extract geometric features and proposes panoptic segmentation over shape point clouds.

We render each object in the PartNet-Mobility dataset into RGB and RGB-D images

from 20 randomly sampled views, with resolution 512×512. The camera positions are randomly sampled over the upper hemisphere to ensure space coverage. Simple ambient and directional lighting without shadows are provided for RGB rendering. With known camera intrinsics, we lift the 2.5D RGB-D images into 3D partial scans for PartNet-InsSeg experiments. We use all 2,346 objects over 46 categories from the PartNet-Mobility dataset for this task. We use 75% of data (1,772 shapes) for training and 25% (574 shapes) for testing. For quantitative evaluation, we report per-part-category Average Precision (AP) scores as commonly used for object detection tasks and average across all part categories to compute mAP for each algorithm.

Table 2.4 shows the quantitative results of Mask R-CNN on RGB and RGB-D settings and PartNet-InsSeg on the XYZ (depth-only) and XYZRGB (RGB-D images) settings. We observe that both methods perform poorly on detecting small parts (e.g., table wheel and table caster), and the phenomenon is less severe for object categories that have relatively balanced sizes (e.g., fan and faucet). Small movable parts (e.g., button, switch, and handle) often play critical roles in robot-object interaction, and will demand more well-designed algorithms in the future. Figure 2.4 visualizes the Mask-RCNN and PartNet-InsSeg part detection results on two example RGB-D partial scans.

Table 2.5. Motion recognition results. *H* acc. and *S* acc. denotes classification accuracy for hinge and slider respectively. H_o err. denotes average distance from predicted hinge origin to ground truth axis. H_a/S_a denotes average hinge/slider angle difference from predicted axis to ground truth. door err. is average angle difference from predicted door pose to ground truth. drawer err. is average length difference from predicted drawer pose to ground truth.

Setting	Algorithm	*H* acc.	*S* acc.	H_o err (*m*)	H_a err (°)	S_a err. (°)	door err. (°)	drawer err. (*m*)
RGB-D	ResNet50	95.5%	95.5%	0.168	18.9	6.35	14.4	0.0645
RGB-pc	PointNet++	95.4%	95.5%	0.195	18.5	7.75	20.8	0.0918

Motion Attributes Estimation

Estimating motion attributes for articulated parts gives strong priors for robots before interacting with objects. In this section, we perform the motion attributes estimation task that jointly predicts the motion type, motion axis, and part state for articulated parts.

We consider two types of rigid part motions: 3D rotation and translation. Some parts, such as bottle cap, may have both rotation and translation motions. For translation motions, we use a 3-dim vector to represent the direction. For rotation motions, we parameterize the outputs as two 3-dim vectors to specify rotation axis direction and a pivot point on the axis. We define relative positions of the articulated part with respect to its semantic rest positions as part states. For example, the rest position for drawers and doors is when they are closed. However, defining part rest states has intrinsic ambiguities. For example, round knobs with rotation symmetry do not present a detectable rest position. Thus, we use a subset of 640 models over 10 categories, which consists of 779 doors and 529 drawers for this task, following the same train and test splits used in the previous section.

We evaluate two baseline algorithms, ResNet-50 [47] and PointNet++ [112], that deals with the input RGB-D partial scans using either 2D or 3D formats. For ResNet-50, we input RGB-D images augmented with target part mask (5-channel in total). For PointNet++, we substitute the 5-channel image with its camera-space RGB point cloud. We train both networks to output a 14-dim motion vector $m = \left(T_r, T_t, p_1^r, p_2^r, p_3^r, d_1^r, d_2^r, d_3^r, d_1^t, d_2^t, d_3^t, x_{\text{door}}, x_{\text{drawer}}\right)$, where T_r and T_t respectively output the probability of this joint being rotational and translational, (p_1^r, p_2^r, p_3^r) and (d_1^r, d_2^r, d_3^r) indicate pivot point and rotation axis for hinge joints, (d_1^t, d_2^t, d_3^t) represents the direction of a proposed slider axis, and finally, x_{door} and x_{drawer} regress the part poses for doors and drawers respectively. The part pose is a number normalized within $[0, 1]$ indicating the current joint position.

We summarize the experimental results in Table 2.5. The classification of different motion types achieves quite high accuracy, and the axis prediction for sliders (translational joints) achieves lower error than for hinges (rotational joints). In our experiments, ResNet50 achieves better performance than PointNet++. This could be explained by the much higher number of network parameters in ResNet. However, intuition suggests that such 3D information should be more easily predicted directly on 3D data. Future research should focus more on how to improve 3D axis prediction with 3D grounding.



Figure 2.5. Robotic Interaction tasks. We study two robotic interaction tasks: door-opening and drawer-pulling.

2.4.2 Robotic Interaction

With the large-scale PartNet-Mobility dataset, SAPIEN also supports various robotic interaction tasks, including solving low-level control tasks, such as button pushing, handle grasping, and drawer pulling, and planning tasks that require long-horizon logical planning and low-level controls, e.g., removing the mug from a microwave oven and then putting it on a table. Having both diverse object categories and rich intra-class instance variations allows us to perform such tasks on multiple object instances at category levels. Figure 2.3 shows a rich variety of robotic interaction tasks that SAPIEN enables.

In SAPIEN, we enable two modes for robotic interaction tasks: 1) using perception ground-truth (e.g., part mask, part motion information, and 3D locations) to accomplish the task. In this way, we factor out the perception module and allow algorithms to focus on robotic control and interaction tasks; 2) using the raw image/point-cloud as inputs, the method needs to develop its own perception, planning and control modules, which is our end-goal for the home-assistant

robots to achieve. Also, this mode enables end-to-end learning for perception and interactions (e.g., learning perception with a specific interaction target).

Door-opening and Drawer-pulling.

We perform two manipulation tasks: door-opening and drawer-pulling, as shown in Figure 2.5. We use a flying gripper (Kinova Gripper 3 [15]) that can move freely in the workspace. All dynamics properties except gravity, (e.g. contact, friction, and damping) are simulated in our environment. We perform our drawer-pulling tasks on 108 cabinet instances and door-opening tasks on 77 cabinet instances.

In our tasks, if the gripper can move a given joint (e.g. slider joint for drawer, hinge joint for door) through 90% of its motion range, then it will be regarded as a success. If the agent cannot move the joint to the given threshold or move in the opposite direction, then it fails. The input of the agent consists of point clouds, normal maps and segmentation masks captured by three fixed cameras mounted on the left, right and front of the arena respectively. The agent can also access all information about its self (e.g. 6 DoF pose).

Heuristic Based Manipulation.

To demonstrate our simulator in manipulation tasks, we first use manually designed heuristic pipelines to solve the tasks. For drawer-pulling, we use point cloud with ground-truth segmentation to detect a valid grasp pose for drawer handle. Then we use velocity controller to pull it to the joint limit. Using ground-truth visual information, we can achieve a 95.3% success rate. As for the door-opening task, we first open the door with a small angle using a similar approach (grasp a handle at first). Then we use Position Based Visual Servoing (PBVS) [56] to track and clamp the edge of the door. Finally, the door is opened by rotating the edge. This method (PBVS) achieves an 81.8% success rate for door opening.

Learning Based Manipulation.

We also demonstrate the above two tasks using reinforcement learning. We test the generalizability of the RL agent by training on limited objects and testing on unseen objects with

different size, density, and motion properties. We adopt Soft Actor-Critic(SAC) [45], which is one of the SOTA reinforcement learning algorithms, trained on $2, 4, 8, 16$ doors or drawers, and test on the rest unseen models.

We provide three different state representations: 1) raw state of the whole scene (**raw-exp**), consisting of current positions and velocities of all the parts; 2) mobility-based representation (**mobility-exp**), with 6D pose of motion axis and average normal, and current joint angles and velocities of the target part; 3) visual inputs (**visual-exp**), where we set a front-view camera capturing RGB-D images for the object every time step, augmented with segmentation mask for the target part.

We use the same flying gripper and initialize it on the handle. The grasp pose is generated by the heuristic method as described in the above section. During training, agents receive positive rewards when the target part approaches the joint limit with the opening door/drawer, while obtaining negative rewards when the gripper falls off the handle. We interact with multiple objects simultaneously during training, and use a shared replay buffer to collect experiences to train SAC. After 1M interaction steps, we evaluate the performance on the unseen objects, each for 20 episodes.

For doors, the evaluation metric is the average achieved degree. For drawers, we report the success rate of opening 80% of joint limits. Table 2.6 shows our experimental results. For door-opening, the RL agent tends to overfit the training objects, as when the number of training objects grows, the performance drops. However, training on more scenarios will improve the generalization capability with increased test performance. For drawer-pulling, although the performance follows the same pattern as the door, it is relatively stable across the number of training objects. This is because drawers are relatively easier to pull out, as the movement for the gripper almost follows the same pattern every time step.

Among all the representations, **mobility-exp** gives the best performance. For doors, **visual-exp** representation also performs close to **mobility-exp**; however for drawers, **raw-exp** is better than **visual-exp**. This is because the camera is fixed during the interaction. For drawer-

Table 2.6. SAC results on door and drawer opening.

Tasks		Door (Final Angle Degree)				Drawer (Success Rate)			
		2	4	8	16	2	4	8	16
raw-exp	train	85.4	70.5	50.5	38.4	**0.84**	**0.82**	0.77	0.75
	test	14.7	18.7	21.2	27.3	0.61	0.63	0.66	0.66
mobility-exp	train	88.7	**78.6**	**59.2**	**41.1**	0.83	0.81	**0.79**	**0.78**
	test	**22.9**	**27.3**	27.5	**32.8**	**0.65**	**0.65**	**0.69**	**0.68**
visual-exp	train	**90.2**	65.2	56.7	32.1	0.80	0.72	0.69	0.63
	test	21.7	24.5	**28.1**	29.6	0.59	0.60	0.61	0.60

opening, the visual features remain almost the same every time step from the front view, so it provides little information about state changing. These observations lead us to some interesting future work. First, we need proper vision methods to encode the geometric information of the scene, which may change during interaction procedures. Second, although these tasks are not hard for heuristic algorithms, RL-based approaches fail to perform well on all the objects. Future works may study how to enhance the transferability and efficiency of RL on the tasks.

2.5 PartNet-Mobility Annotation System

We developed a web interface (Figure 2.7) for mobility annotation. This tool is a question answering (QA) system, which proposes questions based on current stage of annotation. It exploits the hierarchical structures of PartNet to propose objects without relative mobility, and generates new questions based on past annotations. Using this tool, annotators will not miss any movable parts if they answer every question correctly, and they will not face any redundant questions by design. The output mobility annotations are guaranteed to satisfy tree properties suitable for simulation.

The annotation procedure has the following steps:

- We start with a PartNet semantic tree, and traverse the tree nodes. Annotators are prompted with questions asking if current subtree has relative motion. If it does not, all parts in this

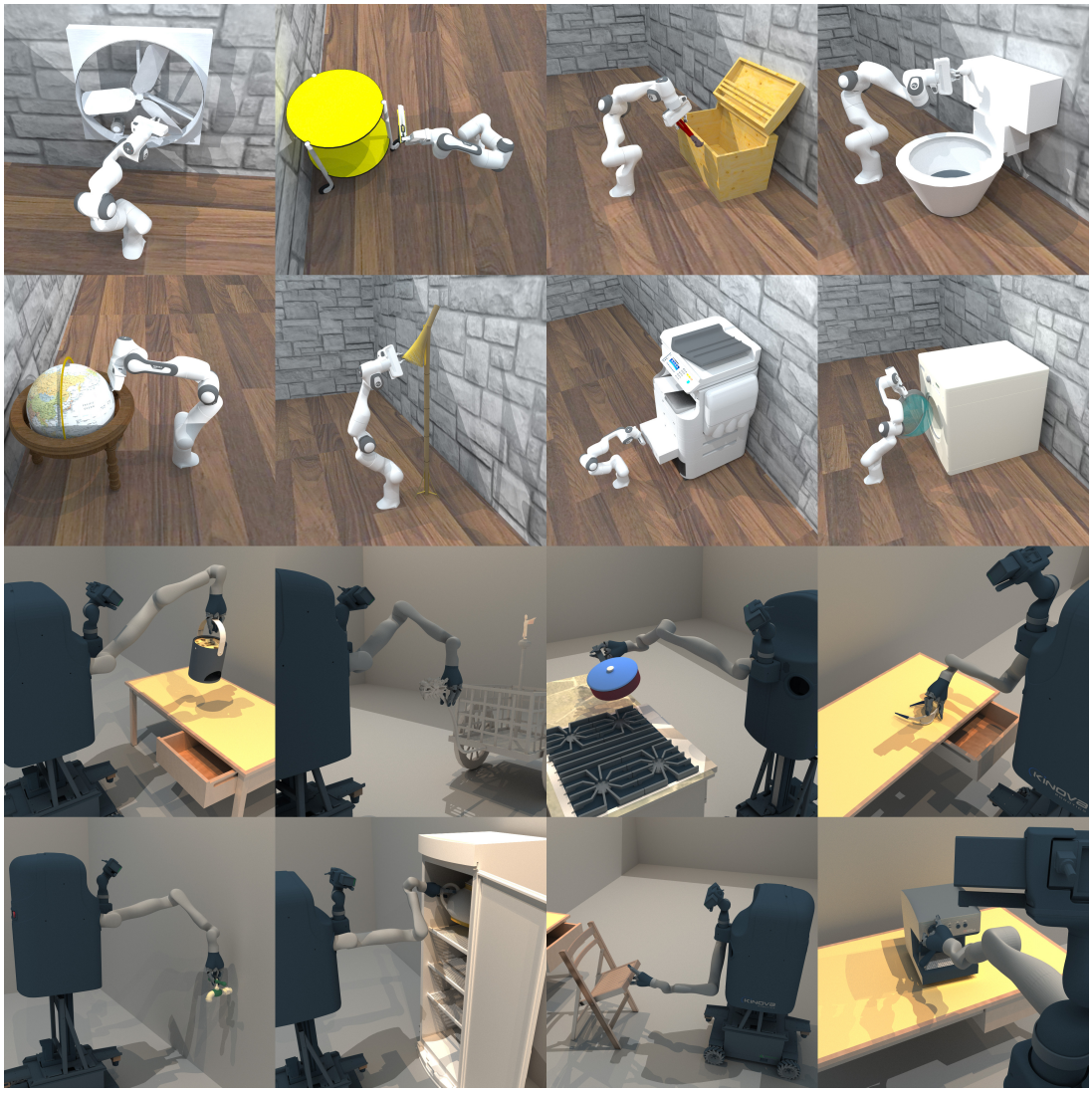


Figure 2.6. Diverse manipulation tasks supported by SAPIEN.

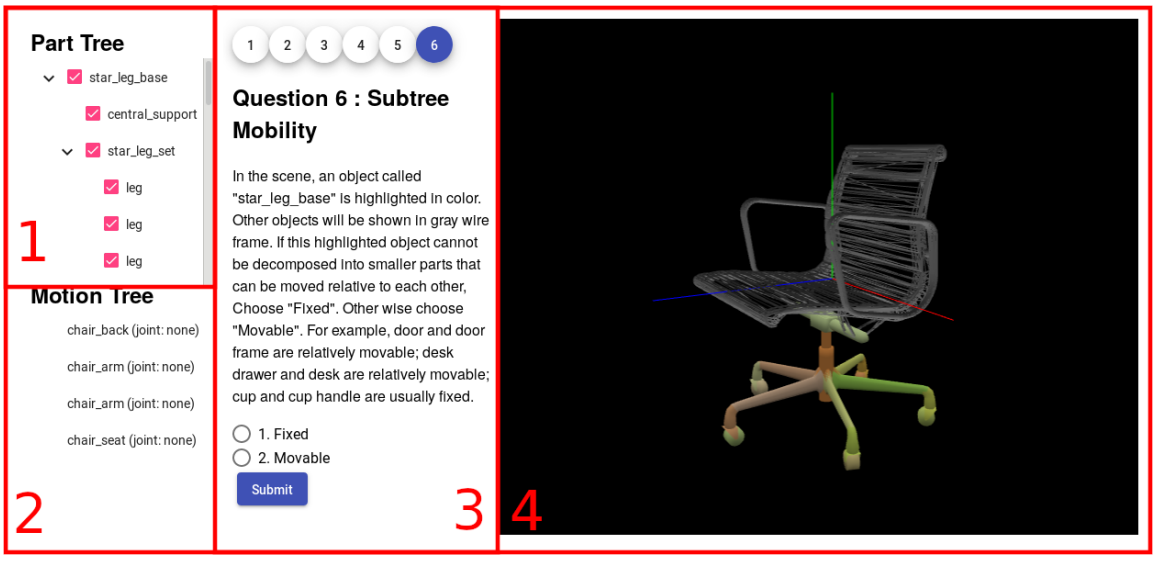


Figure 2.7. Annotation interface. 1) Part Tree: PartNet semantic tree that proposes fixed parts. 2) Motion tree: annotated movable parts. 3) Question: auto-generated exhaustive questions. 4) Visualization for current question and for motion axis annotation.

tree will be fixed together; otherwise, the same question is asked again on the child nodes of this subtree.

- When the PartNet semantic tree traversal is finished, annotators are asked to choose parts that are fixed together.

- Next, annotators are asked to choose parts that are connected with a hinge (rotational) joint. They will then choose parent-child relation, and annotate axis position/motion limit with our 3D annotation tool.

- Next, annotators are asked to choose parts that are connected with a slider (translational) joint. They will similarly choose motion parameters and decide if this axis also bears rotation (screw joint).

- Finally, annotators will annotate each separate object in the scene as “fixed base”, “free””, or “out lier”.

The procedure is summarized in the following pseudo-code block.

Algorithm 1. Annotating PartNet-Mobility dataset

```
Propose fixed parts based on PartNet tree
for There are parts can be fixed together do
    Select a group of relatively fixed parts
end for
for Rotation relationship exists do
    Select parent and child
    Pick rotation axis
    Input motion range
end for
for Translation relationship exists do
    Select parent and child
    Pick translation axis
    Input motion range / whether it can also rotate
end for
Choose whether root nodes are fixed/free
```

2.6 Movable Part Segmentation and Motion Recognition

Movable Part Segmentation: complete results

Table 2.7 shows the movable part segmentation results for all categories in PartNet-Mobility dataset.

Motion Recognition: experiment details

For this task, we normalize the $[0, 2\pi]$ hinge joint range to $[0, 1]$. For sliders, we normalize by the maximum motion range over the dataset to make the motion range prediction within $[0, 1]$.

Algorithm.

The baseline algorithm we use is a a ResNet[47] classification and Regression network. The input is the ground truth RGB-D image and the segmentation mask for the target movable part. The output has 7 terms: $\hat{T}_r \in \{0, 1\}$, whether this part has a rotational joint.

$\hat{T}_t \in \{0, 1\}$, whether this part has a translational joint.

$\hat{\mathbf{p}}^r \in \mathbb{R}^3$, pivot of a predicted rotational axis.

$\hat{\mathbf{d}}^r \in [-1, 1]^3$, direction of a predicted rotational axis.

Table 2.7. Movable part segmentation results for all categories

		Bottle			Box		Bucket		Cabinet				Camera				Cart		Chair		
Algorithm	Setting	tr. lid	body	rot. lid	rot. lid	body	handle	body	door	body	door	drawer	lens	button	body	knob	wheel	body	wheel	seat	leg
Mask-	RGB	0.0%	57.4%	69.3%	49.3%	65.7%	2.7%	91.7%	62.0%	94.2%	27.7%	66.4%	26.7%	20.9%	79.0%	4.8%	54.6%	95.6%	25.1%	97.0%	88.3%
RCNN	RGB-D	13.9%	68.3%	67.8%	51.5%	66.5%	1.6%	100.0%	61.7%	93.0%	26.3%	63.0%	26.4%	17.0%	92.6%	8.1%	55.3%	93.9%	23.1%	99.0%	85.2%
PartNet	XYZ	24.5%	47.7%	53.5%	27.6%	46.2%	63.4%	99.7%	20.6%	65.9%	9.8%	35.1%	17.0%	0.0%	51.4%	0.0%	6.2%	71.7%	1.2%	93.0%	86.4%
InsSeg	XYZRGB	5.9%	41.3%	54.8%	24.2%	36.8%	60.7%	98.9%	17.4%	64.3%	5.0%	23.6%	10.5%	0.0%	46.1%	1.0%	9.4%	77.3%	1.9%	95.7%	89.2%

		Chair			Clock		CoffeeMachine						Dishwasher		Dispenser		Display			Door	
Algorithm	Setting	knob	caster	lever	hand	body	button	lid	body	lever	knob	container	rot. Door	body	lid	body	rot. screen	base	button	frame	rot. door
Mask-	RGB	0.0%	2.5%	20.0%	11.4%	61.4%	14.7%	73.4%	65.7%	0.0%	43.0%	100.0%	70.4%	90.0%	74.9%	90.1%	74.4%	34.7%	0.0%	39.4%	40.7%
RCNN	RGB-D	0.0%	3.6%	13.4%	12.5%	68.3%	10.4%	61.4%	67.4%	1.0%	35.6%	98.0%	66.8%	87.8%	73.2%	88.1%	71.3%	33.4%	0.0%	35.7%	54.6%
PartNet	XYZ	0.0%	1.0%	0.0%	0.0%	77.0%	0.0%	43.6%	62.4%	0.0%	0.0%	94.0%	50.5%	67.0%	49.1%	57.6%	66.1%	37.1%	0.0%	49.2%	35.3%
InsSeg	XYZRGB	0.0%	1.0%	0.0%	0.0%	79.4%	0.0%	81.2%	45.8%	0.0%	0.0%	85.1%	58.2%	73.3%	27.4%	39.5%	58.2%	39.1%	0.0%	34.6%	24.6%

		Eyeglasses		Fan		Faucet			FoldingChair		Globe		Kettle		Keyboard		KitchenPot		Knife		
Algorithm	Setting	leg	body	rotor	frame	switch	base	spout	seat	leg	sphere	frame	lid	body	base	key	lid	body	blade	body	blade
Mask-	RGB	51.2%	85.2%	54.4%	67.5%	52.5%	47.9%	99.7%	90.6%	46.1%	98.0%	71.1%	75.2%	99.4%	15.0%	17.5%	99.0%	94.5%	11.7%	88.5%	33.4%
RCNN	RGB-D	49.2%	84.9%	39.4%	67.4%	52.1%	55.8%	98.9%	93.8%	47.2%	96.0%	69.6%	94.1%	100.0%	8.8%	5.1%	100.0%	95.0%	10.0%	77.8%	34.5%
PartNet	XYZ	62.1%	93.8%	50.9%	74.8%	34.4%	55.9%	64.2%	91.2%	79.4%	83.0%	77.6%	71.1%	74.1%	6.8%	1.0%	94.6%	94.4%	3.1%	80.1%	9.4%
InsSeg	XYZRGB	80.6%	92.4%	42.0%	63.5%	29.9%	64.1%	78.0%	86.3%	75.6%	79.0%	82.0%	87.2%	90.7%	4.0%	1.0%	93.5%	95.0%	5.0%	82.7%	10.1%

		Lamp			Laptop		Lighter				Microwave			Mouse			Oven				
Algorithm	Setting	base	rot. bar	head	base	screen	wheel	button	body	rot. lid	door	body	button	button	wheel	body	door	knob	body	tr. tray	button
Mask-	RGB	54.6%	14.6%	64.5%	51.9%	93.1%	35.0%	80.8%	96.8%	97.0%	53.8%	94.0%	0.0%	0.0%	46.5%	98.0%	54.0%	49.9%	86.8%	1.0%	0.0%
RCNN	RGB-D	48.8%	10.8%	69.5%	47.2%	92.8%	57.2%	94.1%	89.2%	92.1%	49.5%	97.1%	0.0%	1.0%	45.3%	95.2%	53.4%	42.3%	93.3%	1.0%	0.0%
PartNet	XYZ	51.8%	8.8%	38.5%	93.0%	97.7%	1.0%	0.0%	77.4%	80.9%	25.9%	45.8%	0.0%	1.0%	0.0%	76.0%	23.1%	0.0%	36.6%	1.0%	0.0%
InsSeg	XYZRGB	50.6%	9.3%	39.7%	89.8%	96.1%	9.5%	61.4%	82.5%	84.9%	24.3%	48.7%	0.0%	1.0%	1.0%	61.1%	26.9%	0.0%	49.1%	0.0%	0.0%

		Pen			Phone		Pliers	Printer		Refrigerator		Remote		Safe			Scissors	Stapler		
Algorithm	Setting	cap	body	button	button	base	leg	button	body	body	door	button	base	knob	button	body	door	leg	body	lid	base
Mask-	RGB	94.1%	91.0%	52.8%	18.4%	51.4%	79.9%	2.8%	87.1%	83.0%	60.7%	35.6%	75.2%	34.1%	0.0%	88.5%	68.5%	34.2%	32.1%	60.2%	84.6%
RCNN	RGB-D	94.1%	96.2%	57.6%	12.8%	50.2%	78.7%	1.5%	72.3%	81.2%	55.0%	25.6%	78.2%	24.5%	0.0%	92.1%	74.6%	57.4%	33.6%	75.0%	90.5%
PartNet	XYZ	67.9%	98.0%	53.0%	1.0%	38.0%	37.9%	0.0%	34.8%	30.0%	16.2%	1.0%	63.2%	0.0%	0.0%	40.5%	30.5%	20.6%	31.7%	49.2%	76.7%
InsSeg	XYZRGB	15.0%	96.2%	25.4%	0.0%	27.0%	46.0%	0.0%	48.5%	40.2%	27.7%	1.0%	75.9%	0.0%	0.0%	60.8%	42.3%	36.4%	28.5%	83.3%	89.5%

		Suitcase					Switch				Table			Toaster				Toilet	
Algorithm	Setting	rot. handle	body	tr. handle	wheel	caster	frame	lever	button	slider	drawer	body	wheel	door	caster	knob	slider	body	button	lever	lid
Mask-	RGB	25.5%	81.7%	74.3%	6.2%	0.0%	85.9%	24.3%	73.6%	60.8%	54.3%	88.0%	3.4%	6.3%	0.0%	40.1%	39.0%	90.1%	5.9%	51.6%	98.3%
RCNN	RGB-D	36.4%	97.3%	70.0%	18.1%	0.0%	74.0%	26.0%	65.8%	22.8%	58.6%	89.9%	1.4%	13.2%	0.0%	40.6%	33.0%	94.1%	4.0%	36.4%	98.0%
PartNet	XYZ	3.7%	53.7%	63.6%	1.4%	0.0%	52.3%	2.3%	4.9%	1.0%	15.7%	71.3%	1.7%	1.0%	0.0%	0.0%	9.9%	79.3%	0.0%	0.0%	69.3%
InsSeg	XYZRGB	4.3%	53.2%	64.5%	2.0%	0.0%	53.5%	1.0%	2.1%	1.7%	16.4%	81.8%	1.3%	2.0%	1.0%	2.6%	20.3%	72.9%	0.0%	0.0%	89.6%

		Toilet				TrashCan					USB			WashingMachine				Window		All
Algorithm	Setting	body	lid	seat	button	pad	lid	body	door	wheel	rotation	body	lid	door	knob	button	body	window	frame	mAP
Mask-	RGB	95.3%	64.3%	61.1%	8.9%	43.4%	68.1%	85.6%	35.9%	73.7%	59.8%	65.7%	71.3%	52.0%	6.8%	4.4%	53.5%	55.9%	12.2%		53.0%
RCNN	RGB-D	91.8%	64.4%	62.5%	3.0%	37.1%	69.7%	84.9%	29.7%	69.3%	74.4%	62.8%	68.6%	41.4%	4.0%	0.0%	73.3%	48.7%	13.4%		52.8%
PartNet	XYZ	83.2%	17.6%	1.4%	0.0%	12.7%	67.1%	57.7%	12.1%	5.5%	30.0%	27.1%	22.2%	22.4%	0.0%	0.0%	30.5%	22.6%	83.5%		36.1%
InsSeg	XYZRGB	86.5%	25.1%	5.2%	0.0%	21.8%	75.4%	73.5%	3.9%	5.0%	23.9%	42.9%	12.2%	14.5%	0.0%	0.0%	22.9%	24.0%	85.2%		37.1%

$\hat{\mathbf{d}}^t \in [-1,1]^3$, direction of a predicted translational axis.

$\hat{x}_{\text{door}} \in [0,1]$, predicted joint position for a door.

$\hat{x}_{\text{drawer}} \in [0,1]$, predicted joint position for a drawer.

In the following, letters without hat indicates their corresponding ground-truth labels.

In our experiment, we modify the input layer of a ResNet50 network to accept 5 channels, and output layer to output 13 numbers. In addition, we apply tanh activation to produce $\hat{\mathbf{d}}^r, \hat{\mathbf{d}}^t$, and sigmoid activation to produce $\hat{x}_{\text{door}}, \hat{x}_{\text{drawer}}$. The loss has 7 terms:

Axis alignment loss, measured by cosine distance:

$$L_{dr} = \sum_{T_r=1} 1 - |\frac{\mathbf{d}^r \cdot \hat{\mathbf{d}}^r}{||\mathbf{d}^r|| ||\hat{\mathbf{d}}^r||}| \quad L_{dt} = \sum_{T_t=1} 1 - |\frac{\mathbf{d}^t \cdot \hat{\mathbf{d}}^t}{||\mathbf{d}^t|| ||\hat{\mathbf{d}}^t||}|$$

Pivot loss, measured by the distance from predicted pivot to ground truth joint axis:

$$L_p = \sum_{T_r=1} ||\hat{\mathbf{p}}^r - \mathbf{p}^r - ((\hat{\mathbf{p}}^r - \mathbf{p}^r) \cdot \mathbf{d}^r)\mathbf{d}^r||_2^2$$

Joint type prediction loss:

$$L_{T_r} = -\sum T_r \log \hat{T}_r + (1 - T_r) \log(1 - \hat{T}_r)$$

$$L_{T_t} = -\sum T_t \log \hat{T}_t + (1 - T_t) \log(1 - \hat{T}_t)$$

Joint position loss, L_2 loss between predicted position and ground truth position.

$$L_{\text{door}} = \sum_{\text{valid hinge}} (x_{\text{door}} - \hat{x}_{\text{door}})^2$$

$$L_{\text{drawer}} = \sum_{\text{valid slider}} (x_{\text{drawer}} - \hat{x}_{\text{drawer}})^2$$

The final loss is a summation of all the losses above:

$$L = L_{dr} + L_{dt} + L_p + L_{T_r} + L_{T_t} + L_{\text{door}} + L_{\text{drawer}}$$

This objective is optimized on mini-batches using proper masking based on H and S values.

We repeat this experiment with PointNet++[112] operating on 3D RGB-point cloud produced by the same images. For each image, we sample 10,000 points from the partial point cloud (create random copies if the total number of points is less than 10,000). Figure 2.8 shows the network structure for the motion recognition tasks.

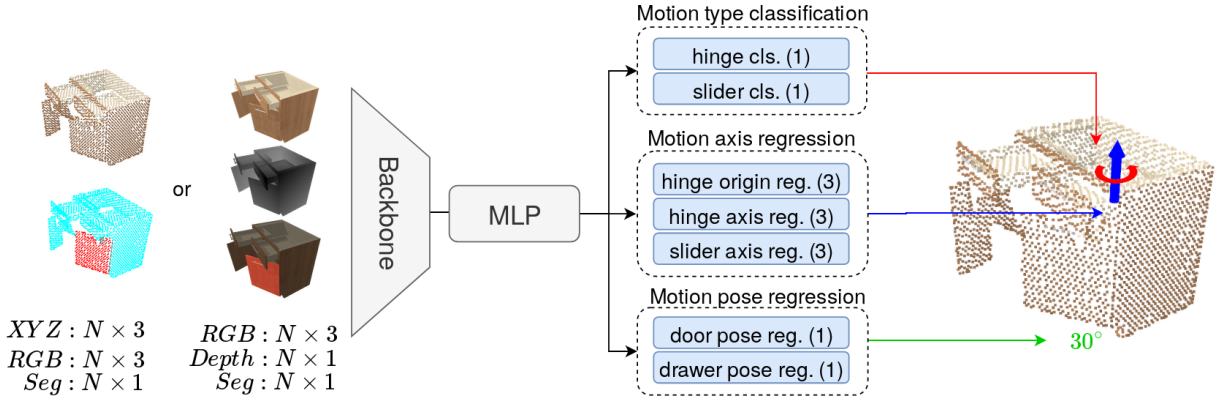


Figure 2.8. Vision Tasks. Show 2 vision task definitions: inputs + outputs.

2.7 Conclusion

We present SAPIEN, a simulation environment for robotic vision and interaction tasks, which provides detailed part-level physical simulation, hierarchical robotics controllers and versatile rendering options. We demonstrate that SAPIEN enables a large variety of robotic perception and interaction tasks.

2.8 Terminology

SAPIEN Engine

- **Articulation:** An articulation is composed of a set of links connected together with transnational or rotational joints [100]. The most common articulation is a robot.
- **Kinematic/Dynamic joint system:** Both joint systems are an assembly of rigid bodies connected by pairwise constraints. Kinematic system does not respond to external forces while dynamic objects do.
- **Force/Joint/Velocity Controller:** Controller which can control the force/position/velocity of one or multiple joints at once. Like real robot, controller may fail depending on whether the target is reachable.

- **Inertial Measurement Unit(IMU):** A sensor which can measure the orientation, acceleration and angular velocity of the mounted link.
- **Trajectory Controller:** A controller which receive trajectory command and execute to move through the trajectory points. Note that trajectory consist of a sequence of position, velocity and acceleration, while path is simply a set of points without a schedule for reaching each point [56].
- **End-effector:** End-effector is a manipulator that performs the task required of the robot, The most common end-effector is gripper.
- **Inverse Kinematics:** Determine the joint position corresponding to a given end-effector position and orientation [128].
- **Inverse Dynamics:** Determining the joint torques which are needed to generate a given motion. Usualy, the input of inverse dynamics is the output of inverse kinematics or motion planning.

SAPIEN Renderer

- **GLSL** is OpenGL's shading language with describes how the GPU draws visuals.
- **Rasterization** is the process of converting shapes to pixels. It is the pipeline used by most real-time graphics applications.
- **Ray tracing** is a rendering technique by simulating light-rays, reflections, refractions, etc. It can achieve physically accurate images at the cost of rendering time. OptiX is Nvidia's GPU based ray-tracing framework.

Acknowledgements

Chapter 2, in full, is a reprint of the material published in the 2020 Conference on Computer Vision and Pattern Recognition (CVPR): "SAPIEN: A Simulated Part-based Interactive

Environment" (Fanbo Xiang, Yuzhe Qin, Kaichun Mo, Yikuan Xia, Hao Zhu, Fangchen Liu, Minghua Liu, Hanxiao Jiang, Yifu Yuan, He Wang, Li Yi, Angel X. Chang, Leonidas J. Guibas, Hao Su). The dissertation author was the primary investigator and author of this paper.

This research was supported by NSF grant IIS-1764078, NSF grant IIS-1763268, a Vannevar Bush Faculty Fellowship, the Canada CIFAR AI Chair program, gifts from Qualcomm, Adobe, and Kuaishou Technology, and grants from the Samsung GRO program and the SAIL Toyota Research Center.

Chapter 3

Simulation Asset Acquisition by Neural Capturing

Recent work [89, 8] has demonstrated that volumetric scene representations combined with differentiable volume rendering can enable photo-realistic rendering for challenging scenes that mesh reconstruction fails on. However, these methods entangle geometry and appearance in a "black-box" volume that cannot be edited. Instead, we present an approach that explicitly disentangles geometry—represented as a continuous 3D volume—from appearance—represented as a continuous 2D texture map. We achieve this by introducing a 3D-to-2D texture mapping (or surface parameterization) network into volumetric representations. We constrain this texture mapping network using an additional 2D-to-3D inverse mapping network and a novel cycle consistency loss to make 3D surface points map to 2D texture points that map back to the original 3D points. We demonstrate that this representation can be reconstructed using only multi-view image supervision and generates high-quality rendering results. More importantly, by separating geometry and texture, we allow users to edit appearance by simply editing 2D texture maps.

3.1 Introduction

Capturing and modeling real scenes from image inputs is an extensively studied problem

Research partially done When F. Xiang was an intern at Adobe Research.

in vision and graphics. One crucial goal of this task is to avoid the tedious manual 3D modeling process and directly provide a renderable and editable 3D model that can be used for realistic rendering in applications, like e-commerce, VR and AR. Traditional 3D reconstruction methods [123, 124, 62] usually reconstruct objects as meshes. Meshes are widely used in rendering pipelines; they are typically combined with mapped textures for appearance editing in 3D modeling pipelines.

However, mesh-based reconstruction is particularly challenging and often cannot synthesize highly realistic images for complex objects. Recently, various neural scene representations have been presented to address this scene acquisition task. Arguably the best visual quality is obtained by approaches like NeRF [89] and Deep Reflectance Volumes [8] that leverage differentiable volume rendering (ray marching). However, these volume-based methods do not (explicitly) reason about the object's surface and entangle both geometry and appearance in a volume-encoding neural network. This does not allow for easy editing—as is possible with a texture mapped mesh—and significantly limits the practicality of these neural rendering approaches.

Our goal is to make volumetric neural reconstruction more practical by enabling both realistic image synthesis and flexible surface appearance editing. To this end, we present NeuTex—an approach that explicitly disentangles scene geometry from appearance. NeuTex represents geometry with a volumetric representation (similar to NeRF) but *represents surface appearance using 2D texture maps*. This allows us to leverage differentiable volume rendering to reconstruct the scene from multi-view images, while allowing for conventional texture-editing operations (see Fig. 3.1).

As in NeRF [89], we march a ray through each pixel, regress volume density and radiance (using fully connected MLPs) at sampled 3D shading points on the ray, accumulate the per-point radiance values to compute the final pixel color. NeRF uses a single MLP to regress both density and radiance in a 3D volume. While we retain this volumetric density-based representation for geometry, NeuTex represents radiance in a 2D (UV) texture space. In particular, we train a

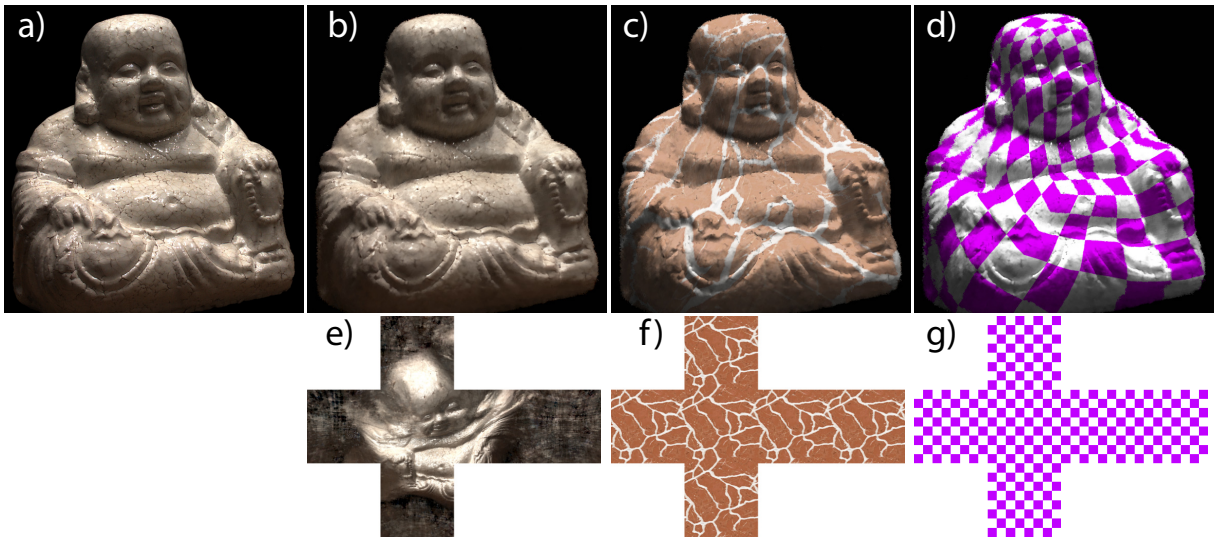


Figure 3.1. NeuTex is a neural scene representation that represents geometry as a 3D volume but appearance as a 2D neural texture in an automatically discovered texture UV space, shown as a cubemap in (e). NeuTex can synthesize highly realistic images (b) that are very close to the ground-truth (a). Moreover, it enables intuitive surface appearance editing directly in the 2D texture space; we show an example of this in (c), by using a new texture (f) to modulate the reconstructed texture. Our discovered texture mapping covers the object surface uniformly, as illustrated in (d), by rendering the object using a uniform checkerboard texture (g).

texture mapping MLP to regress a 2D UV coordinate at every 3D point in the scene, and use another MLP to regress radiance in the 2D texture space for any UV location. Thus, given any 3D shading point in ray marching, our network can obtain its radiance by sampling the reconstructed neural texture at its mapped UV location.

Naively adding a texture mapping network to NeRF (and supervising only with a rendering loss) leads to a degenerate texture mapping that does not unwrap the surface and cannot support texture editing (see Fig. 3.3). To ensure that the estimated texture space reasonably represents the object's 2D surface, we introduce a novel cycle consistency loss. Specifically, we consider the shading points that contribute predominantly to the pixel color along a given ray, and correspond to the points either on or close to the surface. We train an additional *inverse mapping* MLP to map the 2D UV coordinates of these high-contribution points *back* to their 3D locations. Introducing this inverse-mapping network forces our model to learn a consistent mapping (similar to a one-to-one correspondence) between the 2D UV coordinates and the 3D

points on the object surface. This additionally regularizes the surface reasoning and texture space discovery process. As can be seen in Fig. 3.1, our full model recovers a reasonable texture space, that can support realistic rendering similar to previous work while also allowing for intuitive appearance editing.

Our technique can be incorporated into different volume rendering frameworks. In addition to NeRF, we show that it can be combined with Neural Reflectance Fields [7] to reconstruct BRDF parameters as 2D texture maps (see Fig. 3.7), enabling both view synthesis and relighting.

Naturally, NeuTex is more constrained than a fully-volumetric method; this leads to our final rendering quality to be on par or slightly worse than NeRF [89]. Nonetheless, we demonstrate that our approach can still synthesize photo-realistic images and significantly outperform both traditional mesh-based reconstruction methods [124] and previous neural rendering methods [130, 129]. Most importantly, our work is the first to recover a meaningful surface-aware texture parameterization of a scene and enable surface appearance editing applications (as in Fig. 3.1 and 3.6). This, we believe, is an important step towards making neural rendering methods useful in 3D design workflows.

3.2 Related Work

Scene representations. Deep learning based methods have explored various classical scene representations, including volumes [59, 155, 111, 129], point clouds [110, 2, 149], meshes [61, 150], depth maps [76, 53] and implicit functions [24, 87, 99, 166]. However, most of them focus on geometry reconstruction and understanding and do not aim to perform realistic image synthesis. We leverage volumetric neural rendering [89, 7] for realistic rendering; our method achieves higher rendering quality than other neural rendering methods [129, 130].

Mesh-based reconstruction and rendering. 3D polygonal meshes are one of the most popular geometry representations, widely used in 3D modeling and rendering pipelines. Numerous

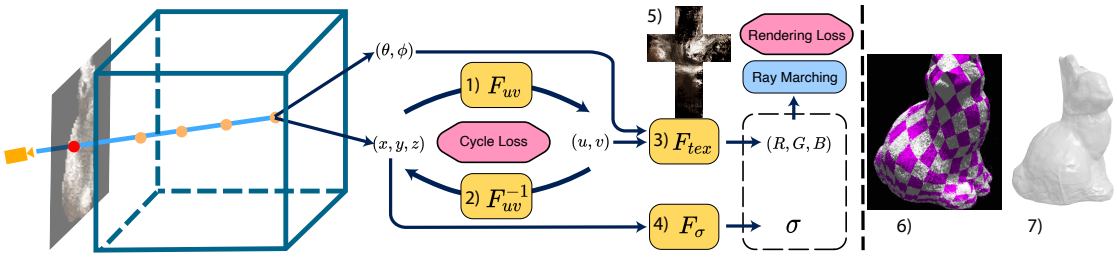


Figure 3.2. Overview. We present a disentangled neural representation consisting of multiple MLPs for neural volumetric rendering. As in NeRF [89], for geometry we use an MLP (4) F_{σ} to regress volume density σ at any 3D point $\mathbf{x} = (x, y, z)$. In contrast, for appearance, we use a texture mapping MLP (1) F_{σ} to map 3D points to 2D texture UVs, $\mathbf{u} = (u, v)$, and a texture network (3) F_{tex} to regress the 2D view-dependent radiance in the UV space given a UV $\mathbf{u}$ and a viewing direction $\mathbf{d} = (\theta, \phi)$. One regressed texture (for a fixed viewing direction) is shown in (5). We also train an inverse mapping MLP (2) F_{uv}^{-1} that maps UVs back to 3D points. We leverage a cycle loss (Eqn. 3.12) to ensure consistency between the 3D-to-2D mapping F_{uv} and the 2D-to-3D F_{uv}^{-1} mapping at points on the object surface. This enables meaningful surface reasoning and texture space discovery, as illustrated by (6, 7). We demonstrate the meaningfulness of the UV space learned by F_{uv} (6) by rendering the object with a uniform checkerboard texture. We also show the result of the inverse mapping network (7) by uniformly sampling UVs in the texture space and unprojecting them to 3D using F_{uv}^{-1}, resulting in a reasonable mesh.

traditional 3D reconstruction techniques have been proposed to directly reconstruct a mesh from multiple captured images, including structure from motion [123], multi-view stereo [36, 66, 124], and surface extraction [79, 62]. Recently, many deep learning based methods [147, 165, 139, 21, 25] have also been proposed, improving the reconstruction quality in many of these techniques. In spite of these advances, it is still challenging to reconstruct a mesh that can directly be used to synthesize photo-realistic images. In fact, many image-based rendering techniques [12, 6, 48] have been presented to fix the rendering artifacts from mesh reconstruction; however, they often leverage view-dependent texture maps [30], which cannot be easily edited. We instead leverage volumetric neural rendering to achieve realistic image synthesis; our approach explicitly extracts surface appearance as view-independent textures, just like standard textures used with meshes, allowing for broad texture editing applications in 3D modeling and content generation.

Neural rendering. Recently, deep learning-based methods have proposed to ameliorate or completely bypass mesh reconstruction to achieve realistic neural renderings of real scenes

for view synthesis [170, 159, 130, 129], relighting [160, 107, 23], and many other image synthesis tasks [78]. In particular, NeRF [89], Deep Reflectance Volumes [8] and other relevant works [7, 77] model a scene using neural volumetric representations (that encode geometry and appearance) and leverage differentiable volume rendering [85] to synthesize highly photo-realistic images. However, these volume representations do not explicitly reason about the 2D surface of a scene and are essentially "black-box" functions that cannot be easily modified after reconstruction. In contrast, we introduce a novel neural scene representation that offers direct access to the 2D surface appearance in volumetric neural rendering. Our representation has disentangled geometry and appearance components, and models appearance as a 2D neural texture in a auto-discovered texture space. Unlike previous volumetric neural rendering methods, this allows for easy texture/appearance editing.

Learning textures. Texture mapping is a standard technique widely used with meshes. Here, surface appearance is represented by a 2D texture image and a 3D-to-2D mapping from every mesh vertex to the texture space. Textures can be easily controlled and edited by artists as needed to create diversities of scene appearance. Recently, many deep learning based methods leverage texture-based techniques to model geometry or appearance in a scene [49, 42, 142]. Many works learn a 2D texture for a mesh by assuming a known mesh template [61, 119, 39], focusing on reconstruction problems for specific object categories. Our approach works for arbitrary shapes, and we instead learn a 2D texture in a volume rendering framework, discovering a 2D surface in the 3D volume space. Thies et al. [142] optimize neural textures to do rendering for a known fixed mesh with given UV mapping. In contrast, our approach simultaneously reconstructs the scene geometry as a volume, discovers a 2D texture UV space, and regresses a neural texture in the self-discovered texture space. Other methods learn appearance by regressing colors directly from 3D points [101, 102], which requires a known mesh and does not provide a 2D UV space necessary for texture editing.

AtlasNet [42] and follow-up work [108] train neural networks to map 2D UV coordinates

into 3D locations (like an inverse texture mapping), modeling an object 2D surface as an unwrapped atlas. These works focus on learning generalized geometry representations, and cannot be directly applied to arbitrary shapes or used for realistic rendering. Our network instead discovers a cycle mapping between a 2D texture space and a 3D volume, learning both a texture mapping and an inverse mapping. We leverage differentiable volume rendering to model scene appearance from captured images for realistic rendering. We also show that our neural texture mapping, supervised using a rendering loss and a cycle mapping loss, can discover a more uniform surface than a simple AtlasNet, supervised by a noisy point cloud from COLMAP [124] (see Fig. 3.8).

3.3 Neural Texture Mapping

3.3.1 Overview

We now present the NeuTex scene representation and demonstrate how to use it in the context of volumetric neural rendering. While NeuTex can enable disentangled scene modeling and texture mapping for different acquisition and rendering tasks, in this section, we demonstrate its view synthesis capabilities with NeRF [89]. An extension to reflectance fields (with [7]) is discussed in Sec. 3.5.4.

As shown in Fig. 3.2, our method is composed of four learned components: $F_\sigma, F_{\text{tex}}, F_{\text{uv}}$ and F_{uv}^{-1}. Unlike NeRF, which uses a single MLP, NeuTex uses a disentangled neural representation consisting of three sub-networks, which encode scene geometry (F_σ), a texture mapping function (F_{uv}), and a 2D texture (F_{tex}) respectively (Sec. 3.3.2). In addition, we use an inverse texture mapping network (F_{uv}^{-1}) to ensure that the discovered texture space reasonably explains the scene surfaces (Sec. 3.3.3). We introduce a cycle mapping loss to regularize the texture mapping and inverse mapping networks, and use a rendering loss to train our neural model end-to-end to regress realistic images (Sec. 3.3.4).

3.3.2 Disentangled neural scene representation

Volume rendering. Volume rendering requires volume density σ and radiance $\mathbf{c}$ at all 3D locations in a scene. A pixel's radiance value (RGB color) $\mathbf{I}$ is computed by marching a ray from the pixel and aggregating the radiance values $\mathbf{c}_i$ of multiple shading points on the ray, as expressed by:

$$\mathbf{I} = \sum_i T_i(1 - \exp(-\sigma_i \delta_i))\mathbf{c}_i, \tag{3.1}$$

$$T_i = \exp(-\sum_{j=1}^{i-1} \sigma_j \delta_j), \tag{3.2}$$

where $i = 1, ..., N$ denotes the index of a shading point on the ray, δ_i represents the distance between two consecutive points, T_i is known as the transmittance, and $\mathbf{c}_i$ and σ_i are the volume density (extinction coefficient) and radiance at shading point i. The above ray marching process is derived as a discretization of a continuous volume rendering integral; for more details, please see previous work [85].

Radiance field. In the context of view synthesis, a general volume scene representation can be seen as a 5D function (i.e. a radiance field, as referred to by [89]):

$$F_{\sigma,\mathbf{c}} : (\mathbf{x}, \mathbf{d}) \rightarrow (\sigma, \mathbf{c}), \tag{3.3}$$

which outputs volume density and radiance $(\sigma, \mathbf{c})$ given a 3D location $\mathbf{x} = (x, y, z)$ and viewing direction $\mathbf{d} = (\theta, \phi)$. NeRF [89] proposes to use a single MLP network to represent $F_{\sigma,\mathbf{c}}$ as a neural radiance field and achieves photo-realistic rendering results. Their single network encapsulates the entire scene geometry and appearance as a whole; however, this "bakes" the scene content into the trained network, and does not allow for any applications (e.g., appearance editing) beyond pure view synthesis.

Disentangling $F_{\sigma,\mathbf{c}}$. In contrast, we propose explicitly decomposing the radiance field $F_{\sigma,\mathbf{c}}$ into

two components, F_σ and $F_\mathbf{c}$, modeling geometry and appearance, respectively:

$$F_\sigma : \mathbf{x} \rightarrow \sigma, \qquad F_\mathbf{c} : (\mathbf{x}, \mathbf{d}) \rightarrow \mathbf{c}. \tag{3.4}$$

In particular, F_σ regresses volume density (i.e., scene geometry), and $F_\mathbf{c}$ regresses radiance (i.e., scene appearance). We model them as two independent networks.

Texture mapping. We further propose to model scene appearance in a 2D texture space that explains the object's 2D surface appearance. We explicitly map a 3D point $\mathbf{x} = (x, y, z)$ in a volume onto a 2D UV coordinate $\mathbf{u} = (u, v)$ in a texture, and regress the radiance in the texture space given 2D UV coordinates and a viewing direction $(\mathbf{u}, \mathbf{d})$. We describe the 3D-to-2D mapping as a texture mapping function F_{uv} and the radiance regression as a texture function F_{tex}:

$$F_{\text{uv}} : \mathbf{x} \rightarrow \mathbf{u}, \qquad F_{\text{tex}} : (\mathbf{u}, \mathbf{d}) \rightarrow \mathbf{c}. \tag{3.5}$$

Our appearance function $F_\mathbf{c}$ is thus a composition of the two functions:

$$F_\mathbf{c}(\mathbf{x}, \mathbf{d}) = F_{\text{tex}}(F_{\text{uv}}(\mathbf{x}), \mathbf{d}). \tag{3.6}$$

Neural representation. In summary, our full radiance field is a composition of three functions: a geometry function F_σ, a texture mapping function F_{uv}, and a texture function F_{tex}, given by:

$$(\sigma, \mathbf{c}) = F_{\sigma,\mathbf{c}}(\mathbf{x}, \mathbf{d}) = (F_\sigma(\mathbf{x}), F_{\text{tex}}(F_{\text{uv}}(\mathbf{x}), \mathbf{d})). \tag{3.7}$$

We use three separate MLP networks for F_σ, F_{uv} and F_{tex}. Unlike the black-box NeRF network, our representation has disentangled geometry and appearance modules, and models appearance in a 2D texture space.

3.3.3 Texture space and inverse texture mapping

As described in Eqn. 3.5, our texture space is parameterized by a 2D UV coordinate $\mathbf{u} = (u, v)$. While any continuous 2D topology can be used for the UV space in our network, we use a 2D unit sphere for most results, where $\mathbf{u}$ is interpreted as a point on the unit sphere.

Directly training the representation networks (F_σ, F_{uv}, F_{tex}) with pure rendering supervision often leads to a highly distorted texture space and degenerate cases where multiple points map to the same UV coordinate, which is undesirable. The ideal goal is instead to uniformly map the 2D surface onto the texture space and occupy the entire texture space. To achieve this, we propose to jointly train an "inverse" texture mapping network F_{uv}^{-1} that maps a 2D UV coordinate $\mathbf{u}$ on the texture to a 3D point $\mathbf{x}$ in the volume:

$$F_{\text{uv}}^{-1} : \mathbf{u} \rightarrow \mathbf{x}. \tag{3.8}$$

F_{uv}^{-1} projects the 2D texture space onto a 2D manifold (in 3D space). This inverse texture mapping allows us to reason about the 2D surface of the scene (corresponding to the inferred texture) and regularize the texture mapping process. We leverage our texture mapping and inverse mapping networks to build a cycle mapping (a one-to-one correspondence) between the 2D object surface and the texture space, leading to high-quality texture mapping.

3.3.4 Training neural texture mapping

We train our full network, consisting of F_σ, F_{tex}, F_{uv}, and F_{uv}^{-1}, from end to end, to simultaneously achieve surface discovery, space mapping, and scene geometry and appearance inference.

Rendering loss. We directly use the ground truth pixel radiance value $\mathbf{I}_{\text{gt}}$ in the captured images to supervise our rendered pixel radiance value $\mathbf{I}$ from ray marching (Eqn. 3.1). The rendering

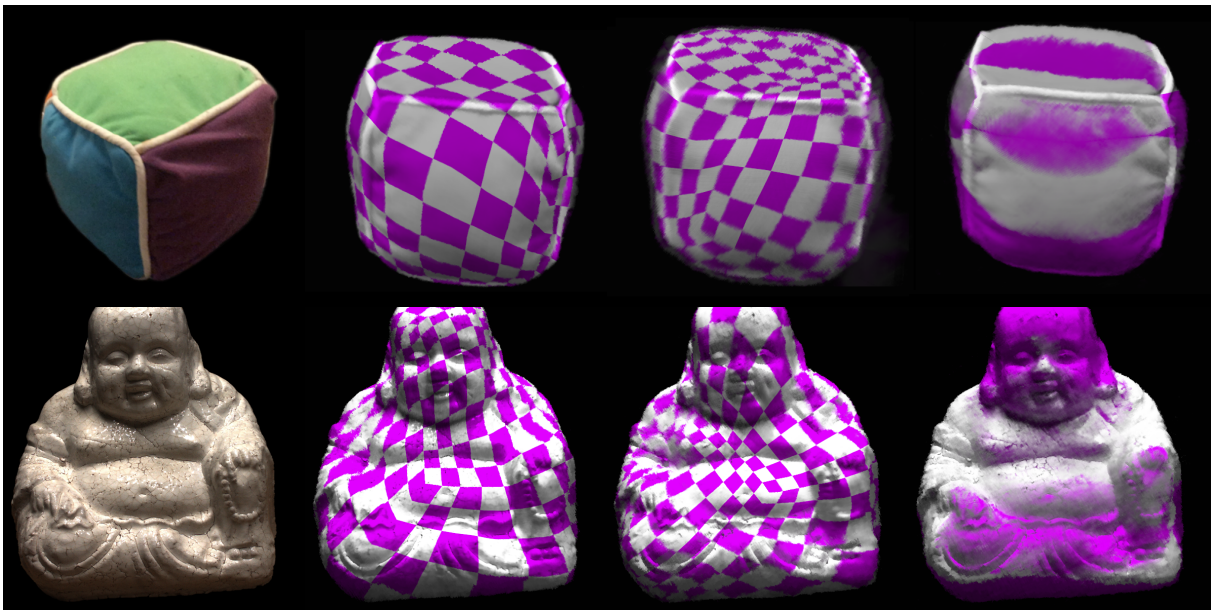


Figure 3.3. A checkerboard texture applied to scenes (a). When trained with or without initialization using coarse point cloud (b,c), the learned texture space is relatively uniform compared to trained without F_{uv}^{-1} and cycle loss (d).

loss for a pixel ray is given by:

$$L_{\text{render}} = \|\mathbf{I}_{\text{gt}} - \mathbf{I}\|_2^2. \tag{3.9}$$

This the main source of supervision in our system.

Cycle loss. Given any sampled shading point $\mathbf{x}_i$ on a ray in ray marching, our texture mapping network finds its UV $\mathbf{u}_i$ in texture space for radiance regression. We use the inverse mapping network to map this UV $\mathbf{u}_i$ back to the 3D space:

$$\mathbf{x}_i' = F_{\text{uv}}^{-1}(F_{\text{uv}}(\mathbf{x}_i)). \tag{3.10}$$

We propose to minimize the difference between $\mathbf{x}_i'$ and $\mathbf{x}_i$ to enforce a cycle mapping between the texture and world spaces (and force F_{uv}^{-1} to learn the inverse of F_{uv}).

However, it is unnecessary and unreasonable to enforce a cycle mapping at any 3D point.

We only expect a correspondence between the texture space and points on the 2D *surface* of the scene; enforcing the cycle mapping in the empty space away from the surface is meaningless. We expect 3D points near the scene surface to have high contributions to the radiance. Therefore, we leverage the radiance contribution weights per shading point to weigh our cycle loss. Specifically, we consider the weight:

$$w_i = T_i(1 - \exp(-\sigma_i \delta_i)), \tag{3.11}$$

which determines the contribution to the final pixel color for each shading point i in the ray marching equation 3.1. Equation 3.1 can be simply written as $\mathbf{I} = \sum_i w_i \mathbf{c}_i$. This contribution weight w_i naturally expresses how close a point is to the surface, and has been previously used for depth inference [89]. Our cycle loss for a single ray is given by:

$$L_{\text{cycle}} = \sum_i w_i \| F_{\text{uv}}^{-1}(F_{\text{uv}}(\mathbf{x}_i)) - \mathbf{x}_i \|_2^2. \tag{3.12}$$

Mask loss. We also additionally provide a loss to supervise a foreground-background mask. Basically, the transmittance (Eqn. 3.2) of the the last shading point T_N on a pixel ray indicates if the pixel is part of the background. We use the ground truth mask M_{gt} per pixel to supervise this by

$$L_{\text{mask}} = \| M_{\text{gt}} - (1 - T_N) \|_2^2. \tag{3.13}$$

We found this mask loss is necessary when viewpoints do not cover the object entirely. In such cases, the network can use the volume density to darken (when the background is black) renderings and fake some shading effects that should be in the texture. When the view coverage is dense enough around an object, this mask loss is often optional.

Full loss. Our full loss function L during training is:

$$L = L_{\text{render}} + a_1 L_{\text{cycle}} + a_2 L_{\text{mask}}. \tag{3.14}$$

We use $a_1 = 1$ for all our scenes in our experiments. We use $a_2 = 1$ for most scenes, except for those that already have good view coverage, where we remove the mask loss by setting $a_2 = 0$.

3.3.5 Extension to reflectance fields

NeuTex can be incorporated into different volume rendering pipelines. We have introduced applying it to NeRF for view synthesis in Secs. 3.3.2 to 3.3.4. We now briefly discuss combining it with the recent Neural Reflectance Fields [7] work that reconstructs BRDFs in volume rendering from flash images.

Instead of directly outputting radiance $\mathbf{c}$ at each shading point, this work [8] regresses normal $\mathbf{n}$ and reflectance parameters $\mathbf{r}$ at each shading point, and introduces a reflectance-aware volume rendering that computes radiance from these shading properties under given viewing and lighting condition. We correspondingly modify our geometry network F_σ to jointly regress volume density and normal, and change the texture regression network F_{tex} to regress the reflectance parameters in the texture space. Our central texture mapping and inverse mapping networks remain the same for this case. The modified network naturally provides the required volume properties in the reflectance-aware volume rendering process. We show that our neural texture mapping can enable high-quality BRDF texture extraction in this setting.

3.4 Implementation Details

3.4.1 Network details

All four sub-networks, F_σ, F_{tex}, F_{uv}, and F_{uv}^{-1}, are designed as MLP networks. We use unit vectors to represent viewing direction $\mathbf{d}$ and UV coordinate $\mathbf{u}$ (for spherical UV). As proposed by NeRF, we use positional encoding to infer high-frequency geometry and appearance details. In particular, we apply positional encoding for our geometry network F_σ and texture network F_{tex} on all their input components including $\mathbf{x}$, $\mathbf{u}$ and $\mathbf{d}$. On the other hand, since the texture mapping is expected to be smooth and uniform, we do not apply positional encoding on

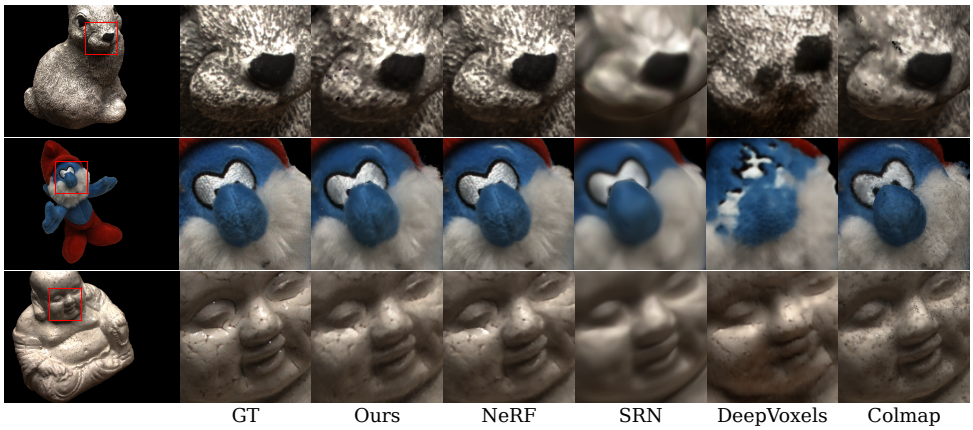


Figure 3.4. Comparisons on DTU scenes. Note how visually close our method is to the state-of-the-art, while enabling editing.

the two mapping networks.

3.4.2 Training details

Before training, we normalize the scene space to the unit box. When generating rays, we sample shading points on each pixel ray inside the box. For all our experiments, we use stratified sampling (uniform sampling with local jittering) to sample 256 point on each ray for ray marching. For each iteration, we randomly select a batch size of 600 to 800 pixels (depending on GPU memory usage) from an input image; we take 2/3 pixels from the foreground and 1/3 pixels from the background.

Our inverse mapping network F_{uv}^{-1} maps the 2D UV space to a 3D surface, which is functionally similar to AtlasNet [42] and can be trained as such, if geometry is available. We thus initialize F_{uv}^{-1} with a point cloud from COLMAP [124] using a Chamfer loss. However, since the MVS point cloud is often very noisy, this Chamfer loss is only used during this initialization phase. We find this initialization facilitates training, though our network still works without it for most cases (see Fig. 3.3). Usually, this AtlasNet-style initialization is very sensitive to the MVS reconstruction noise and leads to a highly non-uniform mapping surface. However, we find that our final inverse mapping network can output a much smoother surface as shown in Fig. 3.8, after jointly training with our rendering and cycle losses.

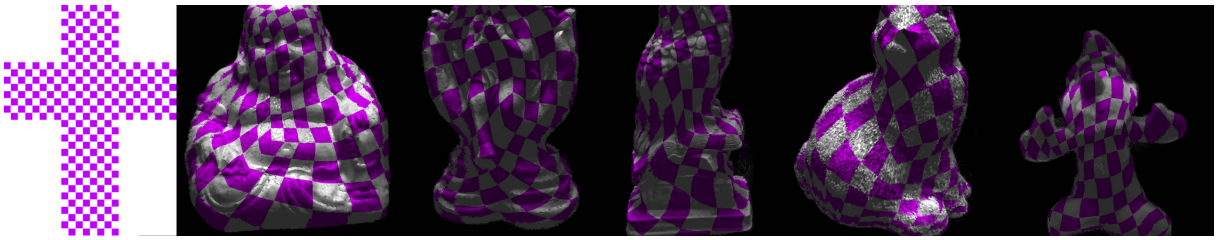


Figure 3.5. Apply checkerboard texture (left) to the DTU objects. Our approach discovers a uniform texture space.

Table 3.1. Average PSNR/SSIM for novel view synthesis on 4 held-out views on 5 DTU scenes.

Method	PSNR	SSIM
SRN [130]	26.05	0.837
DeepVoxels [129]	20.85	0.702
COLMAP [124]	24.63	0.865
NeRF[89]	**30.73**	**0.938**
Ours	**28.23**	**0.894**

Specifically, we initially train our method using a Chamfer loss together with a rendering loss for 50,000 iterations. Then, we remove the Chamfer loss and train with our full loss (Eqn. 3.14) until convergence, after around 500,000 iterations. Finally, we fine-tune our texture network F_{tex} until convergence, freezing the other networks (F_σ, F_{uv} and F_{uv}^{-1}), which is useful to get better texture details. The whole process takes 2-3 days on a single RTX 2080Ti GPU.

3.5 Results

We now show experimental results of our method and comparisons against previous methods on real scenes.

3.5.1 Configuration

We demonstrate our method on real scenes from different sources, including five scenes from the DTU dataset [1] (Fig. 3.1, 3.4, 3.6), two scenes from Neural Reflectance Fields [7] obtained from the authors (Fig. 3.7), and three scenes captured by ourselves (Fig. 3.6). Each DTU scene contains either 49 or 64 input images from multiple viewpoints. Each scene from

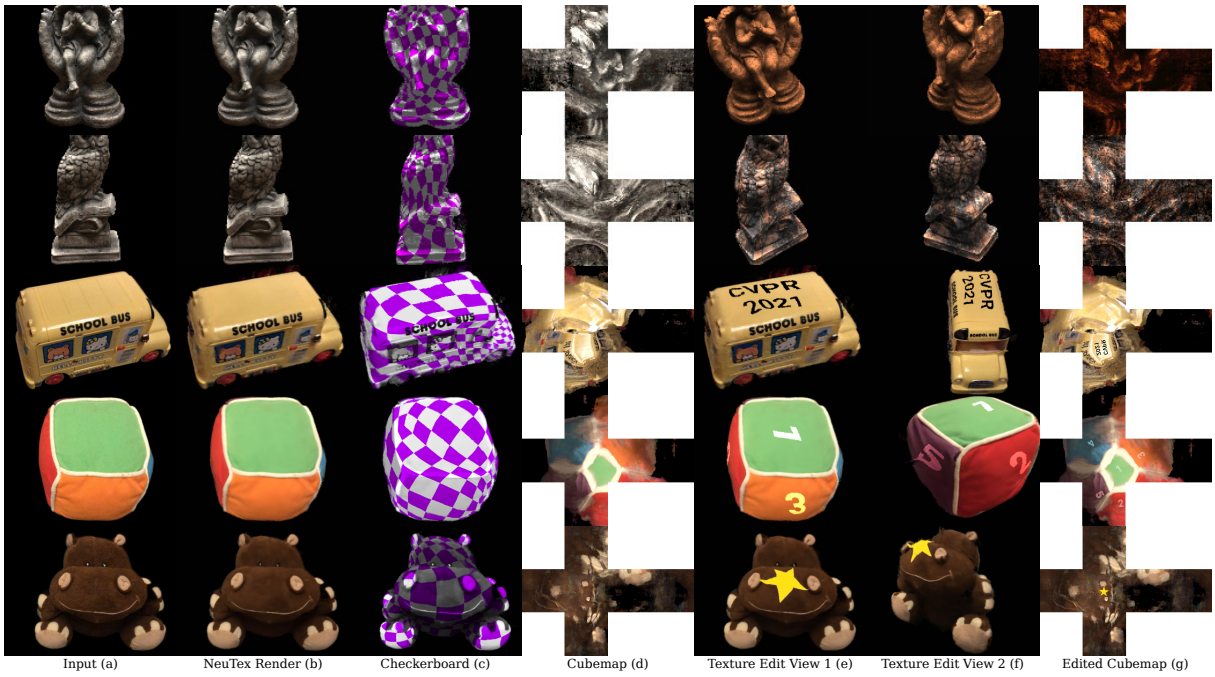


Figure 3.6. Texture editing on DTU (rows 1-2) and our (rows 3-5) scenes. Since our neural texture (F_{tex}) depends on a view direction, we show the cubemap texture (d) with pixelwise maximum values across views. Each texture is edited by multiplying a new specified texture; the resulting texture is shown in (g). The images rendered with the edited textures are shown from two different views in (e) and (f).

[7] contains about 300 images. Our own scenes each contain about 100 images. For our own data, we capture the images using a held-hanld cellphone and use the structure from motion implementation in COLMAP [124] for camera calibration. For other scenes, we directly use the provided camera calibration in the dataset. Since our method focuses on the capture and surface discovery of objects, we require the input images to have a clean, easily segmentable background. We use U2Net [113] to automatically compute masks for our own scenes. For the DTU scenes, we use the background masks provided by [166]. The images from [7] are captured under a single flash light, which already have very dark background; thus we do not apply additional masks for these images.

3.5.2 View synthesis results on DTU scenes

We now evaluate and compare our view synthesis results on five DTU scenes. In particular, we compare with NeRF [89], two previous neural rendering methods, SRN [130] and DeepVoxels [129], and one classical mesh reconstruction method COLMAP [124]. We use the released code from their authors to generate the results for all the comparison methods. For COLMAP, we skip the structure from motion, since we already have the provided camera calibration from the dataset. We hold-out 4 random views as testing views from the original 49 or 64 input views and run all methods on the remaining images for reconstruction.

We show qualitative visual comparisons on zoomed-in crops of testing images of two DTU scenes in Fig. 3.4, and quantitative comparison results of the averaged PSNRs and SSIMs on the testing images across five scenes in Tab. 3.1. Our method achieves high-quality view synthesis results as reflected by our rendered images being close to the ground truth and also our high PSNRs and SSIMs. Note that NeuTex enables automatic texture mapping that none of the other comparison methods can do. Even a traditional mesh-based method like COLMAP [124] needs additional techniques or tools to unwrap its surface for texture mapping, whereas our method unwraps the surface into a texture while doing reconstruction in a unsupervised way. To achieve this challenging task, NeuTex is designed in a more constrained way than NeRF. As a result, our rendering quality is quantitatively slightly worse than NeRF. Nonetheless, as shown in Fig. 3.4, our rendered results are realistic, reproduce many high-frequency details and qualitatively look very close to NeRF's results.

In fact, our results are significantly better than all other comparison methods, including both mesh-based reconstruction [124] and previous neural rendering methods [129, 130] in both qualitative and quantitative comparisons. In particular, COLMAP [124] can reconstruct reasonable shapes, but it cannot recover accurate texture details and intrinsically lacks view-dependent appearance effects (due to the Lambertian materials assumption). DeepVoxels [129] leverages a non-physically-based module for geometry and occlusion inference. While this

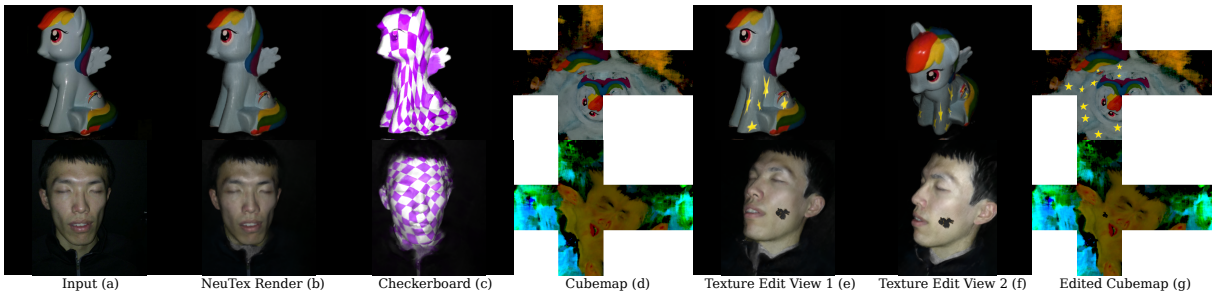


Figure 3.7. NeuTex in reflectance fields setting. We edit captured diffuse albedo (d) as shown in (g) to produce results shown in (e,f).

works well on scenes that have hundreds of input images, it does not work well on DTU scenes that have only about 40 to 60 images, leading to incorrect shapes and serious artifacts in their results. SRN [130], on the other hand, can reproduce reasonable shape in the rendering; however it cannot generate high-frequency appearance details like our method. Our approach is based on physically-based volume rendering, which models scene geometry and appearance accurately, leading to photo-realistic rendering results. More importantly, our approach achieves texture mapping and enables surface appearance editing in a 2D texture space that it automatically discovered, which cannot be done by NeRF nor any other previous neural rendering approaches.

3.5.3 Texture mapping and appearance editing

We now demonstrate our unique results on texture mapping and texture-space appearance editing that previous neural rendering approaches cannot achieve. Figure 3.6 shows such results on diverse real objects of DTU scenes and our own captured scenes. Our method can synthesize realistic view synthesis results (Fig. 3.6.b) that are very close to the ground truth. In addition, our method successfully unwraps the object surface into a reasonable texture (Fig. 3.6.d); the discovered texture space meaningfully expresses the 2D surface and distributes uniformly, as illustrated by the checkerboard rendering shown in Fig. 3.6.c.

Texture editing. Our high-quality texture mapping enables flexible appearance editing applications as shown in Fig. 3.6.e-g. In these examples, we show that we can use a specified full texture map to modulate the original texture, which entirely changes the object appearance. For

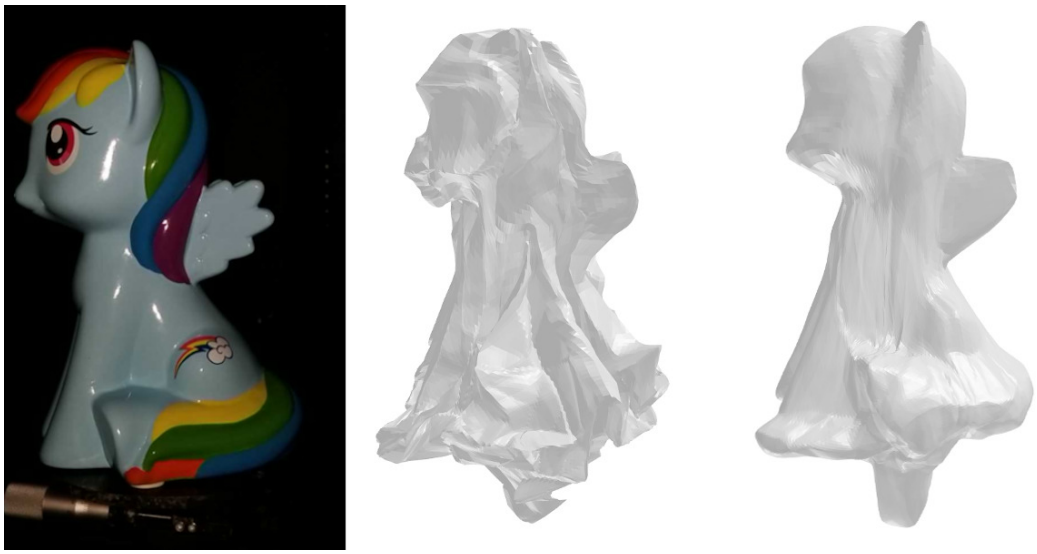


Figure 3.8. Our parametric surface (F_{uv}^{-1}) is strongly affected by noise in the COLMAP point cloud when trained with a Chamfer loss (center). Finetuning with our cycle and rendering losses without point cloud supervision (right) gives a smoother surface.

example, the object in the 1st row is interestingly changed from a stone-like object to a wooden one. We also demonstrate that we can locally modify the texture to add certain patterns on the object surface, such as the CVPR logo, the numbers, and the star in the last three rows. Note that, all our appearance editing is directly done in the texture space, which changes the essential surface appearance and naturally appears consistent across multiple view points, as shown in the rendered images from two different views in Fig. 3.6.e and f. Our NeuTex successfully disentangles the geometry and appearance of real objects and model the surface appearance in a meaningful texture space that explains the surface.

Inverse mapping. We further demonstrate the role of our inverse mapping network F_{uv}^{-1} in discovering this reasonable texture space. When we remove F_{uv}^{-1} from our network and train the system with only the rendering loss, the result always leads to a degenerate texture mapping, where large regions of 3D points are mapped to the same UV coordinates, as illustrated by the checkerboard-texture rendering shown in Fig. 3.3. In contrast, our full network with the cycle loss generally discovers a uniform space. As described in Sec. 3.4.2, we initialize our inverse network using a point cloud with a Chamfer loss; this is done mainly to help the network

converges quickly to a reasonable stage. In Fig. 3.3, we also show that our inverse network can still function well even without the point cloud initialization, using only the cycle loss. Note that, our initial point clouds come from an MVS reconstruction [124], which contains a lot of noise, leading to a noisy surface out of F_{uv}^{-1} as shown in Fig. 3.8. To prevent a degradation in the surface texture quality, we remove the supervision on this initial point cloud after initialization. Instead, our cycle loss can continue improving the noisy initialization and let the inverse mapping network F_{uv}^{-1} discover a smooth surface as shown in Fig. 3.8.

3.5.4 Extension to reflectance fields

NeuTex can be incorporated into different volume rendering pipelines. We now discuss combining it with the recent Neural Reflectance Fields [8, 7] that reconstructs BRDFs in volume rendering from flash images.

Instead of directly outputting radiance $\mathbf{c}$, [7] regresses normal $\mathbf{n}$ and reflectance parameters $\mathbf{r}$ at each shading point, and introduces a reflectance-aware volume rendering that computes radiance from these shading properties under given viewing and lighting condition. We correspondingly modify our geometry network F_σ to jointly regress volume density and normal, and change the texture regression network F_{tex} to regress the reflectance parameters in the texture space. Our central texture mapping and inverse mapping networks remain the same for this case. The modified network naturally provides the required volume properties in the reflectance-aware volume rendering process.

We show that our neural texture mapping can enable high-quality BRDF texture extraction in this setting in Fig. 3.7 on the two scenes from [7]. Our approach achieves realistic rendering (Fig. 3.7.b) that reproduces the original appearance, discovers a reasonably uniform texture space (Fig. 3.7.c), successfully unwraps the surface BRDFs into this space (as shown by the albedo maps in Fig. 3.7.d), and enables realistic rendering with BRDF editing in the texture space (Fig. 3.7.e-g). These results demonstrate the generality of our neural texture mapping framework and inspire potential future applications of our technique on other neural rendering tasks.

3.6 Clarification on Cube map Visualization

We briefly clarify our texture visualization (with cubemaps) used in our main paper. As discussed in Section 3.3.3 in the paper, we use spherical UVs for our texture mapping, where **u** represents a point on the surface a unit sphere. For all the figures in the main paper, we use cubemaps [41] to visualize the spherical domain. A cubemap consists of sixes faces of a unit box, recording all the color information projected from a unit sphere (as shown in Fig 3.9), which is widely used in graphics for spherical mapping. An alternative standard way to visualize a spherical function is to use a equirectangle map. We show the correspondence between a cube map and a equirectangle map in Fig. 3.10. We use cubemaps in the paper since they involve less distortion, avoiding the distorted regions in the top and bottom of equirectangle maps.

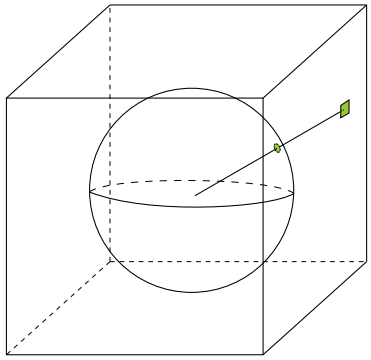


Figure 3.9. Cube map projection. The color at each point on the unit sphere is projected to a point on the cube centered at the origin. A cubemap is obtained by "opening up" the cube.

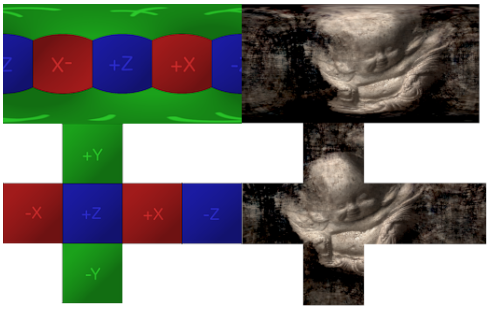


Figure 3.10. Cube maps on the second row corresponds to the equirectangle maps on the first row. They are different projections of the same spherical texture. A cubemap has a smaller distortion on the Y direction.

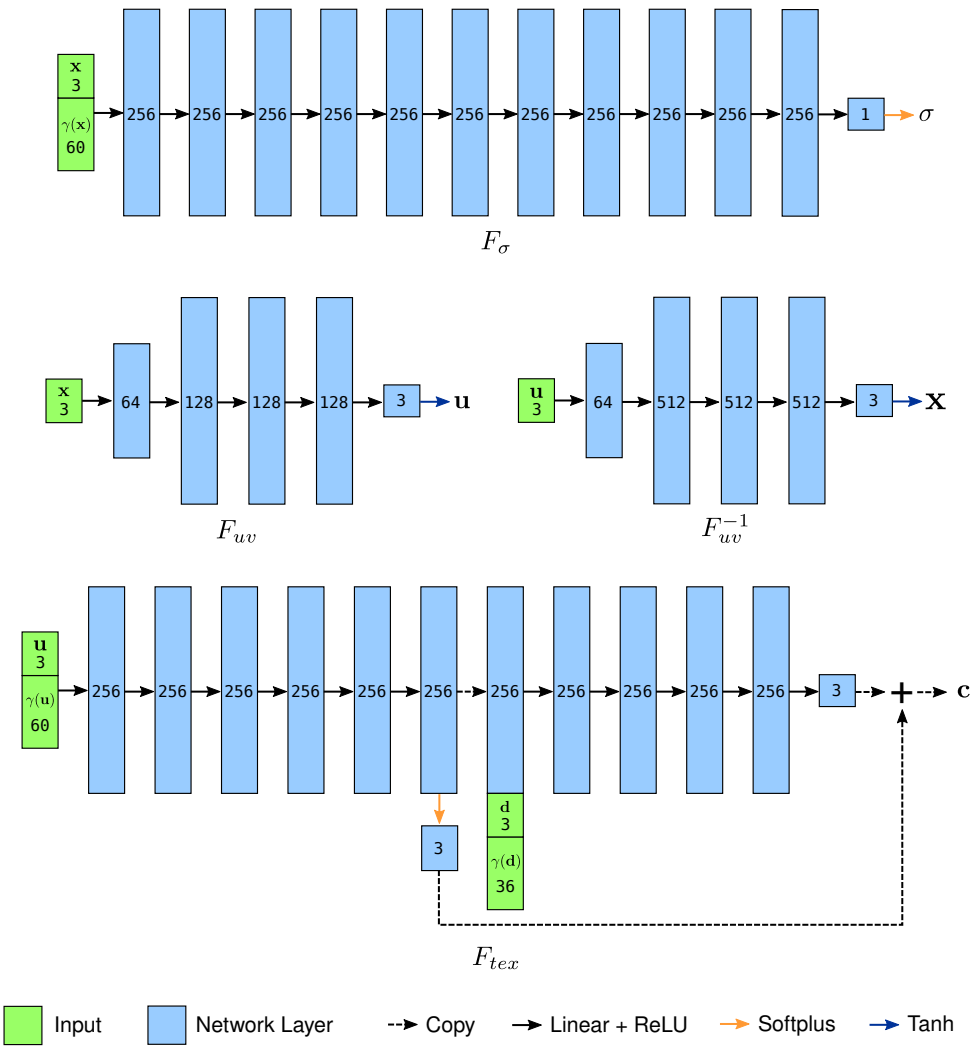


Figure 3.11. Network structure for the 4 networks. **x** represents 3D coordinates. γ denotes positional encoding. **u** represents texture-space points (3D points on the unit sphere). **d** represents a 3D unit vector for viewing direction. σ is predicted volumetric density. **c** is predicted radiance.

3.7 Network Implementation Details

3.7.1 Network structure

We show the detailed network architecture for $F_\sigma, F_{uv}, F_{uv}^{-1}$ and F_{tex} in Figure 3.11.

3.7.2 Training details in initialization

Here we describe in detail how we do the initialization stage mentioned in Sec 4.2. in the paper. Given a point cloud from Colmap, we first downsample it to one with 2,000-3,000 points.

We denote this point cloud as P_{gt}. We then sample 2,500 points uniformly in the UV space (the unit sphere). We denote the set of UV coordinates as P.

Chamfer loss. The Chamfer loss is simply the Chamfer distance between $F_{uv}^{-1}(P)$ and P_{gt}, where $F_{uv}^{-1}(P)$ corresponds to the point cloud generated by inverse-mapping very UV in P to the 3D space using the network F_{uv}^{-1}.

$$L_{\text{chamfer}} = \text{Chamfer}(F_{uv}^{-1}(P), P_{gt})$$

Inverse loss. We also leverage a loss that is similar to our cycle loss to let the initialization also influence the texture mapping network F_{uv}. In particular, instead of the 3D-to-2D-to-3D cycle mapping used the cycle loss in Eqn. 12 of the paper, we leverage a 2D-to-3D-to-2D cycle mapping in the initialization, given by:

$$L_{\text{cycle2}} = ||F_{uv}(F_{uv}^{-1}(P)) - P||_2^2$$

Rendering and mask loss. The same rendering and mask loss as described in section 4.1 are also applied in the initialization stage. So the loss at initialization stage is

$$L_{\text{init}} = L_{\text{chamfer}} + aL_{\text{cycle2}} + bL_{\text{render}} + cL_{\text{mask}}$$

where we set $a = 100$, $b = c = 1$.

3.8 Additional Results

3.8.1 Full quantitative comparison

We have shown the averaged quantitative results across five DTU scenes in Tab. 1 of the paper. Detailed comparisons on individual scenes are provided in Table 3.2. Similar to the average scores, though slightly worse than NeRF, our method significantly outperforms other



Figure 3.12. Comparison on the remaining DTU scenes.

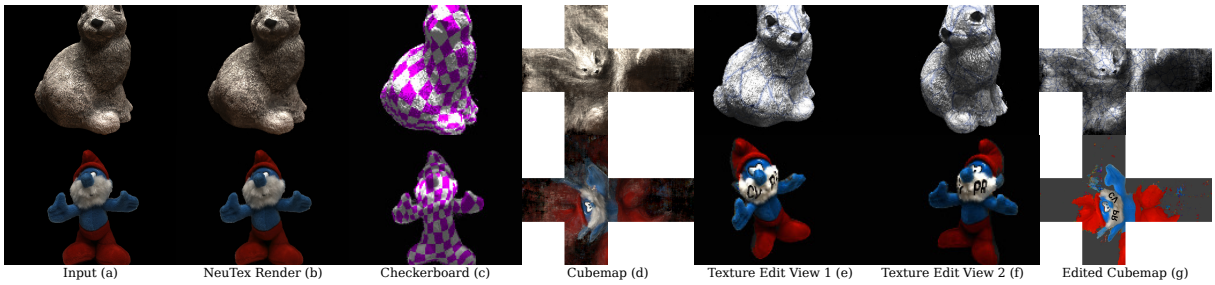


Figure 3.13. Additional texture editing on DTU scenes.

traditional and neural rendering methods.

3.8.2 Additional visual results

Figure 3.12 shows the visual comparison of different methods on the remaining 2 DTU scenes. Figure 3.13 shows additional texture editing results. Please refer to the attached video for more results on view synthesis and editing.

3.9 Conclusion

We have presented a novel approach that enables texture mapping in neural volumetric rendering. We introduce a novel disentangled neural scene representation that models geometry as a 3D volume and models appearance as a 2D texture in an automatically discovered texture space. We propose to jointly train a 3D-to-2D texture mapping network and a 2D-to-3D inverse mapping network to achieve surface reasoning and texture space discovery, using a surface-aware

Table 3.2. PSNR/SSIM for novel view synthesis quality on 4 held-out views on 5 DTU scenes.

	55		83		114		118		122	
Method	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM
SRN[130]	21.35	0.673	28.68	0.929	23.75	0.808	28.74	0.900	27.75	0.877
DeepVoxels[129]	17.21	0.532	23.76	0.858	17.97	0.606	23.18	0.764	22.12	0.748
COLMAP[124]	21.25	0.784	27.11	0.921	20.69	0.809	27.43	0.907	26.66	0.905
NeRF[89]	**26.78**	**0.913**	**31.77**	**0.952**	**27.38**	**0.918**	**33.98**	**0.954**	**33.72**	**0.955**
Ours	**22.67**	**0.808**	**30.61**	**0.931**	**26.45**	**0.891**	**30.67**	**0.916**	**30.75**	**0.925**

cycle consistency loss. As demonstrated, our approach can discover a reasonable texture space that meaningfully explains the object surface. Our method enables flexible surface appearance editing applications for neural volumetric rendering.

Acknowledgements

Chapter 3, in full, is a reprint of the material published in the 2021 Conference on Computer Vision and Pattern Recognition (CVPR): "NeuTex: Neural Texture Mapping for Volumetric Neural Rendering " (Fanbo Xiang, Zexiang Xu, Miloš Hašan, Yannick Hold-Geoffroy, Kalyan Sunkavalli, Hao Su). The dissertation author was the primary investigator and author of this paper.

This research was supported by gifts from Adobe, Kwai, and Qualcomm. This research was partially done when Fanbo Xiang was an intern at Adobe Research.

Chapter 4

Building a General Embodied AI Simulation Framework

SAPIEN is a simulation platform that offers high-performance GPU simulation and rendering, as well as realistic sensor simulation, to support the development of environments and applications that advance the field of embodied AI. In this report, we document the technical details of SAPIEN, including its core architecture, rigid and deformable body simulation systems, high-performance and high-quality rendering systems, realistic sensor simulation methods, example reinforcement learning environments, and software interfaces.

4.1 Introduction

The field of embodied AI focuses on the study of agents that perceive and interact with physical environments. Its cross-disciplinary nature necessitates concerted efforts from a wide range of fields. For instance, if we ask a home-assistant robot to *bring me a bottle of water from the fridge*, it needs to perceive the speech (audio processing), understand the sentence (natural language processing), recognize the fridge and the bottle (visual perception), formulate tasks to execute (planning), reach the fridge through the house (navigation), and use arms and grippers to open doors and grasp objects (control). The high system complexity, hardware cost, and time consumption of real-world embodied AI systems necessitate the use of simulators as testbeds to develop algorithms. In addition, simulators can offer higher flexibility and efficiency over

real-world systems. For example, Habitat [121, 137] and AI2-THOR [65] leverage simulators to abstract away detailed physical interactions and allow researchers to test high-level planning algorithms without developing low-level control; IsaacGym [81] and MuJoCo XLA [93] leverage the computational power of modern hardware to run virtual environments thousands of times faster than real-world speed.

Building simulation systems for embodied AI involves challenges from multiple aspects. 1) The simulation system needs to reproduce real-world physics to some extent, but there currently does not exist a unified framework to simulate common physical phenomena. 2) The simulation system needs to support efficient and realistic rendering to enable fast agent training and to mitigate gaps between simulated and real visual sensors. 3) The system should be equipped with an intuitive programming interface and user interfaces to enable researchers to swiftly prototype and design solutions. 4) The system needs to integrate assets and tasks to allow for the definition of meaningful research problems.

This paper describes our embodied AI simulation framework, SAPIEN [*], which addresses the aforementioned challenges by providing a modularized framework to allow the incorporation of latest progress in physical simulation, rendering, sensor modeling, object capturing, etc., towards building embodied AI environments.

This work is organized as follows. Section 4.2 describes SAPIEN's overall engine architecture. Section 4.3 discusses the physical simulation systems provided by SAPIEN and their interactions. Section 4.4 describes the highly efficient and extensible rendering system. Section 4.5 demonstrates manipulation benchmarks created in SAPIEN to support research on generalizable manipulation. Section 4.6 describes the programming interfaces, user interfaces, and extension systems.

[*] Specifically, this document outlines details of SAPIEN version 3.1, currently being actively developed

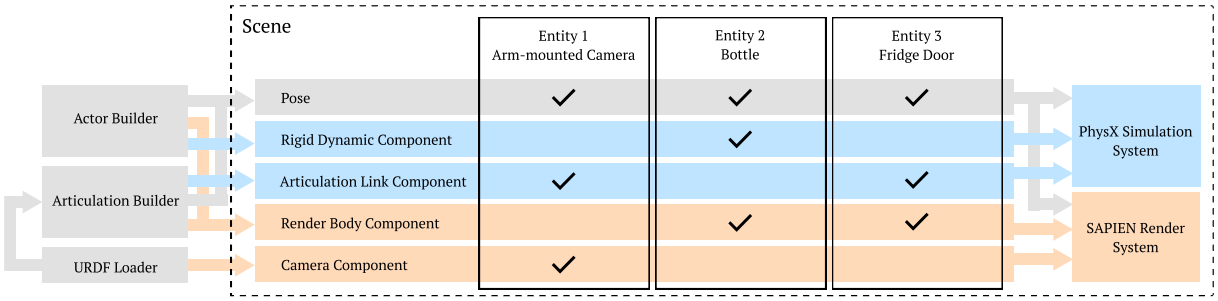


Figure 4.1. SAPIEN uses an Entity-Component System architecture to organize simulation objects in a scene. Each object in the virtual environment is represented as an *Entity* with different types of attached *Components*. For instance, a camera mounted on a robot arm is modeled as an entity with an articulation link component and a camera component. *Systems* update groups of components of specific types. For example, the render system takes camera components and render body components to generate sensor images. On top of the ECS organization, SAPIEN provides actor builders, articulation builders, and asset loaders to simplify the building process of common components. These builder APIs are agnostic to the implementation of components and systems.

4.2 Engine Architecture

SAPIEN's engine architecture has undergone multiple iterations of improvements through feedback from users and developers since the original release [158]. Most notably, SAPIEN 3 has transitioned its core engine architecture from an object-oriented design to an Entity-Component System (ECS), a software design pattern popularized by game engines including Unity [60] and Unreal [34]. While ECS is also commonly associated with data-oriented programming [68, 86, 126], SAPIEN does not enforce a data-oriented paradigm, and ECS is mainly a means for organizing code for flexibility and modularity.

SAPIEN additionally provides higher-level *builders* to offer a simpler interface for building common components. This builder design pattern allows SAPIEN to maintain a stable API when the underlying implementation of systems and components changes. An example of SAPIEN's ECS and builder organization is provided in Fig. 4.1. The relation between entities, components, and systems is summarized in Fig. 4.2.

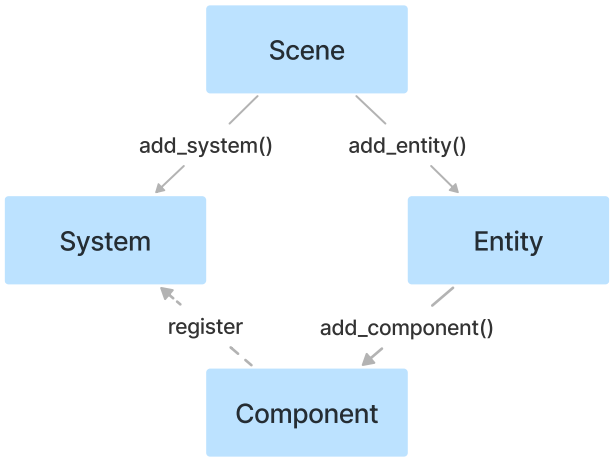


Figure 4.2. Relations between SAPIEN's Scene, Entity, Component, and System.

4.2.1 Entity-Component System

An *Entity* in SAPIEN represents a single object of any kind, such as robot links, tables, lights, and cameras. An entity on its own only defines its 6D pose (position and orientation) in space without any real functionalities. The functionalities of an entity are defined by components attached to it.

A *Component* represents a single functionality attachable to an entity. For instance, attaching the *PhysX Rigid Dynamic Component* indicates that the entity is simulated as a dynamic rigid body with the PhysX simulator; attaching the *Render Camera Component* indicates that the entity can be used to capture images of the visual world. Conceptually, an entity may contain an arbitrary number of components of the same or different types. This property also necessitates a compositional design such as ECS, as an object-oriented architecture would require a unique inheritance structure for any combination of component types.

A *System* performs actions over components of specific types. For example, the *PhysX System* updates all components related to the PhysX simulator; the *Render System* combines visual body components and camera components to render images.

Entities, components, and systems in SAPIEN interact within *Scenes*. Each scene represents a simulated world and contains a list of entities and systems. A system can be added

to multiple scenes to allow updating components across multiple scenes at once; this is especially useful in the case of the PhysX GPU system, which we discuss in section 4.3.1. An entity may only be added to a single scene, and a component may only be attached to a single entity. SAPIEN ties the creation and destruction of actual component functionalities to a scene. A *Component* is required to implement 2 lifecycle methods: *on_add_to_scene* and *on_remove_from_scene*. These methods are invoked at appropriate times when components are added to entities, entities are added to scenes, or components are enabled or disabled. Components typically register or unregister themselves from systems when executing the lifecycle methods so that the systems can keep track of which components need to be updated.

4.2.2 Builders and Asset Loaders

Despite the overhaul of core architecture from OOP to ECS, SAPIEN's world-building APIs have remained mostly unchanged due to an additional layer of builders for creating actors (physical and visual objects) and articulations (robots). Instead of directly creating entities and components, SAPIEN's users create an *Actor Builder* and specify object properties, including initial pose, collision geometry, visual shapes, mass, friction, etc., through a series of build steps. The builders separate entity creation from specific implementations of physical and visual components to maintain a stable API in the event of refactoring in the future.

In the robotics and graphics community, several file formats have been standardized to describe assets, such as collision meshes, visual shapes, and robots. For example, the widely used URDF format describes the kinematic structure, collision, visual, and sensor information about a robot. SAPIEN handles these assets at the builder level. Link and joint tags in a URDF file are parsed to articulation builders, and sensor tags are loaded as sensor components such as the camera component.

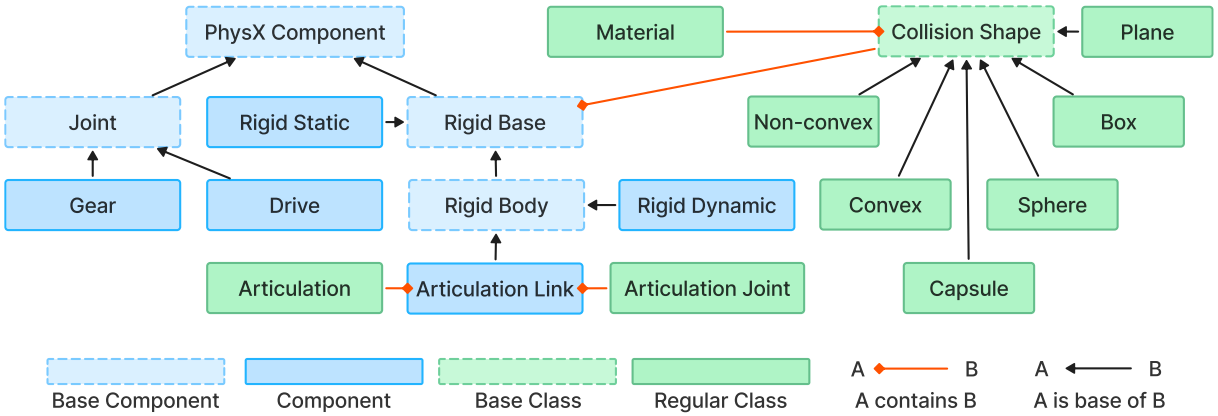


Figure 4.3. Overview of PhysX rigid body features exposed by SAPIEN, including physical bodies — static, dynamic, and articulated, physical constraints — gear and drive, collision geometry — plane, sphere, box, capsule, convex mesh, and non-convex mesh.

4.3 Multi-Physics Systems

Physical simulation sets the scope of robotic problems that can be explored in a virtual environment. Static scene navigation requires a static collision checker; object rearrangement with pick-and-place at least requires rigid body dynamics simulation; manipulating soft and deformable materials requires incorporating computational methods from solid and fluid mechanics.

SAPIEN offers 3 physical simulation systems: a rigid body simulation system for classical navigation and robotic manipulation problems based on PhysX [100] detailed in section 4.3.1, and deformable body simulators we developed in-house featuring finite-element method and material-point method in section 4.3.2.

4.3.1 PhysX Rigid Body Simulator

Among popular rigid body simulation frameworks including PhysX [100], Bullet [28], MuJoCo [144], and Drake [141], SAPIEN chooses PhysX as the rigid body simulation backend since it offers the most flexible API and the highest GPU simulation performance. In this section, we detail the supported simulation features and CPU/GPU interfaces in the rigid body simulation system.

Rigid Body features

SAPIEN exposes rigid body simulation features of PhysX to client applications as SAPIEN components, including physical body components (rigid static, rigid dynamic, and articulation link) and physical constraints such as drive and gear. We summarize the relation between common rigid body components in a class diagram in figure 4.3. The functionalities of these components are outlined below. We refer to PhysX documentation* for a more detailed understanding of the inner workings of the simulator.

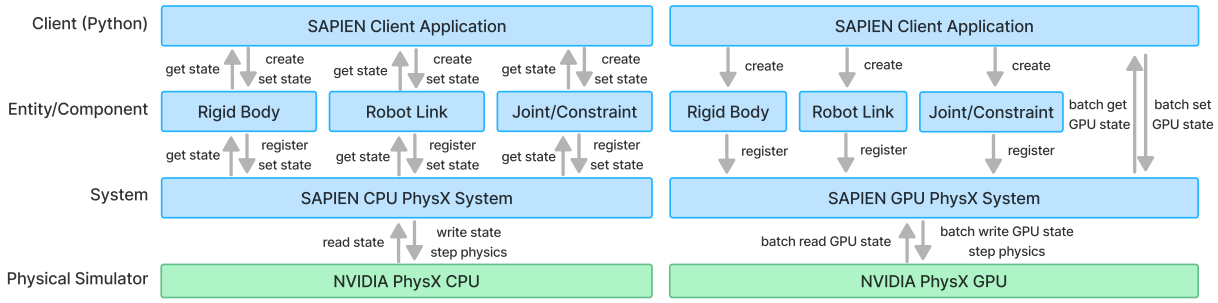
Physical Bodies

Different types of physical bodies differ in their dynamic behavior. *Static bodies* maintain a fixed pose during the simulation and do not respond to any forces. *Dynamic bodies* react to external forces such as gravity, contact force, or user-specified force. *Articulation links* are simulated with PhysX reduced-coordinate articulations. Many articulated links are connected with *articulation joints* to form tree-structured articulations. SAPIEN exposes 0-DOF fixed joints and 1-DOF revolute (hinge) and prismatic (slider) joints. These articulation joints can be controlled directly with joint torque, or by PhysX's built-in PD position/velocity controllers.

Collision Shapes

All 3 types of physical bodies above can have *collision shapes* attached to define their collision geometry, including 4 types of primitive shapes — plane, sphere, box, capsule, and 2 types of meshes — convex and non-convex. Collision detection for non-convex shapes is only supported by CPU static rigid body or any type of GPU body using PhysX's SDF-based collision detection feature [98]. A *physical material* is associated with each collision shape to define its surface properties including friction and restitution coefficients.

*https://nvidia-omniverse.github.io/PhysX/physx/5.3.1, accessed Aug. 1, 2024



(a) CPU PhysX Pipeline (b) GPU PhysX Pipeline

Figure 4.4. Illustration for SAPIEN's data-passing pipelines with NVIDIA PhysX CPU backend (a) and GPU backend (b). CPU and GPU backends share the same component creation and physical stepping logic but differ in how they get and set states. CPU system supports the most intuitive API and allows client applications to directly fetch and apply states of individual physical components. GPU system maintains batched simulation states for all components as GPU tensors and client applications directly work with GPU tensors to avoid CPU-GPU communication and achieve higher performance.

Physical Constraints

PhysX uses *Joints* to define physical constraints between physical bodies. While PhysX supports many types of joints such as revolute, prismatic spherical, gear, and D6 joints, some joints can be used to implement others. For example, the D6 joint allows defining constraints on arbitrary dimensions of the relative poses of jointed bodies and can be used to implement revolute, prismatic, and spherical joints. Therefore, SAPIEN exposes a minimal set of joints, including the D6 joint named *Drive* and the *Gear* joint.

CPU and GPU APIs

PhysX provides an open-source cross-platform CPU implementation, and a proprietary GPU implementation*. The GPU implementation features a *direct mode* allowing for data-oriented access to GPU buffers which enables us to design low-overhead systems that combine simulation, rendering, and agent training on the GPU. SAPIEN mirrors PhysX's design and provides a CPU PhysX system and a GPU PhysX system. How a SAPIEN client application constructs and communicates with PhysX systems is summarized in Figure 4.4 and discussed in

*GPU PhysX also supports some types of particle and FEM simulation. These are independent of the rigid body engine and thus not discussed here.

the following paragraphs.

Physical scene construction

Both the CPU PhysX system and GPU PhysX system share the same scene construction procedure. A client application creates physical components, attaches them to entities, and adds entities to scenes, preferably with builder APIs. A small distinction between CPU and GPU systems is that SAPIEN supports creating a separate CPU PhysX system per scene, whereas all scenes using the GPU PhysX system must share the same global system due to the internal design of PhysX. SAPIEN handles this by assigning each SAPIEN scene a configurable offset in the PhysX world. SAPIEN then hides this offset from client applications by automatically adding or subtracting it in all APIs.

CPU APIs

SAPIEN prioritizes simple and intuitive APIs for CPU rigid body simulation. The client application can directly read and write simulation states by invoking member functions on corresponding physical components, for example, **get_linear_velocity** and **set_force_torque** on dynamic bodies and **set_target_position** on articulation joints attached to articulation links. Setting and getting poses are handled slightly differently since pose is a common property shared by all entities instead of physical components. SAPIEN always keeps entity poses in sync with PhysX's internal poses by updating the internal poses immediately when the pose of an entity is modified, and it updates the poses of all entities immediately after the simulation is stepped.

GPU Tensor APIs

SAPIEN prioritizes performance for GPU rigid body simulation and exposes data-oriented APIs. Client applications fetch and write to simulation states via flat CUDA tensors, a design pioneered by Isaac Gym [81]. For example, the position, orientation, linear velocity, and angular velocity of all dynamic bodies and articulation links in all scenes are available in a single CUDA tensor; forces and torques can be applied to simulation bodies via the force/torque

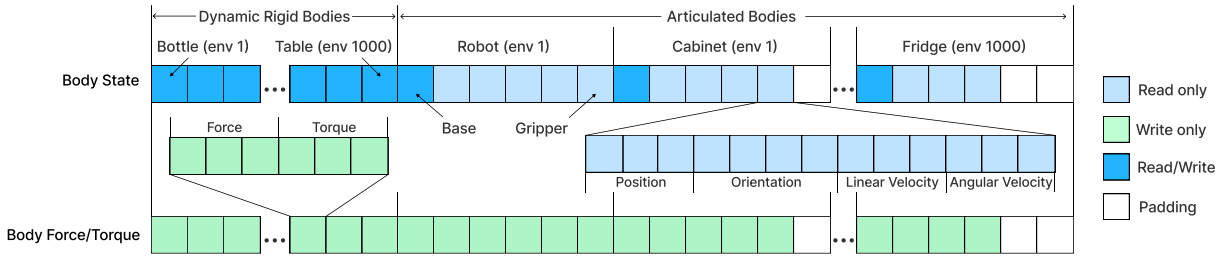


Figure 4.5. SAPIEN's PhysX GPU system uses global tensors to store simulation states. For example, the body states for dynamic rigid bodies and articulation links in all scenes are stored in a single body state tensor. All articulation states are padded to equal length so that link states can be retrieved per object or per articulation based on different views and slices. Similarly, the client application applies forces and torques to any rigid body by writing the global body force/torque tensor. These tensors can be accessed by most Python GPU libraries such as Torch, Jax, CuPy, and Numba.

tensor. Values in these tensors can be fetched and/or applied to the physical simulator depending on the tensor type and body type, as summarized in Figure 4.5.

SAPIEN further improves upon IsaacGym's design to support *heterogeneous* scenes. In IsaacGym and other existing frameworks using GPU PhysX, all scenes are required to share the same robots and objects, or at least assets with the same degrees of freedom for the scene states to be batched. However, this is not a fundamental limitation of the physical simulator. SAPIEN features a design that allows each scene to contain physical bodies of arbitrary numbers and types.

First, unlike IsaacGym which batches simulation states by scene, SAPIEN batches states of individual rigid bodies. States for bodies of the same scene do not need to be placed in consecutive memory. The body state tensor contains tightly packed states of all rigid dynamic bodies followed by all articulation links. Instead of tightly packing states of articulation links, SAPIEN allocates the same amount of state slots for each articulation based on the articulation with the maximum number of links. Link states are then filled into the first slots of each articulation, leaving the rest as padding. This design allows the body state tensor to be viewed by individual bodies or by articulations and simplifies data access. A client application could easily access data associated with a robot with a global articulation index, or data associated with a link

with a global body index. For example, imagine solving a generalizable cabinet door opening task, the application loads a different cabinet in each scene and needs to access the pose of a different cabinet door in each scene. It simply needs to maintain an index tensor of all doors and then index the global body state array with the index tensor. SAPIEN applies the same design principle to all CUDA tensors and enables the designing of parallel heterogeneous environments, a feature crucial for training generalizable agents in diverse environments.

4.3.2 Deformable Body Simulators

Rigid bodies constitute only a small portion of everyday objects. To support training agents to manipulate a wider range of materials, more advanced simulation techniques are required. For example, the finite-element method (FEM) subdivides 3D objects into tetrahedral or hexahedral elements with movable vertices to model deformation. FEM is suitable for soft bodies undergoing small deformables and requires significant modifications such as remeshing to support large deformation and fracture. Particle-based models and solvers such as smoothed-particle hydrodynamics (SPH) [133, 5] and material-point methods (MPM) [51] are often preferred in this case since they naturally handle large deformation and topology changes.

FEM System

SAPIEN provides a finite-element-method-based deformable mesh simulation through the Incremental-Potential Contact (IPC) [71] method, which guarantees non-penetration contacts at the cost of simulation speed. This IPC system is an optimized GPU implementation intended to support accurate simulation of soft contact sensors, where in our previous work [22], we have validated that IPC-based simulation can accurately and stably simulate deformations on visuotactile sensors, and reinforcement learning policies for peg insertion and lock opening using tactile signal as input trained on the FEM simulator can transfer to the real world. We refer to the original paper for details. In SAPIEN IPC, we support the Neo-Hookean [131] material model and solve incremental potential minimization with the Newton line search method. We

Table 4.1. Simulation parameters supported by SAPIEN IPC system.

Type	Parameter	Description
Scene	Time step	Simulation time step
	Gravity	Scene gravity
Constraint model	Distance threshold	Distance threshold for IPC Barrier
	Relative velocity threshold	IPC friction smoothing
	Maximum velocity	Maximum velocity for FEM vertices
	Barrier stiffness	IPC Barrier function stiffness
	Attachment stiffness	Constraint stiffness for point attachment
Linear solver (conjugate gradient)	Preconditioner	Enable Jacobi preconditioner for CG
	CG tolerance	Error threshold for CG
	Max iterations	Maximum CG iterations
Newton solver	Stopping threshold	Stopping threshold for Newton's method
	Max iterations	Max iterations for Newton's method
CCD parameters	CCD slackness	CCD slackness parameter
	max iterations	Max number of iterations in line search
Material model	Young's modulus	Compression stiffness for a tet element
	Poisson's ratio	Characterize Poisson effect for a tet element
	Density	Tet element density to compute mass
	Friction coefficient	Coulomb's friction coefficient for tet surface

implement a matrix-free parallel preconditioned conjugate gradient as the linear system solver and use additive continuous collision detection [72] to find the maximum line search step length. Here we summarize the parameters of the provided IPC system in Table 4.1.

MPM System

SAPIEN MPM system is a systematic GPU re-implementation of the MPM environments in our ManiSkill 2 based on the MLS-MPM algorithm [51]. We refer to the original ManiSkill 2 [43] paper for details.

SAPIEN MPM supports materials with 2 types of different yield criteria, von Mises and Drucker-Prager. Von Mises model is suitable for modeling clay-like materials such as plasticine. Drucker-Prager model is suitable for modeling granular material such as sand and soil. SAPIEN models contact between MPM material and rigid boundaries with a particle-based soft contact model where an elastic force is applied to particles penetrating object surfaces and friction is computed with Coulomb's model. The MPM particles are simulated in a fixed grid domain,

Table 4.2. Simulation parameters supported by SAPIEN MPM system.

Type	Parameter	Description
Scene	Time step	Simulation time step
	Gravity	Scene Gravity
Particle	Mass	Initial mass for a MPM particle
	Volume	Initial volume for a MPM particle
	Young's Modulus	Compression stiffness for a particle
	Poisson ratio	Characterize Poisson effect of a particle
	Type	Yield model: von Mises or Drucker-Prager
	Friction angle (Drucker-Prager)	Angle of internal friction for granular material
	Cohesion (Drucker-Prager)	Cohesive strength for granular material
Body	Contact stiffness	Stiffness for soft contact
	Contact damping	Damping for soft contact
	Contact friction	Contact friction coefficient
Grid	Grid dimension	Total dimension for MPM simulation domain
	Grid length	Side length of a grid cell



Figure 4.6. Deformable manipulation environments supported by SAPIEN MPM system, rendered with SAPIEN ray-tracer (section 4.4.2) after marching cubes on particles. Image extracted from ManiSkill 2 [43].

where greater cell length typically results in a more stable MPM system but with fewer details. We summarize exposed simulation parameters in Table 4.2. We show sample environments created with the MPM simulator in Figure 4.6.

Coupling between Rigid and Deformable Simulation

To allow using existing robots in the rigid body simulator to manipulate deformable bodies. A coupling mechanism is required to allow interaction between simulators. We have implemented a two-way dynamic coupling for rigid-MPM systems, and rigid-FEM coupling can be implemented with the same principle.

When simulation begins, SAPIEN copies collision shapes from the rigid body simulator to the deformable simulator. Primitive shapes (box, capsule, etc.) are represented as analytical

signed distance functions (SDFs) and meshes are converted to dense volume SDFs stored as GPU textures or sparse volume SDFs stored in NanoVDB format [96]. After each rigid body simulation step, poses and velocities of rigid bodies are copied to the deformable simulator to update the state of the SDF shapes. During deformable simulation, these SDF shapes are used as contact boundary conditions. Forces applied by the boundaries are recorded and accumulated as rigid body forces and torques, which are copied back to the rigid body simulator. This procedure is summarized in Figure 4.7.

While this coupling method does not solve rigid body and deformable body interactions in a unified scheme and can result in unstable contacts for stiff contact parameters, this issue can be mitigated by taking smaller time steps. Additionally, this method allows any two physical systems to couple at the geometric level, enabling users to experiment with multi-physics systems without modifying core simulation algorithms.



Figure 4.7. Two-way dynamic coupling between rigid and deformable physical systems. Rigid and deformable body simulators step alternatively. After each step, rigid body poses are transferred to the deformable body simulator, and deformable body forces are transferred to the rigid body simulator.

4.4 Unified Rendering System

Image rendering serves two crucial purposes in Embodied AI simulation systems. First, rendering is the foundation for visual sensor simulation, such as RGB cameras, stereo/time-of-flight depth sensors, and LiDARs. These simulated sensors are then used to test a wide variety of downstream tasks that require visual inputs, ranging from autonomous driving to navigation and object grasping. Second, rendered images can often provide privileged information unobtainable in real captures, such as normal and depth maps. Such information can be used as ground-truth information to develop vision and robotics algorithms, including depth estimation and object pose prediction.

While real-time rendering methods and photorealistic rendering techniques have been extensively researched over the past decades, especially in movie and game industries , rendering in embodied AI simulators faces several unique challenges (a) While previous real-time rendering applications typically target a fixed frame rate such as 30 FPS or 60 FPS, an embodied AI learning system can always benefit from improvements in rendering throughput and thus has more room for optimization. (b) Modern AI systems use batched environments, which involve rendering multiple (sometimes thousands of) scenes, which is a different workload compared to existing game or movie renderers, which are optimized for rendering a single scene with many objects. (c) Specific rendering algorithms still need to be developed to simulate sensor-specific effects, such as noise patterns in stereo and time-of-flight depth sensors. (d) The rapid advancement of 3D capture techniques and their associated 3D representations necessitates a highly customizable and flexible system to allow the implementation of new rendering techniques to support them.

In this section, we detail how SAPIEN renderer addresses these challenges above. Section 4.4.1 and Section 4.4.2 discuss the architecture for SAPIEN renderer and demonstrate how it improves the rendering performance of previous state-of-the-art by orders of magnitudes. Section 4.4.4 and Section 4.4.5 discuss our implementation of depth simulation algorithms that allows models trained on synthetic images to transfer to real-world ones.

4.4.1 Visual Learning System Optimization: End-to-end GPU Execution

The main goal of visual learning system optimization is to **maximize the total throughput**: the total number of samples (physical states and rendered images) collected over all simulation environments. An immediate solution to improve throughput is through parallelization. Classical visual learning frameworks, including robosuite [171], Habitat [121, 137], and AI2-THOR [65] parallelizes environments through multi-processing: executing a simulator that does CPU physics and GPU rendering per process as illustrated in Figure 4.8(a). While this strategy is the most flexible, it prevents CPU and GPU resource sharing across different environments. EnvPool [152] introduces a more efficient multi-threading model but requires implementing

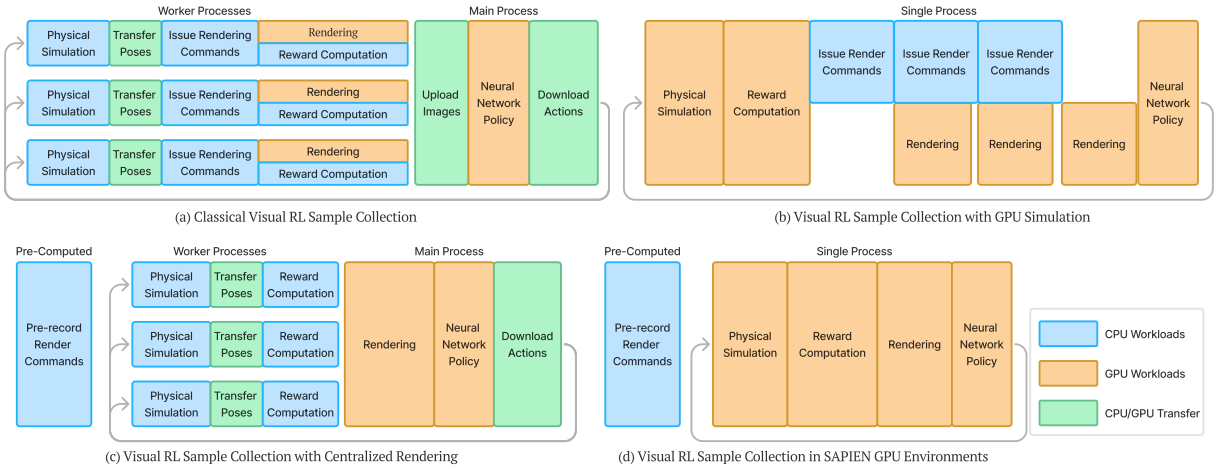


Figure 4.8. (a) Classical visual RL pipelines with multi-process simulation and rendering (Most flexible but low throughput). (b) Single-process GPU parallel simulation and classical rendering (bottlenecked by rendering command recording and execution). (c) Multi-process simulation, centralized rendering (high system complexity). (d) SAPIEN's fully optimized GPU environments.

environments in C++, which limits its adoption. In the previous version of SAPIEN, we introduced a render-server architecture (Figure 4.8(c)) to execute the physical simulation in multiple processes but rendering in a central process. We refer to [43] for more details. The render server enables developers to create highly efficient visual environments while maintaining the flexibility of using Python for building these environments. However, the server-client architecture involves complex communication protocols and incurs significant maintenance costs.

Recent advancements in GPU simulation [73, 98] and Python GPU frameworks [105, 9] have given rise to fully GPU-based learning environments such as Isaac Gym [81]. With the computation power of modern GPUs, Isaac Gym significantly accelerates physical simulation speed. However, it lacks native rendering support. Attempts to add rendering to Isaac Gym environments result in significant CPU overhead due to the issuing of rendering commands for thousands of scenes, as illustrated in Figure 4.8(b).

We identify the major bottlenecks in these GPU environment designs and present end-to-end GPU visual environment architecture(Figure 4.8(d)), which maximizes GPU utilization and minimizes CPU overhead with 3 key optimization techniques, *pre-recording*, *single-pass*

Table 4.3. Environment speedups from rendering techniques. PR: pre-recording. SP: single-scene multi-pass. CT: compute-based transfer. The baseline includes none of these techniques. Environment performance is measured in FPS (camera image frames per second).

Technique	Cartpole (RGB)		Cartpole (RGB+PCD)		Replica (RGB)		Replica (RGB+PCD)	
	128×128	512×512	128×128	512×512	128×128	512×512	128×128	512×512
Baseline	69.6K	11.7K	69.7K	11.7K	5.30K	3.91K	5.20K	3.60K
PR	142K	15.9K	141K	15.9K	60.8K	14.9K	53.8K	9.94K
PR+SP	218K	16.2K	217K	16.2K	69.5K	15.8K	60.3K	15.8K
PR+SP+CT	1246K	58.7K	1239K	58.6K	75.0K	20.9K	73.1K	19.1K
GPU Memory	2.26G	10.1G	2.26G	10.1G	2.57G	10.6G	2.57G	10.6G



Figure 4.9. Visualization for rendered RGB images (512×512) in Cartpole (left) and Replica (right).

multi-scene rendering, and *computed-based transfer*. We test the speedups for these techniques under different scene complexity, rendering modes, and image sizes. In all tests, RGB images are rendered in 4-channel uint8 format, and point clouds in 4-channel int16 format quantized to millimeters, and the last channel additionally stores object segmentation. The results are summarized in Table 4.3 and rendered RGB images are shown in Figure 4.9.

Pre-recording

Pre-recording refers to the technique of recording rendering commands once ahead of time, and reusing the recorded commands every time a frame needs to be rendered. In a legacy

OpenGL-style rendering pipeline (Figure 4.10(a)) widely used in modern simulators including MuJoCo [143] and Habitat [121], the renderer uploads data and issues draw calls for each rendered object. However, these operations are CPU intensive and may cause the GPU to idle because CPUs cannot dispatch draw calls fast enough. This issue is especially prominent since most embodied AI simulators do not support recording draw calls from multiple CPU threads. Even when they do, many GPU compute clusters lack the necessary CPU power to handle such workloads.

We observe that, unlike games where objects are constantly changing and require re-drawing, learning environments typically use the same set of objects throughout the episode. Consequently, the same set of drawing commands can be reused for all frames. Since SAPIEN 2 [43], our renderer pre-records the rendering commands for all objects and all cameras ahead of time (Figure 4.10(b)) and rendering a frame only involves updating GPU data and executing the pre-recorded commands, achieving state-of-the-art rendering performance at the time. Implementing pre-recording requires a modern graphics API and we use Vulkan for the best cross-platform compatibility. However, other rendering techniques, such as multi-draw and bindless rendering, can also address the same problem and have support in some OpenGL drivers. We will further discuss these techniques in section 4.4.2.

The benefit of pre-recording is best demonstrated when rendering complex scenes with many objects as shown in Table 4.3 Replica scenes, where pre-recording draw calls can provide $10\times$ speedups.

Single-pass multi-scene rendering

We further observe that when rendering simple environments, frequently switching render passes becomes a major bottleneck. In an ideal scenario, each render pass should execute as many draw calls as possible. Since batched learning environments typically use the same render process across all scenes and all cameras, it allows for all cameras to share a single render pass as long as they are rendered into different regions (specified by viewports) of the same set of

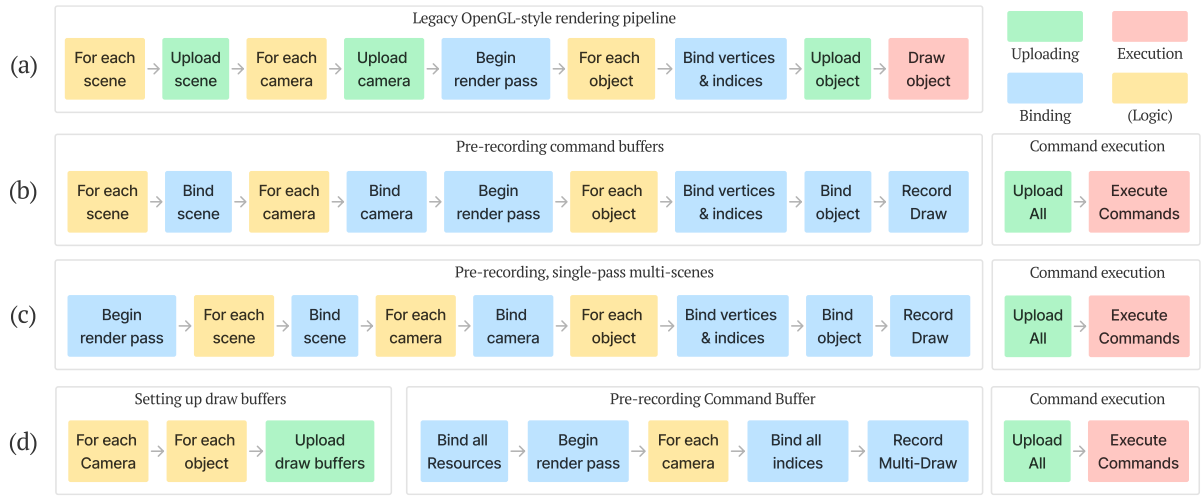


Figure 4.10. (a) Legacy CPU-bound OpenGL-style rendering pipelines upload resources and execute draw calls for each frame. (b) Pre-recording draw commands and executing them in each frame alleviate dependency on CPUs. (c) Single-pass multi-scene rendering further accelerates the rendering process with fewer render passes. (d) Rendering with bindless resources and multi-draw simplifies resource management and command recording without degradation in performance.

images (Figure 4.10(c)). With this technique implemented, we see a significant performance improvement in fast environments where rendering is fast and transfer is the bottleneck, for example, the Cartpole environment rendering FPS goes from from 142K to 218K FPS. A similar technique is also described in NVIDIA's concurrent work IsaacLab (successor of Orbit [90]), which they refer to as tiled rendering. However, IsaacLab's renderer only achieves about 60k FPS on the same setting. Apart from system optimizations, a major contributing factor to this performance difference is that IsaacLab adopts a single-scene architecture similar to game engines, where all objects and cameras of all environments are added to the same scene. As a result, every camera in IsaacLab issues draw calls for all objects in all environments, leading to wasted draw calls on invisible objects.

Compute-based transfer

When the rendering speed reaches 200K FPS for 128×128 images, we observe that transferring data from Vulkan images to Vulkan buffers, which are then exposed as CUDA tensors, has become the major bottleneck. We find through experiments that performing the same

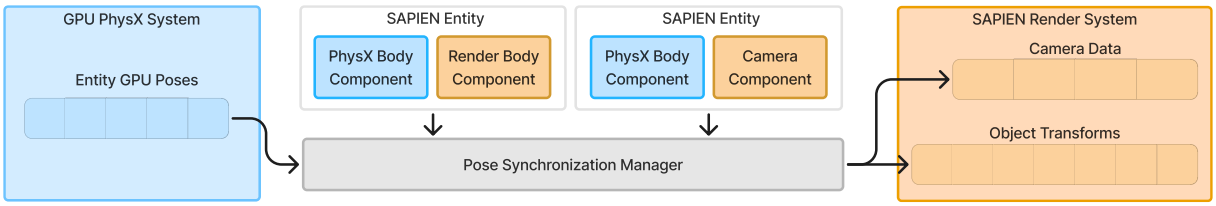


Figure 4.11. SAPIEN supports a direct data path to transfer PhysX body poses to render poses on the GPU. The Pose Synchronization Manager takes a list of entities with both PhysX body components and render body or camera components and caches the pose relation between them. Before each rendering step, the synchronization manager invokes a CUDA kernel to compute updated render poses based on PhysX poses and cached information.

data copying with compute shaders takes significantly less time than using copy commands. We believe this phenomenon is due to specific implementations in NVIDIA's GPU driver. Therefore, we replace the copying commands with compute commands to gain further performance improvements, allowing the simplest Cartpole environment to generate more than 1M 128×128 images per second.

Optimizing data transfer between GPU simulation and rendering

In addition to optimizations within the renderer, SAPIEN optimizes data transfer between GPU physical simulation and rendering by directly transferring poses on the GPU through a synchronization manager, bypassing the PhysX-CPU-renderer path, as shown in Figure 4.11.

4.4.2 Ray-tracing and Bindless Rendering

In addition to the blazingly fast multi-environment rendering, SAPIEN supports photorealistic rendering through ray-tracing, which is suitable for high-fidelity sensor simulation where a smaller domain gap from the real world is preferred over rendering speed. SAPIEN's ray tracing implements the standard GGX shading [148] model and standard PBR material. SAPIEN also integrates 2 image space denoisers: NVIDIA OptiX [16] and OpenImageDenoise [3]. The unique advantage of SAPIEN ray tracing framework is its unified resource management with rasterization, which allows for resource sharing and even mixing the two techniques to synthesize a single image.

In classical rasterization pipelines, resources such as vertex buffers and textures are *bound* to each draw call and shaders only have access to the bound resources, as illustrated in Figure 4.12(a). This is feasible since we know what resources are needed to draw each triangle. However, ray tracing's nature requires shader programs to access all GPU resources at all times since we cannot predict what triangle a ray would hit. Other simulators [171, 90] and previous versions of SAPIEN choose to separate rasterization and ray tracing implementations completely. These designs prevent efficient memory sharing and pipeline mixing between the two rendering modes.

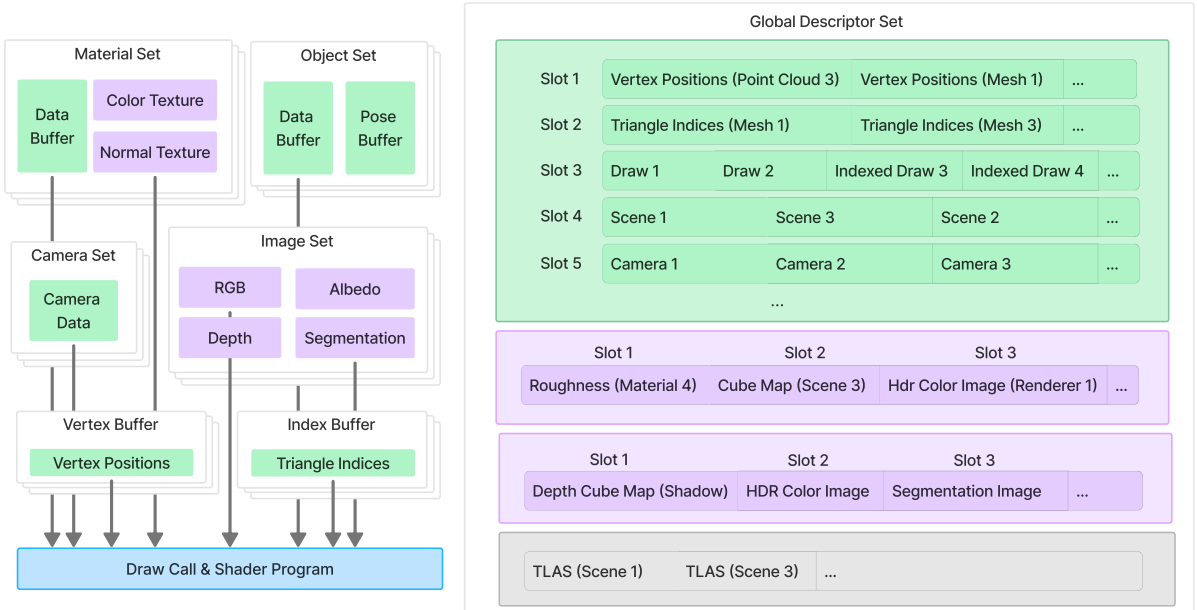
In SAPIEN, We have redesigned GPU resource management based on the ray tracing model by leveraging modern rasterization features like bindless rendering. Bindless rendering allows rasterization shaders to access all graphics resources globally, as shown in Figure 4.12(b). Instead of accessing resources from fixed bindings, shader programs in bindless rendering access resources by dynamic indices stored in GPU buffers as shown in Figure 4.12(c). Bindless rendering also enables the multi-draw feature, where instead of using the CPU to issue draw calls one by one, multi-draw allows encoding multiple draw calls into a GPU buffer and executing with a single command. This paradigm can serve as an alternative approach to minimize CPU workload during rendering.

4.4.3 Software Architecture

SAPIEN renderer's software focuses on providing a flexible and configurable system. Conceptually, SAPIEN renderer consists of 4 modules: shader manager, scene graph, resource manager, and renderer, which we summarize in figure 4.13.

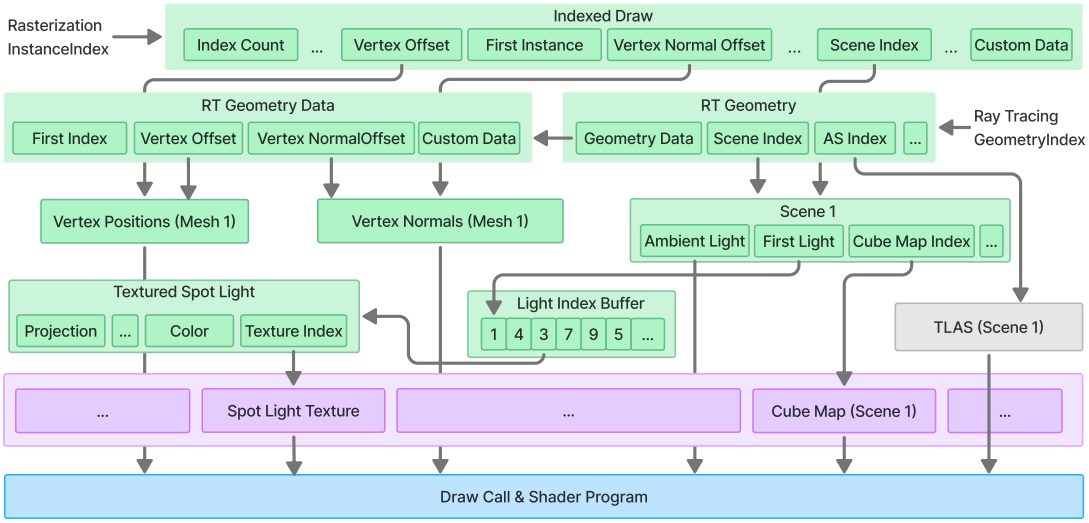
Shader Manager

Shader manager is a static configuration system for SAPIEN's rendering pipelines. We use *Shader Pack* to refer to a set of configs that describes a complete rendering procedure. A shader pack consists of a sequence of *Shader Pipelines*, each describing a rendering step. For



(a) GPU resource layout and access in classical rendering

(b) GPU resource layout in SAPIEN bindless rendering



(c) An example resource access pattern in SAPIEN bindless rendering

Figure 4.12. (a) Classical rendering pipelines group buffers and textures into many sets and shaders may only access a specified set of bound resources. (b) Bindless rendering defines a single descriptor set globally and all resources are visible to all shaders. SAPIEN currently provides global resources of 4 types, buffers, sampled images (textures), storage images, and acceleration structures. (c) SAPIEN's shader programs access buffers with dynamic indexing. For example, the shader program first accesses the scene buffer with the scene index stored in the draw buffer of rasterization or geometry buffer in ray tracing, then retrieves the environment map index from the scene buffer, and finally retrieves the environment map from the texture array.

example, a shader pack may consist of a rasterization pipeline for auxiliary information such as segmentation, a ray tracing pipeline for RGB images, a denoising pipeline for denoising ray-traced images, and a composite pipeline that combines the rasterized and ray-traced images. Rasterization and ray-tracing pipelines further decompose to *shader files* (vertex shader, fragment shader, ray miss shader, etc.) and *config variables* exposed to end users.

Scene Graph

The scene graph is a standard tree structure that stores transformations of objects, lights, and cameras. Conceptually, Nodes in the scene graph can contain GPU resource handles. For example, object nodes contain meshes with GPU vertex buffers and materials with GPU textures. These resource handles are allocated and managed by the *resource manager*.

Resource Manager

The resource manager is a singleton that manages all graphics resources such as buffers, images, and acceleration structures. The resource manager binds all resources to the global descriptor set used in bindless rendering. For images and acceleration structures, each resource created is assigned a unique ID and bound to their corresponding global arrays. The IDs are recycled when resources are deleted.

When shader files are loaded into shader packs, SAPIEN renderer parses their content through code reflection and reads the buffer variables declared in the shader files. Buffer variables with the same name are required to share a similar data type. For example, vertex position buffers must be float3 arrays and light buffers must all be 256 bytes in size. GPU data corresponding to shader variables with the same name are placed into the same Vulkan buffer with different offsets. For example, the GPU data for all cameras are grouped into an array, and users can access individual camera data with an index. We manage the allocation and deallocation of data buffers with virtual allocators provided by VulkanMemoryAllocator [40] and dynamically grow the underlying GPU memory allocation when needed.

Renderer Manager

A render manager is instantiated from a shader pack. It contains a sequence of render steps and a set of render target images. Each pipeline in the shader pack becomes a render step in the render manager, and image names and formats defined as shader pack config variables are instantiated as render target images. When recording rendering commands, the render manager takes a list of cameras and their parent scene graphs as input; rasterization and ray tracing steps then record draw calls for objects in the scene graphs. Later when executing rendering commands, output pixels are written to the render target images, which can be read by the user program.

4.4.4 Active Stereo Sensor Simulation

Active stereo depth sensors function by casting an infrared (IR) pattern onto the scene and capturing it with two IR cameras. The depth is then computed through stereo-matching between the two images. However, the depth map captured by active stereo sensors is significantly different from perfect simulation depth for several reasons. The stereo-matching process also results in missing depth values at object boundaries where the view of one camera is occluded. In unoccluded regions, active stereo sensors generate satisfactory results on mostly diffuse surfaces, but their performance degrades on glossy or transparent objects due to indirect lighting effects generated by specular reflection and refraction of the projected light pattern.

SAPIEN provides a physics-grounded stereo depth simulator that faithfully reproduces realistic noises and artifacts to mitigate the domain gap between synthetic and real-world depth images. The simulator simulates the light transport with ray tracing and performs stereo-matching with parallel SGM and SGBM algorithms. SAPIEN also supports replacing ray tracing with rasterization when indirect light is not a significant concern to gain additional simulation speed. Figure 4.14 provides a visual comparison between images produced by real and our simulated sensors.

The system is an optimized re-implementation of our previous work [168] with improved

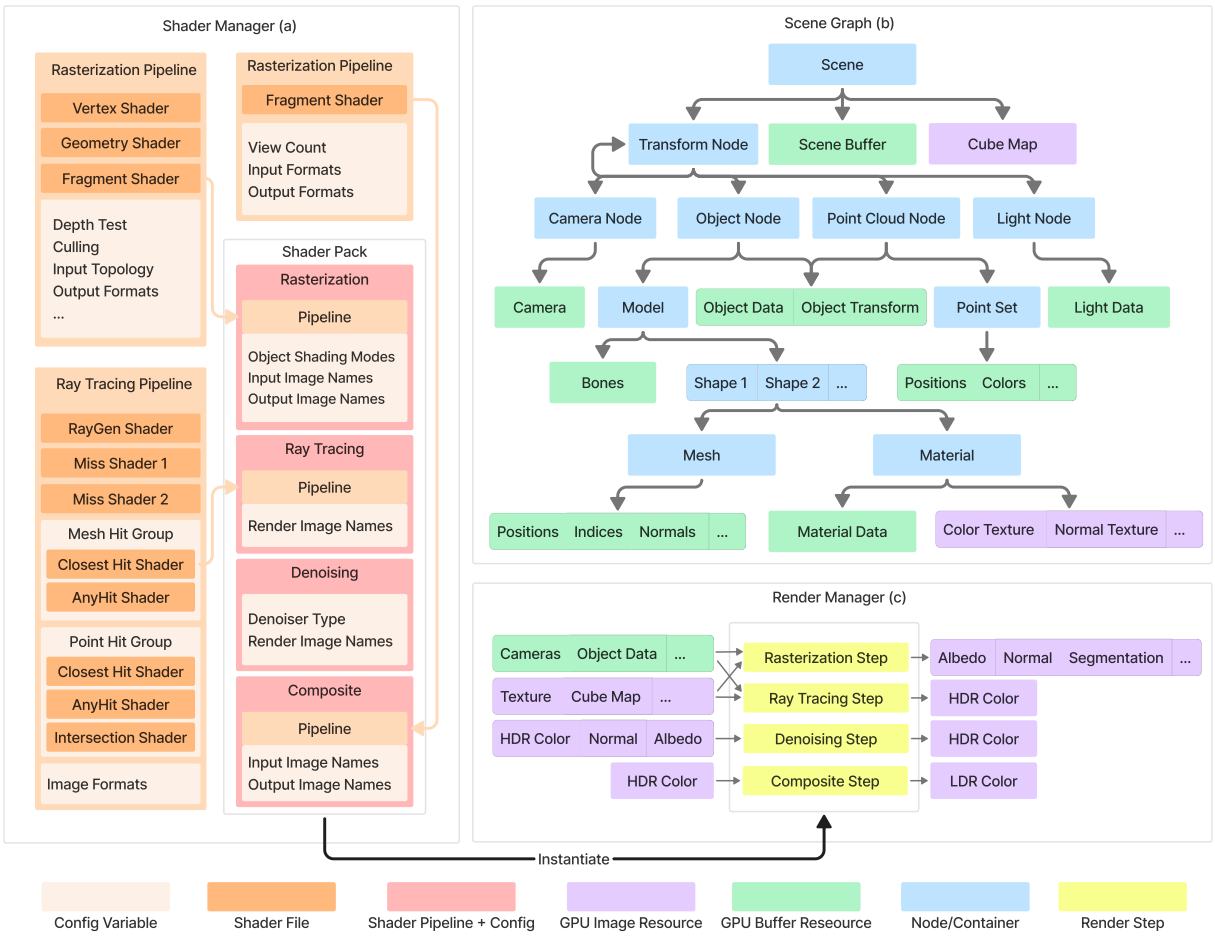


Figure 4.13. (a) Users configure shader packs with shader files and config variables. Shader packs are instantiated as shader managers used in image rendering. (b) Scenes are defined as graphics with intermediate transform nodes, objects, cameras, and lights. Scene nodes contain GPU graphics resource handles used by render managers to render images. (c) A render manager contains render target images and render steps instantiated from a shader pack. During command pre-recording, it takes a list of cameras and their corresponding scene graphs to record draw calls for each rendering step. During execution, commands stored in render steps are executed.

stereo-matching performance and rasterization speed using Vulkan multiview extension. Vision and reinforcement learning algorithms trained on the simulated sensor can transfer to the real world without fine-tuning. We refer to the original work for more technical details.

4.4.5 LiDAR Simulation

LiDAR is a different type of depth sensor based on the time-of-flight (ToF) principle. In LiDAR sensors, the sensor emits a laser toward objects in the scene and measures the distance

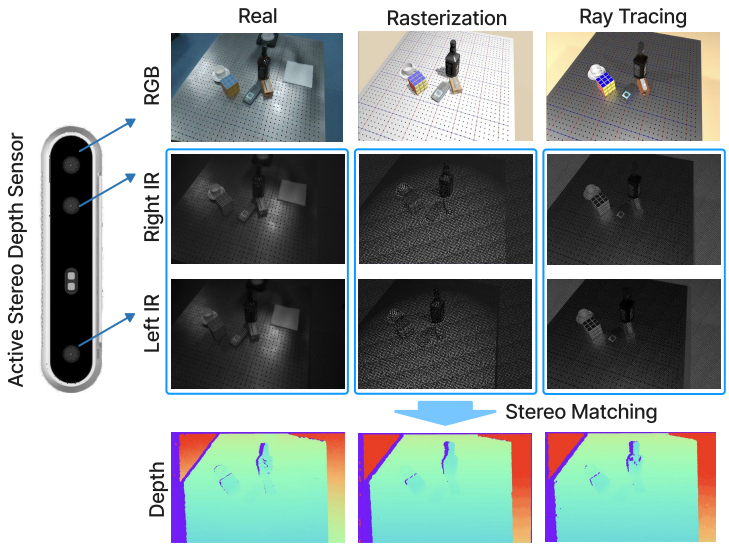


Figure 4.14. Our simulated active-stereo sensor reproduces depth errors caused by occlusions in the fast rasterization mode and artifacts caused by glossy and transparent surfaces in the ray tracing mode. Image extracted from [168].

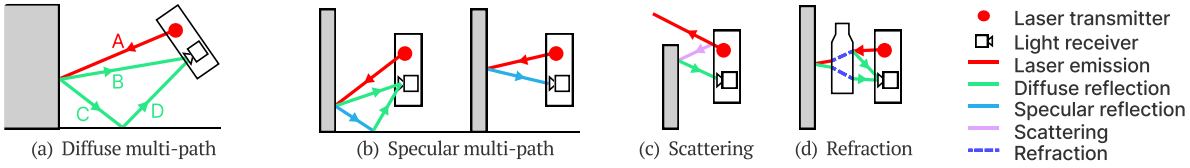


Figure 4.15. Common causes of multi-path artifacts. (a) LiDAR computes depth assuming light travels in path A-B, but some light travels in A-C-D, resulting in depth over-estimation. (b) Specular surfaces can have significant depth overestimation when viewed at an angle but accurate depth when viewed near the normal direction. (c) Light scattering causes spurious depth readings near objects when the background scene is far away. (d) Light traveling through transparent objects causes over-estimation.

from the time for reflected light to return to the receiver. LiDAR cameras such as RealSense L515 camera scan laser beams over a region to obtain a depth image. Compared to active stereo-depth sensors, ToF sensors are less susceptible to motion blur due to the short exposure time of lasers and can often provide higher quality depth values, but ToF sensors present different types of artifacts due to low signal-to-noise ratio, multiple light reception, and light scattering as detailed in [35]. The signal-to-noise ratio is a major concern when little reflected lights are detected. These regions are commonly assigned a low confidence score and filtered out. Multi-path artifacts, as illustrated in Figure 4.15, can result in inaccurate depth with high confidence.

SAPIEN provides a physics-grounded LiDAR depth simulator that simulates the multi-path artifacts with Monte-Carlo path-tracing. Our algorithm is based on the ToF path-tracing technique discussed in [162]. In fact, this system only requires minimal changes to a classical ray-tracing engine. In a word, we model the laser source as a render camera and the light receiver as a point light and record the *average light path length weighted by radiance* over multiple samples. The resulting average path length is the sensor depth output and the resulting total radiance reflects the confidence level. This approach is based on the following observations.

First, while LiDAR simulation involves a laser light source and an omnidirectional light receiver, the light path distribution and light intensity can be equivalently modeled as if the laser emitter is a camera and the receiver is a point-light source in an existing rendering framework.

Next, the ToF camera operates based on the modulation phase shift between the received light and the emitted light signal. Computing the resulting phase of multiple superposed signals requires knowledge of the modulation frequency, which varies across different sensors. We adopt a simpler approach by taking the intensity-averaged path length as an approximation. This approximation preserves several important properties of the original formulation: the simulated depth does not depend on emitted laser strength; the approximation produces the correct distance in an ideal scenario where only a single light path exists; inter-reflections result in an overestimation of depth in both the correct model and the approximation.

Finally, artifacts caused by signal-to-noise ratio can be simulated by adding noise to the total received radiance before computing the weighted average path length. In this model, depth readings at pixels with lower received radiance have greater variances, which aligns with the real observations.

Experimental Validation

We demonstrate that the point clouds generated by our LiDAR simulator have realistic error patterns in Figure 4.16. We additionally conduct quantitative comparisons to characterize the sim-to-real discrepancy. First, we use the RealSense L515 camera to capture RGB-D images

Table 4.4. Our LiDAR simulation method demonstrate smaller discrepancies between simulated depth map and real-captured ones compared to clean depth maps.

Method	↓ Chamfer ($\times 10^{-4}$)			↓ Depth MSE ($\times 10^{-4}$)		
	Banana	Cube	Can	Banana	Cube	Can
Clean	3.27	1.20	13.5	1.43	0.468	11.4
Ours	1.257	0.312	2.00	0.537	0.150	3.64

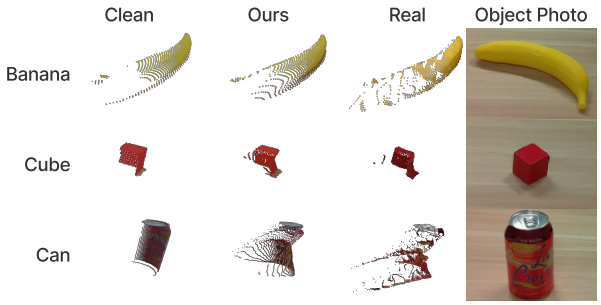


Figure 4.16. Visual comparison between real-captured and simulated point clouds. Compared to clean simulated depth, our LiDAR simulation method reproduces realistic artifacts caused by inter-reflections between objects and the environment.

of objects with known geometry, including a mostly diffuse 3D printed banana model, a red cube with specular reflection, and a beverage can with a glossy coating. The poses of the captured objects are optimized through differentiable rendering with mask loss using RGB images only. Next, we apply appropriate visual materials to the objects. The materials are not fine-tuned and we only artistically adjust the specular reflection levels by assigning different roughness and metallic parameters. Next, the objects with known poses are placed in the simulator and captured with the simulated sensor. We evaluate the sim-to-real discrepancy using 2 metrics: *point cloud Chamfer distance* between simulated and real point clouds after lifting the depth map, and *depth MSE* between the simulated and real depth maps. These metrics are computed on the point cloud for the object of interest using object masks computed from object poses. The results are summarized in Table 4.4. Despite the lack of visual material tuning, our simulator already provides significant improvements in reducing the sim-and-real discrepancies. Future works could explore material capturing to produce more realistic simulated depths.

Deployment Sim-to-Real for Visual Policies

We demonstrate that the LiDAR simulator can effectively reduce the sim-to-real domain gap, enhancing the performance of policies trained in the simulator when deployed in the real world. We replicate a real-world robot setup in SAPIEN simulator as shown in Figure 4.17. The robot is required to pick up a 3cm cube to a designated location. We perform camera extrinsic calibration by matching masks between rendered robot and real-world robot as described in [20]. We use a delta joint pose controller for the arm and an absolute joint pose controller for the gripper.

Next, we train a reinforcement learning pick-cube policy with Soft Actor-Critic (SAC), using robot joint poses and the cube pose as observation. Next, the RL policy is used as the teacher policy to train visual policies in a DAgger [118] framework. The visual policy takes the segmented object point cloud as observation and encodes it with PointNet [110]. We train the state-to-visual DAgger under 3 different settings: clean-depth point cloud where the rendered depth buffer is directly lifted, augmented point cloud where we add Gaussian noise to the depth with a standard deviation of 3mm, and simulated LiDAR point cloud. Throughout these experiments, we randomize the camera position within a 3mm cube and randomize cube size from 2.5cm to 3.5cm. The cube is initialized with random orientation in a 20cm $\times$ 20cm square. We train the state-to-visual DAgger until convergence, where all 3 policies reach 95% to 100% success rate in the simulation environment. For this experiment, the LiDAR simulator is implemented without introducing random perturbations in the signal-to-noise simulation, thereby eliminating the effects of observation randomization.

In the real world, we test sim-to-real policy deployment with 18 configurations (9 positions and 2 orientations) within a 12cm $\times$ 12cm square as shown in Figure 4.18. We obtain object point clouds from segmentation masks generated by EfficientViT-SAM [169]. As illustrated in the figure, the policy trained on simulated LiDAR point cloud without augmentation performs significantly better than clean depth augmented with Gaussian noise. The policies trained without LiDAR simulation appear to overestimate the distance between the camera and

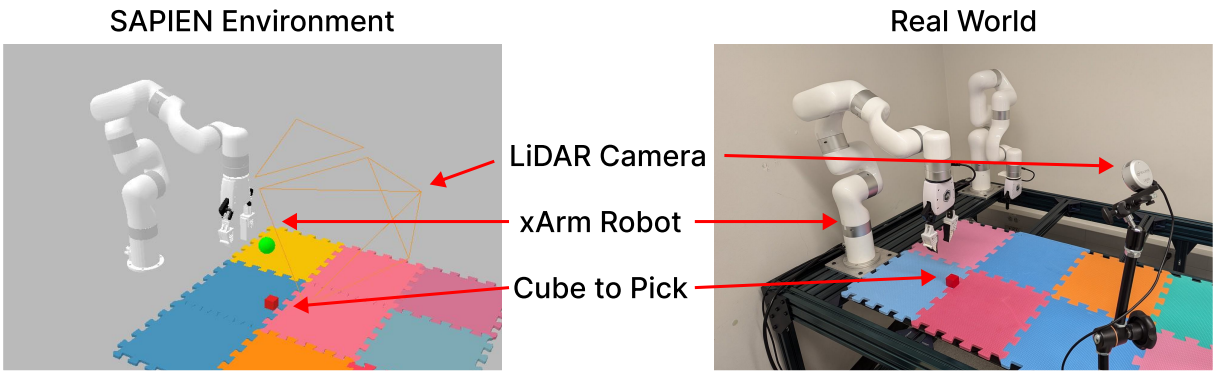


Figure 4.17. Simulation and real-world setups for LiDAR camera policy sim-to-real

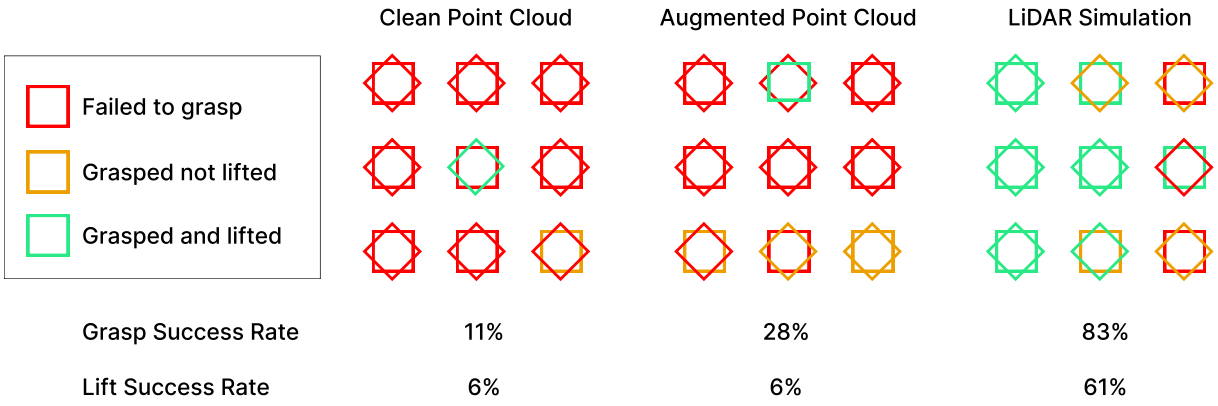


Figure 4.18. Success rate and distribution for sim-to-real grasping. Each box in the graph represents a placement of the cube (9 positions in a grid and 2 orientations of 0° and 45°). The color represents whether the robot successfully grasps and lifts the cube.

the cube. This aligns with the fact that the LiDAR camera tends to overestimate object depth due to reflections. Our LiDAR simulator simulates multiple light paths, causing this artifact to naturally appear, which results in a smaller sim-to-real domain gap.

4.5 Manipulation Environment Construction

SAPIEN has supported multiple generations of generalizable manipulation benchmark *ManiSkill*. ManiSkill v1 [93] and v2 [43] have provided more than 20 *generalizable manipulation* tasks with over 2 thousand objects and 4 million demonstration trajectories to support benchmarking reinforcement learning and learning from demonstration algorithms. The current ManiSkill v3 further expands the scope to include locomotion and navigation tasks, with a focus

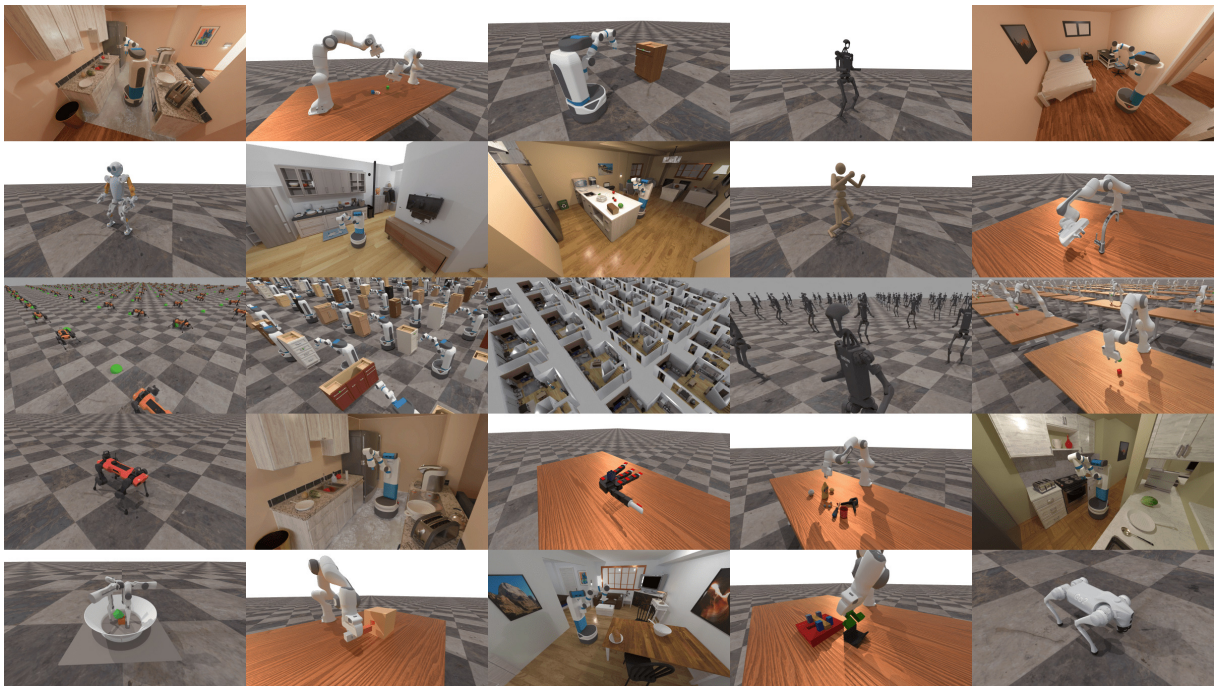


Figure 4.19. Samples of ManiSkill v3 environments rendered with ray-tracing, including scene assets sourced from AI2THOR [65] and ReplicaCAD [136].

on GPU parallelized simulation and rendering to enable high-performance learning environments. With our versatile simulation framework, ManiSkill v3 can incorporate embodied AI environments from various sources including Habitat [137] and AI2-THOR [65], and enhance them with additional features such as photorealistic rendering and sensor simulation. Figure 4.19 provides a ray-traced visualization for part of the environments supported by ManiSkill v3.

In this section, we character the performance of several typical environments in ManiSkill, highlighting the speed under state and image observations and the typical performance of RL algorithms. For the full scope of ManiSkill, including sensors, controllers, reward implementations, and tasks, we refer to ManiSkill's documentation* for details.

4.5.1 Sample Environments

ManiSkill follows the standard Gym (now Gymnasium [145]) interface to define the underlying POMDP for each environment. Currently, ManiSkill v3 has released 4 control

*https://maniskill.readthedocs.io/en/latest, accessed Aug. 15, 2024

tasks, 15 table-top manipulation tasks, 5 articulated object manipulation tasks, 2 quadruped locomotion tasks, and many new tasks are being continuously added. It also includes houses and rooms from AI2THOR and ReplicaCAD datasets to support defining tasks within more complex environments. Here we select some representative environments and characterize their problem setup, state, and visual performance, as well as metrics for baseline reinforcement learning algorithms. For all environments, the robot is controlled with the default joint delta position PD controller. All performances are reported on a desktop workstation with a single NVIDIA RTX 4090 GPU. CPU models should have minimal impact on the performance as ManiSkill adopts end-to-end GPU simulation/rendering/learning pipelines. For all RL (PPO) experiments, we use 1024 parallel environments and show the mean-variance plot over 3 successful runs with different seeds. Note that the PPO episode reward may decrease near the end due to early termination when the success condition is satisfied.

Control: CartpoleBalance

The cartpole balance environment reproduces the classic control problem of balancing a pole on a cart at the upright position by controller the cart. It is the simplest environment commonly used to verify the implementation of control and reinforcement learning algorithms. Under random actions, the environment achieves 1200K steps per second under state observations and 400k steps per second under 128×128 RGB-D observations as shown in Figure 4.20.

Quadruped Locomotion: AnymalC-Reach

AnymalC-Reach environment controls the AnymalC robot to reach a target in front of it. Under random actions, the environment achieves 100k steps per second under state observations and 20k steps per second under 128×128 RGB-D observations as shown in Figure 4.21.

Object Manipulation: PickCube

PickCube is the simplest form of tabletop manipulation task where the robot arm needs to pick up a cube with a random initial placement and lift it to a random location. The environment

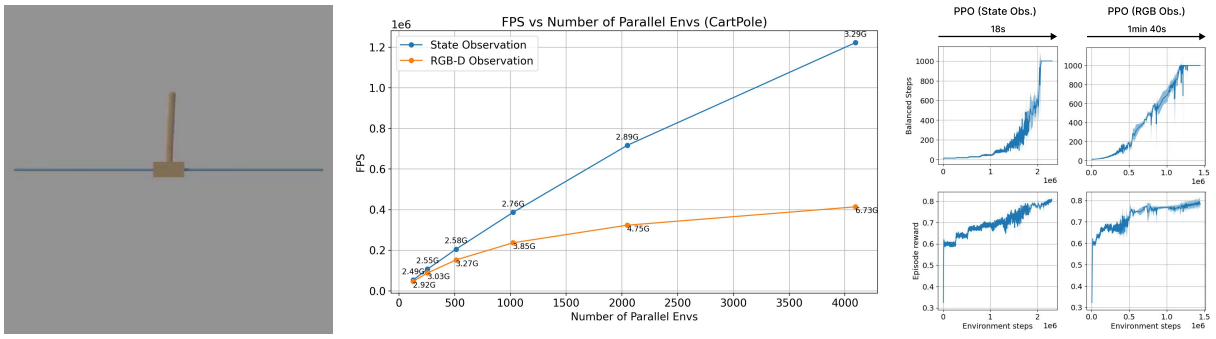


Figure 4.20. Cartpole performance metrics. Left: environment visualization. Middle: sample collection speed on state and RGB-D observation modes with different numbers of environments. Right: sample count and wall time to solve the environment with PPO in state and RGB observation modes.

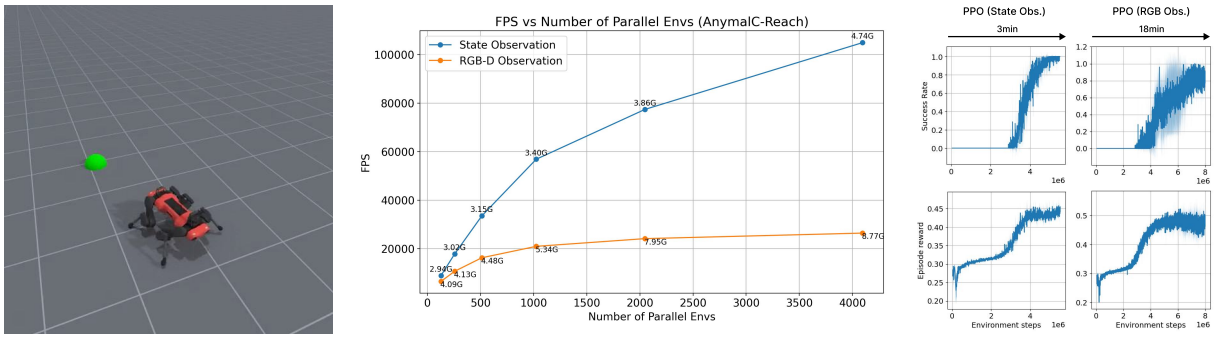


Figure 4.21. AnymalC-Reach performance metrics. Left: environment visualization. Middle: sample collection speed on state and RGB-D observation modes with different numbers of environments. Right: sample count and wall time to solve the environment with PPO in state and RGB observation modes.

achieves up to 96k steps per second under state observations and 39k under 128× RGB-D observations as shown in Figure 4.22.

Generalizable Manipulation: PickSingleYCB

PickSingleYCB environment requires the robot agent to learn to pick up a random object sampled from a list of 74 objects from the YCB dataset [14] and move it to a random location. This is one of the simplest forms of generalizable manipulation tasks. Under random actions, the environment achieves 70k steps per second under state observations and 30k steps per second under 128×128 RGB-D observations as shown in Figure 4.23. Note that our PPO implementation only solves PickSingleYCB with a 60% success rate in state mode and fails to

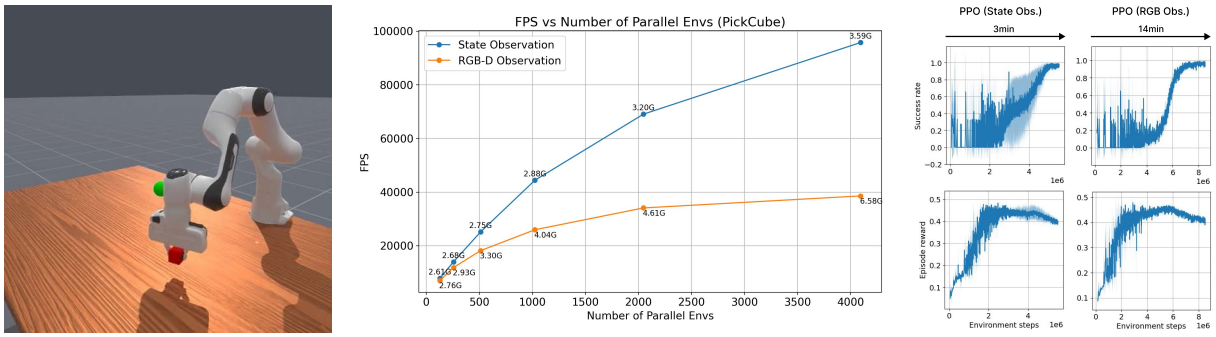


Figure 4.22. PickCube performance metrics. Left: environment visualization. Middle: sample collection speed on state and RGB-D observation modes with different numbers of environments. Right: sample count and wall time to solve the environment with PPO in state and RGB observation modes.

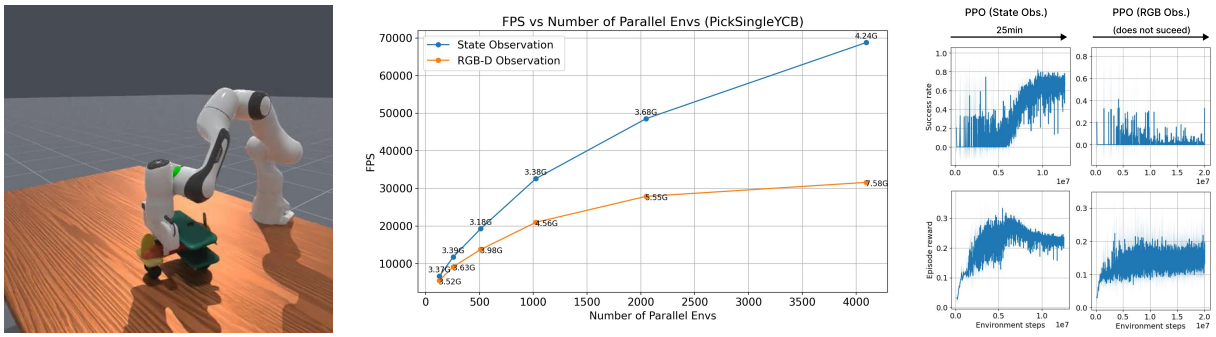


Figure 4.23. PickSingleYCB performance metrics. Left: environment visualization. Middle: sample collection speed on state and RGB-D observation modes with different numbers of environments. Right: sample count and wall time to solve the environment with PPO in state and RGB observation modes.

achieve meaningful success rates in RGB mode, and it can be readily used as a starting point for research in generalizable manipulation.

4.6 Software Package and Interfaces

4.6.1 SAPIEN Python library

Most core features of SAPIEN, such as the PhysX system and render system, are implemented in C++ and exposed to Python through Pybind11 [57]. The SAPIEN library is released as a single pip-installable Python wheel containing pre-built binaries, Python utilities, header files for creating C++ extensions as described in section 4.6.4, and Python stub files for typing

and auto-complete support in IDEs.

SAPIEN's OS and hardware compatibility are summarized in Table 4.5. SAPIEN prioritizes feature development and maintenance for Linux and supports wide ranges of hardware. Core features including CPU physical simulation and GPU rasterized rendering can also work on Windows and MacOS*.

Table 4.5. Platform compatibility for SAPIEN's features.

	PhysX Sim	Deformable Sim	Rast Rendering	Ray Tracing
CPU				
NVIDIA GPU				
AMD GPU				
Apple GPU				

4.6.2 Graphical User Interface

SAPIEN provides interactive and customizable scene visualization through the SAPIEN viewer. The viewer displays scenes through a user-controlled camera and a list of GUI windows. The GUI windows consist of GUI widgets, including text display, buttons, option menus, number value sliders, image display, etc. All GUI widgets are exposed to SAPIEN's Python API and users may create custom GUI windows by implementing a viewer *plugin*.

SAPIEN viewer provides a list of useful default plugins. The **control window** supports scene navigation with mouse and keyboard, adding auxiliary visual elements such as coordinate axes and camera indicators, and displaying images captured by cameras in the scene. The **scene window** supports browsing all entities in the scene. The **transform window** attaches a movable gizmo to the selected entity to allow dragging and setting poses; it additionally supports moving by inverse kinematics for entities with articulated link components. The **articulation window** supports browsing and editing robot poses and control parameters. The

*Linux is strongly encouraged as Windows and MacOS versions currently require case-by-case tuning.

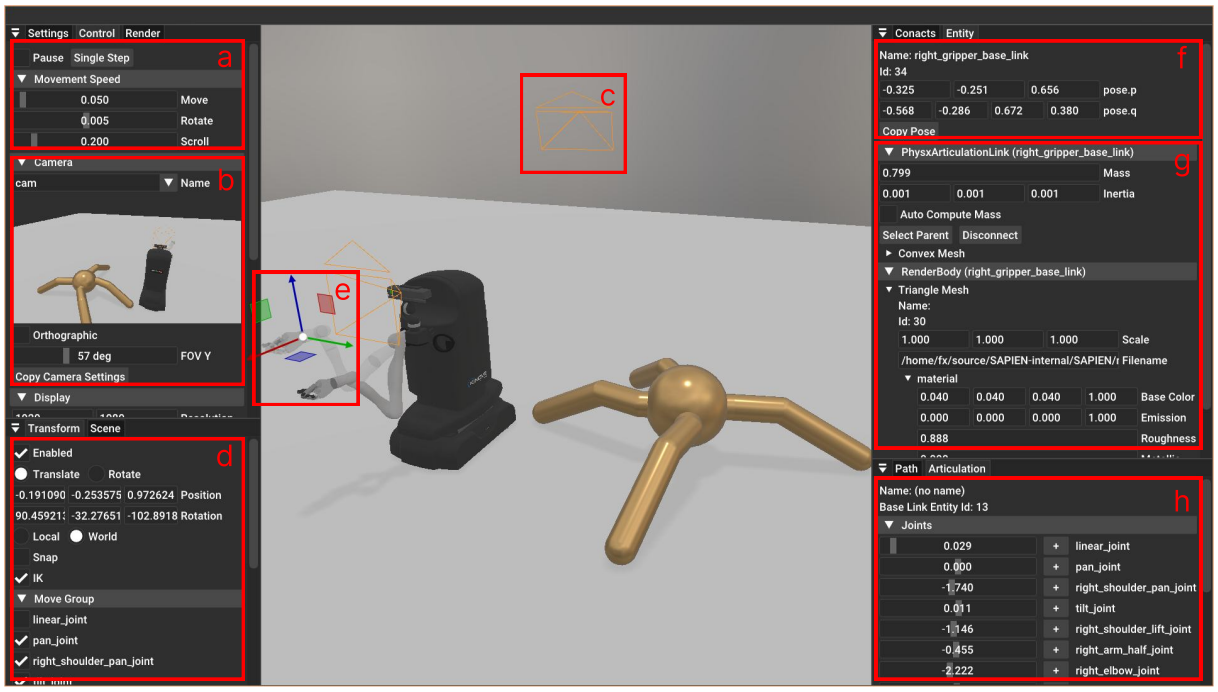


Figure 4.24. SAPIEN GUI setup example. (a) Mouse and keyboard navigation control. (b) Visualization for output of cameras in the scene. (c) visual indicator for camera placement. (d) Transform Gizmo and inverse kinematics control. (e) Draggable Gizmo and inverse kinematics visualization. (f) Entity pose information. (g) Information for attached components. (h) Articulation joint angle and controller panel. The display elements and logic for all GUI widgets are configured by Python scripts through SAPIEN Python API.

render window supports switching shaders and editing shader parameters. The **entity window** supports browsing entity pose and all components attached to an entity. For example, PhysX bodies support visualizing collision shapes; render bodies support browsing and editing visual materials. The visualization logic for each component can be registered dynamically, allowing users to implement visualization logic for custom components. A SAPIEN GUI setup example is demonstrated in Figure 4.24.

4.6.3 Virtual Reality

SAPIEN supports display on virtual reality (VR) devices via Steam's OpenVR application interface and the ALVR driver interface, which communicates with various VR devices. The VR viewer first interacts with OpenVR to obtain VR headset information, such as display

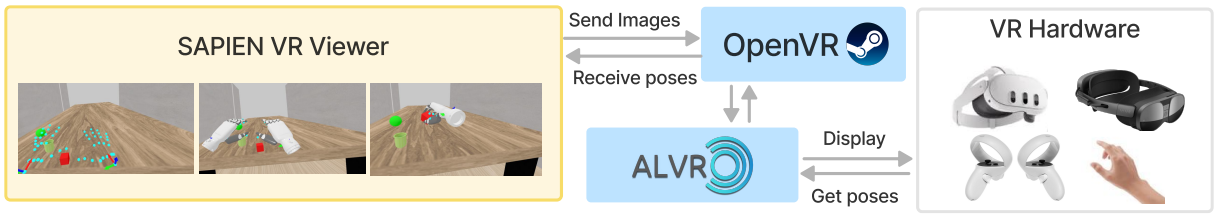


Figure 4.25. SAPIEN VR viewer sends left/right eye images to VR hardware and receives head/hand/controller poses through OpenVR and ALVR software. The received poses can be re-targeted to robot poses.

image intrinsics. Next for each frame, it reads the updated hand skeleton, controller pose, and headset pose from OpenVR and sends rendered left and right eye images. This system setup is summarized in Figure 4.25. SAPIEN VR viewer is implemented as part of SAPIEN renderer and anything renderable to a camera can be rendered to the VR display.

4.6.4 Extensions

One major advantage of SAPIEN's ECS design is the support for extensions. Users may extend SAPIEN simply by implementing new systems and components. SAPIEN supports implementing extensions with both Python and C++. For example, the MPM and FEM physical simulation systems are implemented as SAPIEN's Python extensions.

Python extensions define system classes by inheriting SAPIEN's *System* class and implement the *get_name* and *step* functions. The name of the class is used to ensure no 2 classes of the same type are added to the same scene. Custom components are created by inheriting SAPIEN's *Component* class and implementing its lifecycle methods. When Python systems and components are added to a SAPIEN scene, the scene keeps these Python objects alive through Pybind11's *shared_holder* feature.

C++ extensions are implemented by including SAPIEN's header files provided along with SAPIEN Python package. Compiled Python extension libraries are compatible with SAPIEN as long as the extension uses the same C++ ABI version and the same Pybind11 version as SAPIEN.

Acknowledgements

Chapter 4, in full, is a coauthored/unpublished technical report: “Building a General Embodied AI Simulation Framework” (Fanbo Xiang, Stone Tao, Xiaodi Yuan, Ang Li, Yulin Liu, Zhiao Huang, and Hao Su). The dissertation author was the primary investigator and author of this material.

Chapter 5

Conclusion

In this dissertation, I have presented our work on building a simulation platform for embodied AI, addressing key challenges including enhancing the performance of physical simulation and rendering, narrowing the sim-to-real domain gap, and addressing the scarcity of simulation assets. To facilitate research in articulated object manipulation, we have introduced the PartNet-Mobility dataset and the SAPIEN simulator. Additionally, we developed NeuTex, a 3D capturing method that generates 2D appearance maps. Finally, we have advanced SAPIEN into a comprehensive, high-performance simulation platform that supports a broad spectrum of research in embodied AI. Collectively, these contributions establish a solid foundation for future innovations in the creation of robotic and embodied AI systems.

Despite the progress in the field of embodied AI simulation, several open problems remain to be explored:

Accurate real-world modeling for sim-to-real.

Many sim-to-real gaps arise from the lack of simulators capable of accurately modeling underlying phenomena. Our work has applied physics-grounded modeling to optical and tactile sensors, but this approach can be extended to a broader range of robotic components, including motors, motion sensors, force/torque sensors, and sound sensors. Although some analyses for these systems exist, current simulators often treat them as black boxes, offering oversimplified models. Future research can focus on developing more accurate models and identification

methods for these systems, thereby reducing sim-to-real gaps.

Balancing simulation performance and realism for sim-to-real.

As discussed in this dissertation, the speed of physical simulation and rendering systems can vary significantly depending on whether they prioritize performance or realism. These systems serve distinct purposes, such as testing algorithms or developing policies that can be effectively transferred from simulation to the real world. The performance differences can be substantial, sometimes varying by factors of 1,000 to 10,000, depending on the modeling approach (e.g., 30,000 FPS for clean depth rendering versus 30 FPS for LiDAR sensor simulation; 1 million FPS for rigid body physics versus 100 FPS for deformable body physics). It remains unclear what level of simulation fidelity is "just enough" for sim-to-real applications, which simulation models and algorithms are most appropriate, and how to systematically develop such systems. A future embodied AI simulator that better balances performance and realism could accelerate the development of embodied AI algorithms—and, by extension, the entire field—by orders of magnitude.

Larger-scale 3D simulation asset acquisition.

While large-scale 3D model databases and object capture techniques are maturing, embodied AI simulation demands assets of greater quantity and quality. The development of visual and language foundation models depends on billions or even trillions of data samples, yet the total number of publicly available 3D models remains in the millions. Therefore, scaling up 3D asset availability requires the generation of new assets and the development of cost-effective capture methods. Additionally, simulatable assets must feature high-quality visual materials and realistic physical properties, with their representations carefully balancing fidelity and simulation performance. Another important direction for future research is how to process existing models and create new ones that are suitable for embodied AI simulation.

Bibliography

[1] Henrik Aanæs, Rasmus Ramsbøl Jensen, George Vogiatzis, Engin Tola, and Anders Bjorholm Dahl. Large-scale data for multiple-view stereopsis. *International Journal of Computer Vision*, 120(2):153–168, 2016.

[2] Panos Achlioptas, Olga Diamanti, Ioannis Mitliagkas, and Leonidas Guibas. Learning representations and generative models for 3D point clouds. In *ICML*, pages 40–49, 2018.

[3] Attila T. Áfra. Intel® Open Image Denoise, 2024. https://www.openimagedenoise.org.

[4] Peter Anderson, Qi Wu, Damien Teney, Jake Bruce, Mark Johnson, Niko Sünderhauf, Ian Reid, Stephen Gould, and Anton van den Hengel. Vision-and-Language Navigation: Interpreting visually-grounded navigation instructions in real environments. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[5] Jan Bender and Dan Koschier. Divergence-free smoothed particle hydrodynamics. In *Proceedings of the 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, SCA '15, page 147–155, New York, NY, USA, 2015. Association for Computing Machinery.

[6] Sai Bi, Nima Khademi Kalantari, and Ravi Ramamoorthi. Patch-based optimization for image-based texture mapping. *ACM Transaction on Graphics*, 36(4):106–1, 2017.

[7] Sai Bi, Zexiang Xu, Pratul Srinivasan, Ben Mildenhall, Kalyan Sunkavalli, Miloš Hašan, Yannick Hold-Geoffroy, David Kriegman, and Ravi Ramamoorthi. Neural reflectance fields for appearance acquisition. *arXiv preprint arXiv:2008.03824*, 2020.

[8] Sai Bi, Zexiang Xu, Kalyan Sunkavalli, Miloš Hašan, Yannick Hold-Geoffroy, David Kriegman, and Ravi Ramamoorthi. Deep reflectance volumes: Relightable reconstructions from multi-view photometric images. *arXiv preprint arXiv:2007.09892*, 2020.

[9] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[10] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

[11] Simon Brodeur, Ethan Perez, Ankesh Anand, Florian Golemo, Luca Celotti, Florian Strub, Jean Rouat, Hugo Larochelle, and Aaron Courville. HoME: A household multimodal environment. *arXiv preprint arXiv:1711.11017*, 2017.

[12] Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen. Unstructured lumigraph rendering. In *SIGGRAPH*, pages 425–432. ACM, 2001.

[13] Berk Calli, Arjun Singh, James Bruce, Aaron Walsman, Kurt Konolige, Siddhartha Srinivasa, Pieter Abbeel, and Aaron M Dollar. Yale-CMU-Berkeley dataset for robotic manipulation research. *The International Journal of Robotics Research*, 36(3):261–268, 2017.

[14] Berk Çalli, Aaron Walsman, Arjun Singh, Siddhartha S. Srinivasa, Pieter Abbeel, and Aaron M. Dollar. Benchmarking in manipulation research: The YCB object and model set and benchmarking protocols. *CoRR*, abs/1502.03143, 2015.

[15] Alexandre Campeau-Lecours, Hugo Lamontagne, Simon Latour, Philippe Fauteux, Véronique Maheu, François Boucher, Charles Deguire, and Louis-Joseph Caron L'Ecuyer. Kinova modular robot arms for service robotics applications. In *Rapid Automation: Concepts, Methodologies, Tools, and Applications*, pages 693–719. IGI Global, 2019.

[16] Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. page 1–12, 7 2017.

[17] Angel Chang, Angela Dai, Thomas Funkhouser, Maciej Halber, Matthias Niessner, Manolis Savva, Shuran Song, Andy Zeng, and Yinda Zhang. Matterport3D: Learning from RGB-D data in indoor environments. *International Conference on 3D Vision (3DV)*, 2017.

[18] Angel X. Chang, Thomas A. Funkhouser, Leonidas J. Guibas, Pat Hanrahan, Qi-Xing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. ShapeNet: An information-rich 3d model repository. *arXiv preprint arXiv:1512.03012*, 2015.

[19] Yevgen Chebotar, Ankur Handa, Viktor Makoviychuk, Miles Macklin, Jan Issac, Nathan Ratliff, and Dieter Fox. Closing the sim-to-real loop: Adapting simulation randomization with real world experience. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8973–8979, 2019.

[20] Linghao Chen, Yuzhe Qin, Xiaowei Zhou, and Hao Su. Easyhec: Accurate and automatic hand-eye calibration via differentiable rendering and space exploration. 2023.

[21] Rui Chen, Songfang Han, Jing Xu, and Hao Su. Point-based multi-view stereo network. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1538–1547, 2019.

[22] Weihang Chen, Jing Xu, Fanbo Xiang, Xiaodi Yuan, Hao Su, and Rui Chen. General-purpose sim2real protocol for learning contact-rich manipulation with marker-based visuotactile sensors. *IEEE Transactions on Robotics*, 40:1509–1526, 2024.

[23] Zhang Chen, Anpei Chen, Guli Zhang, Chengyuan Wang, Yu Ji, Kiriakos N. Kutulakos, and Jingyi Yu. A neural rendering framework for free-viewpoint relighting. In *CVPR*, June 2020.

[24] Zhiqin Chen and Hao Zhang. Learning implicit fields for generative shape modeling. *arXiv preprint arXiv:1812.02822*, 2018.

[25] Shuo Cheng, Zexiang Xu, Shilin Zhu, Zhuwen Li, Li Erran Li, Ravi Ramamoorthi, and Hao Su. Deep stereo using adaptive thin volume representation with uncertainty awareness. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2524–2534, 2020.

[26] Sachin Chitta, Eitan Marder-Eppstein, Wim Meeussen, Vijay Pradeep, Adolfo Tsouroukdissian, Jonathan Bohren, David Coleman, Bence Magyar, Gennaro Raiola, Mathias Lüdtke, and Enrique Perdomo. ros_control: A generic and simple control framework for ROS. *Journal of Open Source Software*, 2(20):456, 2017.

[27] Sachin Chitta, Ioan Sucan, and Steve Cousins. Moveit![ROS Topics]. *IEEE Robotics & Automation Magazine*, 19(1):18–19, 2012.

[28] Erwin Coumans and Yunfei Bai. PyBullet, a python module for physics simulation for games, robotics and machine learning. *GitHub repository*, 2016.

[29] Abhishek Das, Samyak Datta, Georgia Gkioxari, Stefan Lee, Devi Parikh, and Dhruv Batra. Embodied question answering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 2054–2063, 2018.

[30] Paul Debevec, Yizhou Yu, and George Borshukov. Efficient view-dependent image-based rendering with projective texture-mapping. In *Rendering Techniques' 98*, pages 105–116. Springer, 1998.

[31] Matt Deitke, Dustin Schwenk, Jordi Salvador, Luca Weihs, Oscar Michel, Eli VanderBilt, Ludwig Schmidt, Kiana Ehsani, Aniruddha Kembhavi, and Ali Farhadi. Objaverse: A universe of annotated 3d objects. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13142–13153, 2023.

[32] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338, 2016.

[33] Kiana Ehsani, Winson Han, Alvaro Herrasti, Eli VanderBilt, Luca Weihs, Eric Kolve, Aniruddha Kembhavi, and Roozbeh Mottaghi. Manipulathor: A framework for visual object manipulation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4497–4506, June 2021.

[34] Epic Games. Unreal engine, July 2024.

[35] Sergi Foix, Guillem Alenya, and Carme Torras. Lock-in time-of-flight (tof) cameras: A survey. *IEEE Sensors Journal*, 11(9):1917–1926, 2011.

[36] Yasutaka Furukawa and Jean Ponce. Accurate, dense, and robust multiview stereopsis. *IEEE transactions on pattern analysis and machine intelligence*, 32(8):1362–1376, 2009.

[37] Chuang Gan, Jeremy Schwartz, Seth Alter, Martin Schrimpf, James Traer, Julian De Freitas, Jonas Kubilius, Abhishek Bhandwaldar, Nick Haber, Megumi Sano, Kuno Kim, Elias Wang, Damian Mrowca, Michael Lingelbach, Aidan Curtis, Kevin T. Feigelis, Daniel M. Bear, Dan Gutfreund, David D. Cox, James J. DiCarlo, Josh H. McDermott, Joshua B. Tenenbaum, and Daniel L. K. Yamins. Threedworld: A platform for interactive multi-modal physical simulation. *CoRR*, abs/2007.04954, 2020.

[38] Xiaofeng Gao, Ran Gong, Tianmin Shu, Xu Xie, Shu Wang, and Song-Chun Zhu. VRKitchen: an interactive 3D virtual environment for task-oriented learning. *arXiv*, abs/1903.05757, 2019.

[39] Shubham Goel, Angjoo Kanazawa, and Jitendra Malik. Shape and viewpoint without keypoints. *arXiv preprint arXiv:2007.10982*, 2020.

[40] AMD GPUOpen. Vulkan Memory Allocator. https://gpuopen.com/vulkan-memory-allocator.

[41] Ned Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11):21–29, 1986.

[42] Thibault Groueix, Matthew Fisher, Vladimir G Kim, Bryan C Russell, and Mathieu Aubry. A papier-mâché approach to learning 3d surface generation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 216–224, 2018.

[43] Jiayuan Gu, Fanbo Xiang, Xuanlin Li, Zhan Ling, Xiqiang Liu, Tongzhou Mu, Yihe Tang, Stone Tao, Xinyue Wei, Yunchao Yao, Xiaodi Yuan, Pengwei Xie, Zhiao Huang, Rui Chen, and Hao Su. Maniskill2: A unified benchmark for generalizable manipulation skills.

In *International Conference on Learning Representations*, 2023.

[44] Saurabh Gupta, James Davidson, Sergey Levine, Rahul Sukthankar, and Jitendra Malik. Cognitive mapping and planning for visual navigation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2616–2625, 2017.

[45] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[46] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask R-CNN. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[47] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[48] Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. Deep blending for free-viewpoint image-based rendering. *ACM Transactions on Graphics (TOG)*, 37(6):1–15, 2018.

[49] Philipp Henzler, Niloy J Mitra, and Tobias Ritschel. Learning a neural 3d texture space from 2d exemplars. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8356–8364, 2020.

[50] Ruizhen Hu, Wenchao Li, Oliver Van Kaick, Ariel Shamir, Hao Zhang, and Hui Huang. Learning to predict part mobility from a single static snapshot. *ACM Transactions on Graphics (TOG)*, 36(6):227, 2017.

[51] Yuanming Hu, Yu Fang, Ziheng Ge, Ziyin Qu, Yixin Zhu, Andre Pradhana, and Chenfanfu Jiang. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Transactions on Graphics (TOG)*, 37(4):150, 2018.

[52] Jingwei Huang, Hao Su, and Leonidas Guibas. Robust watertight manifold surface generation method for shapenet models. *arXiv preprint arXiv:1802.01698*, 2018.

[53] Po-Han Huang, Kevin Matzen, Johannes Kopf, Narendra Ahuja, and Jia-Bin Huang. DeepMVS: Learning multi-view stereopsis. In *CVPR*, pages 2821–2830, 2018.

[54] Zhiao Huang, Yuanming Hu, Tao Du, Siyuan Zhou, Hao Su, Joshua B Tenenbaum, and Chuang Gan. Plasticinelab: A soft-body manipulation benchmark with differentiable physics. *arXiv preprint arXiv:2104.03311*, 2021.

[55] Louis Hugues and Nicolas Bredeche. Simbad: an autonomous robot simulation package for education and research. In *International Conference on Simulation of Adaptive Behavior*, pages 831–842. Springer, 2006.

[56] Seth Hutchinson, Gregory D Hager, and Peter I Corke. A tutorial on visual servo control. *IEEE transactions on robotics and automation*, 12(5):651–670, 1996.

[57] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 — seamless operability between c++11 and python, 2016. https://github.com/pybind/pybind11.

[58] Stephen James, Zicong Ma, David Rovick Arrojo, and Andrew J Davison. RLBench: The robot learning benchmark & learning environment. *arXiv preprint arXiv:1909.12271*, 2019.

[59] Mengqi Ji, Juergen Gall, Haitian Zheng, Yebin Liu, and Lu Fang. SurfaceNet: An end-to-end 3D neural network for multiview stereopsis. In *ICCV*, pages 2307–2315, 2017.

[60] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*, 2018.

[61] Angjoo Kanazawa, Shubham Tulsiani, Alexei A Efros, and Jitendra Malik. Learning category-specific mesh reconstruction from image collections. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 371–386, 2018.

[62] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Proceedings of the fourth Eurographics symposium on Geometry processing*, volume 7, 2006.

[63] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkuehler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4):1–14, 2023.

[64] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, 2004.

[65] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. AI2-THOR: An interactive 3D environment for visual AI. *arXiv:1712.05474*, 2017.

[66] Kiriakos N Kutulakos and Steven M Seitz. A theory of shape by space carving. *ICCV*, 38(3):199–218, 2000.

[67] Youngwoon Lee, Edward S Hu, Zhengyu Yang, Alex Yin, and Joseph J Lim. IKEA furniture assembly environment for long-horizon complex manipulation tasks. *arXiv preprint arXiv:1911.07246*, 2019.

[68] Nik Lever, Steve McGreal, Daniel Andersen, and Laurent Gibert. *Introduction to the Data-Oriented Technology Stack for advanced Unity developers*. Unity Technologies, 2024.

[69] Sergey Levine, Peter Pastor, Alex Krizhevsky, Julian Ibarz, and Deirdre Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*, 37(4-5):421–436, 2018.

[70] Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Chen Wang, Gabrael Levine, Michael Lingelbach, Jiankai Sun, Mona Anvari, Minjune Hwang, Manasi Sharma, Arman Aydin, Dhruva Bansal, Samuel Hunter, Kyu-Young Kim, Alan Lou, Caleb R Matthews, Ivan Villa-Renteria, Jerry Huayang Tang, Claire Tang, Fei Xia, Silvio Savarese, Hyowon Gweon, Karen Liu, Jiajun Wu, and Li Fei-Fei. BEHAVIOR-1k: A benchmark for embodied AI with 1,000 everyday activities and realistic simulation. In *6th Annual Conference on Robot Learning*, 2022.

[71] Minchen Li, Zachary Ferguson, Teseo Schneider, Timothy Langlois, Denis Zorin, Daniele Panozzo, Chenfanfu Jiang, and Danny M. Kaufman. Incremental potential contact: intersection-and inversion-free, large-deformation dynamics. *ACM Trans. Graph.*, 39(4), aug 2020.

[72] Minchen Li, Danny M Kaufman, and Chenfanfu Jiang. Codimensional incremental potential contact. *arXiv preprint arXiv:2012.04457*, 2020.

[73] Jacky Liang, Viktor Makoviychuk, Ankur Handa, Nuttapong Chentanez, Miles Macklin, and Dieter Fox. GPU-accelerated robotic simulation for distributed reinforcement learning. *arXiv preprint arXiv:1810.05762*, 2018.

[74] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[75] Xingyu Lin, Yufei Wang, Jake Olkin, and David Held. Softgym: Benchmarking deep reinforcement learning for deformable object manipulation. In *Conference on Robot Learning*, pages 432–448. PMLR, 2021.

[76] Fayao Liu, Chunhua Shen, Guosheng Lin, and Ian Reid. Learning depth from single monocular images using deep convolutional neural fields. *IEEE transactions on pattern analysis and machine intelligence*, 38(10):2024–2039, 2015.

[77] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. Neural sparse voxel fields. *Advances in Neural Information Processing Systems*, 33, 2020.

[78] Stephen Lombardi, Tomas Simon, Jason Saragih, Gabriel Schwartz, Andreas Lehrmann, and Yaser Sheikh. Neural volumes: Learning dynamic renderable volumes from images. *arXiv preprint arXiv:1906.07751*, 2019.

[79] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM siggraph computer graphics*, 21(4):163–169, 1987.

[80] Michael Lutter, Christian Ritter, and Jan Peters. Deep lagrangian networks: Using physics as model prior for deep learning. *arXiv preprint arXiv:1907.04490*, 2019.

[81] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. Isaac gym: High performance GPU based physics simulation for robot learning. In Joaquin Vanschoren and Sai-Kit Yeung, editors, *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, 2021.

[82] Khaled Mamou, E Lengyel, and Ed AK Peters. Volumetric hierarchical approximate convex decomposition. *Game Engine Gems 3*, pages 141–158, 2016.

[83] Ajay Mandlekar, Yuke Zhu, Animesh Garg, Jonathan Booher, Max Spero, Albert Tung, Julian Gao, John Emmons, Anchit Gupta, Emre Orbay, Silvio Savarese, and Li Fei-Fei. ROBOTURK: A crowdsourcing platform for robotic skill learning through imitation. In *2nd Annual Conference on Robot Learning, CoRL 2018, Zürich, Switzerland, 29-31 October 2018, Proceedings*, volume 87 of *Proceedings of Machine Learning Research*, pages 879–893. PMLR, 2018.

[84] Roberto Martín-Martín, Clemens Eppner, and Oliver Brock. The RBO dataset of articulated objects and interactions. *The International Journal of Robotics Research*, 38(9):1013–1019, 2019.

[85] Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[86] Sander Mertens. Flecs, July 2024.

[87] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3d reconstruction in function space. *arXiv preprint arXiv:1812.03828*, 2018.

[88] Johannes Meyer, Alexander Sendobry, Stefan Kohlbrecher, Uwe Klingauf, and Oskar

Von Stryk. Comprehensive simulation of quadrotor UAVs using ROS and Gazebo. In *International conference on simulation, modeling, and programming for autonomous robots*, pages 400–411. Springer, 2012.

[89] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *arXiv preprint arXiv:2003.08934*, 2020.

[90] Mayank Mittal, Calvin Yu, Qinxi Yu, Jingzhou Liu, Nikita Rudin, David Hoeller, Jia Lin Yuan, Ritvik Singh, Yunrong Guo, Hammad Mazhar, Ajay Mandlekar, Buck Babich, Gavriel State, Marco Hutter, and Animesh Garg. Orbit: A unified simulation framework for interactive robot learning environments. *IEEE Robotics and Automation Letters*, 8(6):3740–3747, 2023.

[91] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[92] Kaichun Mo, Shilin Zhu, Angel X. Chang, Li Yi, Subarna Tripathi, Leonidas J. Guibas, and Hao Su. PartNet: A large-scale benchmark for fine-grained and hierarchical part-level 3D object understanding. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[93] Tongzhou Mu, Zhan Ling, Fanbo Xiang, Derek Cathera Yang, Xuanlin Li, Stone Tao, Zhiao Huang, Zhiwei Jia, and Hao Su. Maniskill: Generalizable manipulation skill benchmark with large-scale demonstrations. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[94] Matthias Müller, Vincent Casser, Jean Lahoud, Neil Smith, and Bernard Ghanem. Sim4CV: A photo-realistic simulator for computer vision applications. *International Journal of Computer Vision*, 126(9):902–919, 2018.

[95] Adithyavairavan Murali, Tao Chen, Kalyan Vasudev Alwala, Dhiraj Gandhi, Lerrel Pinto, Saurabh Gupta, and Abhinav Gupta. PyRobot: An open-source robotics framework for research and benchmarking. *arXiv preprint arXiv:1906.08236*, 2019.

[96] Ken Museth. Nanovdb: A gpu-friendly and portable vdb data structure for real-time rendering and simulation. In *ACM SIGGRAPH 2021 Talks*, SIGGRAPH '21, New York, NY, USA, 2021. Association for Computing Machinery.

[97] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Transactions on Graphics*, 41(4):1–15, 2022.

[98] Yashraj S. Narang, Kier Storey, Iretiayo Akinola, Miles Macklin, Philipp Reist, Lukasz Wawrzyniak, Yunrong Guo, Ádám Moravánszky, Gavriel State, Michelle Lu, Ankur Handa, and Dieter Fox. Factory: Fast contact for robotic assembly. *ArXiv*, abs/2205.03532, 2022.

[99] Michael Niemeyer, Lars Mescheder, Michael Oechsle, and Andreas Geiger. Differentiable volumetric rendering: Learning implicit 3d representations without 3d supervision. In *CVPR*, pages 3504–3515, 2020.

[100] Nvidia. PhysX physics engine. https://www.geforce.com/hardware/technology/physx.

[101] Michael Oechsle, Lars Mescheder, Michael Niemeyer, Thilo Strauss, and Andreas Geiger. Texture fields: Learning texture representations in function space. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4531–4540, 2019.

[102] Michael Oechsle, Michael Niemeyer, Lars Mescheder, Thilo Strauss, and Andreas Geiger. Learning implicit surface light fields. *arXiv preprint arXiv:2003.12406*, 2020.

[103] OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving rubik's cube with a robot hand. *CoRR*, abs/1910.07113, 2019.

[104] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. OptiX: a general purpose ray tracing engine. In *ACM transactions on graphics (tog)*, volume 29, page 66. ACM, 2010.

[105] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[106] Xue Bin Peng, Erwin Coumans, Tingnan Zhang, Tsang-Wei Lee, Jie Tan, and Sergey Levine. Learning agile robotic locomotion skills by imitating animals. *arXiv preprint arXiv:2004.00784*, 2020.

[107] Julien Philip, Michaël Gharbi, Tinghui Zhou, Alexei A Efros, and George Drettakis. Multi-view relighting using a geometry-aware network. *ACM Transactions on Graphics*, 38(4):1–14, 2019.

[108] Omid Poursaeed, Matthew Fisher, Noam Aigerman, and Vladimir G Kim. Coupling explicit and implicit surface representations for generative 3d modeling. *arXiv preprint arXiv:2007.10294*, 2, 2020.

[109] Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. VirtualHome: Simulating household activities via programs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[110] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. PointNet: Deep learning on point sets for 3D classification and segmentation. In *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, volume 1, page 4, 2017.

[111] Charles R Qi, Hao Su, Matthias Nießner, Angela Dai, Mengyuan Yan, and Leonidas J Guibas. Volumetric and multi-view cnns for object classification on 3d data. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5648–5656, 2016.

[112] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. PointNet++: Deep hierarchical feature learning on point sets in a metric space. In *Advances in Neural Information Processing Systems*, pages 5099–5108, 2017.

[113] Xuebin Qin, Zichen Zhang, Chenyang Huang, Masood Dehghan, Osmar Zaiane, and Martin Jagersand. U2-net: Going deeper with nested u-structure for salient object detection. *Pattern Recognition*, 106:107404, 2020.

[114] Yuzhe Qin, Rui Chen, Hao Zhu, Meng Song, Jing Xu, and Hao Su. S4G: Amodal single-view single-shot SE (3) grasp detection in cluttered scenes. *arXiv preprint arXiv:1910.14218*, 2019.

[115] Weichao Qiu, Fangwei Zhong, Yi Zhang, Siyuan Qiao, Zihao Xiao, Tae Soo Kim, and Yizhou Wang. Unrealcv: Virtual worlds for computer vision. In *Proceedings of the 25th ACM International Conference on Multimedia*, MM '17, page 1221–1224, New York, NY, USA, 2017. Association for Computing Machinery.

[116] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[117] Eric Rohmer, Surya PN Singh, and Marc Freese. V-REP: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1321–1326. IEEE, 2013.

[118] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.

[119] Shunsuke Saito, Lingyu Wei, Liwen Hu, Koki Nagano, and Hao Li. Photorealistic facial

texture inference using deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5144–5153, 2017.

[120] Manolis Savva, Angel X. Chang, Alexey Dosovitskiy, Thomas Funkhouser, and Vladlen Koltun. MINOS: Multimodal indoor simulator for navigation in complex environments. *arXiv:1712.03931*, 2017.

[121] Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, Devi Parikh, and Dhruv Batra. Habitat: A Platform for Embodied AI Research. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[122] Connor Schenck and Dieter Fox. Spnets: Differentiable fluid dynamics for deep neural networks. *arXiv preprint arXiv:1806.06094*, 2018.

[123] Johannes Lutz Schönberger and Jan-Michael Frahm. Structure-from-motion revisited. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[124] Johannes Lutz Schönberger, Enliang Zheng, Marc Pollefeys, and Jan-Michael Frahm. Pixelwise view selection for unstructured multi-view stereo. In *European Conference on Computer Vision (ECCV)*, 2016.

[125] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[126] Brennan Shacklett, Luc Guy Rosenzweig, Zhiqiang Xie, Bidipta Sarkar, Andrew Szot, Erik Wijmans, Vladlen Koltun, Dhruv Batra, and Kayvon Fatahalian. An extensible, data-oriented architecture for high-performance, many-world simulation. *ACM Transactions on Graphics (TOG)*, 42(4):1–13, 2023.

[127] Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. Alfred: A benchmark for interpreting grounded instructions for everyday tasks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10740–10749, 2020.

[128] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics: modelling, planning and control*. Springer Science & Business Media, 2010.

[129] Vincent Sitzmann, Justus Thies, Felix Heide, Matthias Nießner, Gordon Wetzstein, and Michael Zollhofer. Deepvoxels: Learning persistent 3D feature embeddings. In *CVPR*, pages 2437–2446, 2019.

[130] Vincent Sitzmann, Michael Zollhöfer, and Gordon Wetzstein. Scene representation networks: Continuous 3d-structure-aware neural scene representations. In *Advances in*

Neural Information Processing Systems, pages 1119–1130, 2019.

[131] Breannan Smith, Fernando De Goes, and Theodore Kim. Stable neo-hookean flesh simulation. *ACM Trans. Graph.*, 37(2), mar 2018.

[132] Russell Smith. Open dynamics engine. 2005.

[133] B. Solenthaler and R. Pajarola. Predictive-corrective incompressible sph. In *ACM SIGGRAPH 2009 Papers*, SIGGRAPH '09, New York, NY, USA, 2009. Association for Computing Machinery.

[134] Shuran Song, Fisher Yu, Andy Zeng, Angel X Chang, Manolis Savva, and Thomas Funkhouser. Semantic scene completion from a single depth image. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[135] Yuhang Song, Jianyi Wang, Thomas Lukasiewicz, Zhenghua Xu, Mai Xu, Zihan Ding, and Lianlong Wu. Arena: A general evaluation platform and building toolkit for multi-agent intelligence. *arXiv preprint arXiv:1905.08085*, 2019.

[136] Julian Straub, Thomas Whelan, Lingni Ma, Yufan Chen, Erik Wijmans, Simon Green, Jakob J. Engel, Raul Mur-Artal, Carl Ren, Shobhit Verma, Anton Clarkson, Mingfei Yan, Brian Budge, Yajie Yan, Xiaqing Pan, June Yon, Yuyang Zou, Kimberly Leon, Nigel Carter, Jesus Briales, Tyler Gillingham, Elias Mueggler, Luis Pesqueira, Manolis Savva, Dhruv Batra, Hauke M. Strasdat, Renzo De Nardi, Michael Goesele, Steven Lovegrove, and Richard Newcombe. The Replica dataset: A digital replica of indoor spaces. *arXiv preprint arXiv:1906.05797*, 2019.

[137] Andrew Szot, Alexander Clegg, Eric Undersander, Erik Wijmans, Yili Zhao, John M. Turner, Noah Maestre, Mustafa Mukadam, Devendra Singh Chaplot, Oleksandr Maksymets, Aaron Gokaslan, Vladimir Vondrus, Sameer Dharur, Franziska Meier, Wojciech Galuba, Angel X. Chang, Zsolt Kira, Vladlen Koltun, Jitendra Malik, Manolis Savva, and Dhruv Batra. Habitat 2.0: Training home assistants to rearrange their habitat. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 251–266, 2021.

[138] Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. *arXiv preprint arXiv:1804.10332*, 2018.

[139] Chengzhou Tang and Ping Tan. Ba-net: Dense bundle adjustment network. *arXiv preprint arXiv:1806.04807*, 2018.

[140] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, Timothy Lillicrap, and Martin Riedmiller. DeepMind control suite. Technical report, January 2018.

[141] Russ Tedrake and the Drake Development Team. Drake: Model-based design and verification for robotics, 2019.

[142] Justus Thies, Michael Zollhöfer, and Matthias Nießner. Deferred neural rendering: Image synthesis using neural textures. *ACM Transactions on Graphics (TOG)*, 38(4):1–12, 2019.

[143] Emanuel Todorov. Convex and analytically-invertible dynamics with contacts and constraints: Theory and implementation in mujoco. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6054–6061, 2014.

[144] 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.

[145] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A standard interface for reinforcement learning environments, 2024.

[146] Yusuke Urakami, Alec Hodgkinson, Casey Carlin, Randall Leu, Luca Rigazio, and Pieter Abbeel. DoorGym: A scalable door opening environment and baseline agent. *arXiv preprint arXiv:1908.01887*, 2019.

[147] Sudheendra Vijayanarasimhan, Susanna Ricco, Cordelia Schmid, Rahul Sukthankar, and Katerina Fragkiadaki. Sfm-net: Learning of structure and motion from video. *arXiv preprint arXiv:1704.07804*, 2017.

[148] Bruce Walter, Stephen R Marschner, Hongsong Li, and Kenneth E Torrance. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, pages 195–206. Eurographics Association, 2007.

[149] Jinglu Wang, Bo Sun, and Yan Lu. Mvpnet: Multi-view point regression networks for 3D object reconstruction from a single image. *arXiv preprint arXiv:1811.09410*, 2018.

[150] Nanyang Wang, Yinda Zhang, Zhuwen Li, Yanwei Fu, Wei Liu, and Yu-Gang Jiang. Pixel2mesh: Generating 3d mesh models from single rgb images. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 52–67, 2018.

[151] Xiaogang Wang, Bin Zhou, Yahao Shi, Xiaowu Chen, Qinping Zhao, and Kai Xu. Shape2Motion: Joint analysis of motion parts and attributes from 3D shapes. In *Pro-*

ceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 8876–8884, 2019.

[152] Jiayi Weng, Min Lin, Shengyi Huang, Bo Liu, Denys Makoviichuk, Viktor Makoviychuk, Zichen Liu, Yufan Song, Ting Luo, Yukun Jiang, Zhongwen Xu, and Shuicheng Yan. Envpool: A highly parallel reinforcement learning environment execution engine. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[153] Lawrence B Wolff, Shree K Nayar, and Michael Oren. Improved diffuse reflection models for computer vision. *International Journal of Computer Vision*, 30(1):55–71, 1998.

[154] Yi Wu, Yuxin Wu, Georgia Gkioxari, and Yuandong Tian. Building generalizable agents with a realistic and rich 3D environment. *arXiv preprint arXiv:1801.02209*, 2018.

[155] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets: A deep representation for volumetric shapes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1912–1920, 2015.

[156] Fei Xia, William B Shen, Chengshu Li, Priya Kasimbeg, Micael Tchapmi, Alexander Toshev, Roberto Martín-Martín, and Silvio Savarese. Interactive Gibson: A benchmark for interactive navigation in cluttered environments. *arXiv preprint arXiv:1910.14442*, 2019.

[157] Fei Xia, Amir R Zamir, Zhiyang He, Alexander Sax, Jitendra Malik, and Silvio Savarese. Gibson Env: Real-world perception for embodied agents. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9068–9079, 2018.

[158] Fanbo Xiang, Yuzhe Qin, Kaichun Mo, Yikuan Xia, Hao Zhu, Fangchen Liu, Minghua Liu, Hanxiao Jiang, Yifu Yuan, He Wang, Li Yi, Angel X. Chang, Leonidas J. Guibas, and Hao Su. SAPIEN: A simulated part-based interactive environment. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 11094–11104. Computer Vision Foundation / IEEE, 2020.

[159] Zexiang Xu, Sai Bi, Kalyan Sunkavalli, Sunil Hadap, Hao Su, and Ravi Ramamoorthi. Deep view synthesis from sparse photometric images. *ACM Transactions on Graphics*, 38(4):76, 2019.

[160] Zexiang Xu, Kalyan Sunkavalli, Sunil Hadap, and Ravi Ramamoorthi. Deep image-based relighting from optimal sparse samples. *ACM Transactions on Graphics*, 37(4):126, 2018.

[161] Claudia Yan, Dipendra Misra, Andrew Bennnett, Aaron Walsman, Yonatan Bisk, and Yoav Artzi. CHALET: Cornell house agent learning environment. *arXiv:1801.07357*, 2018.

[162] Zhiqiang Yan, Hongyuan Wang, Xiang Liu, Qianhao Ning, and Yinxi Lu. Physics-based tof imaging simulation for space targets based on improved path tracing. *Remote Sensing*, 14(12):2868, 2022.

[163] Zihao Yan, Ruizhen Hu, Xingguang Yan, Luanmin Chen, Oliver van Kaick, Hao Zhang, and Hui Huang. RPM-Net: recurrent prediction of motion and parts from point cloud. *ACM Trans. on Graphics (Proc. SIGGRAPH Asia)*, 2019.

[164] Jianwei Yang, Zhile Ren, Mingze Xu, Xinlei Chen, David J Crandall, Devi Parikh, and Dhruv Batra. Embodied amodal recognition: Learning to move to perceive objects. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2040–2050, 2019.

[165] Yao Yao, Zixin Luo, Shiwei Li, Tian Fang, and Long Quan. Mvsnet: Depth inference for unstructured multi-view stereo. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 767–783, 2018.

[166] Lior Yariv, Yoni Kasten, Dror Moran, Meirav Galun, Matan Atzmon, Basri Ronen, and Yaron Lipman. Multiview neural surface reconstruction by disentangling geometry and appearance. *Advances in Neural Information Processing Systems*, 33, 2020.

[167] Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. TossingBot: Learning to throw arbitrary objects with residual physics. *arXiv preprint arXiv:1903.11239*, 2019.

[168] Xiaoshuai Zhang, Rui Chen, Ang Li, Fanbo Xiang, Yuzhe Qin, Jiayuan Gu, Zhan Ling, Minghua Liu, Peiyu Zeng, Songfang Han, Zhiao Huang, Tongzhou Mu, Jing Xu, and Hao Su. Close the optical sensing domain gap by physics-grounded active stereo sensor simulation. *IEEE Trans. Robotics*, 39(3):2429–2447, 2023.

[169] Zhuoyang Zhang, Han Cai, and Song Han. Efficientvit-sam: Accelerated segment anything model without performance loss. *arXiv preprint arXiv:2402.05008*, 2024.

[170] Tinghui Zhou, Richard Tucker, John Flynn, Graham Fyffe, and Noah Snavely. Stereo magnification: learning view synthesis using multiplane images. *ACM Transactions on Graphics*, 37(4):1–12, 2018.

[171] Yuke Zhu, Josiah Wong, Ajay Mandlekar, and Roberto Martín-Martín. robosuite: A modular simulation framework and benchmark for robot learning. *arXiv preprint arXiv:2009.12293*, 2020.