High-Throughput Batch Rendering for Embodied AI

LUC GUY ROSENZWEIG, Stanford University, USA
BRENNAN SHACKLETT, Stanford University, USA
WARREN XIA, Stanford University, USA
KAYVON FATAHALIAN, Stanford University, USA

Fig. 1. A batch of renders from the HSSD dataset, a popular embodied AI dataset with complex apartment-scale scenes (7.4 million triangles per scene on average). When rendering a batch of 1024 views at 128×128 pixels each, our ray tracer generates frames at an aggregate throughput of 25K frames/sec on a H100 GPU. We perform a systematic study of batch rendering performance under a range of embodied AI workloads and find that ray tracing based solutions, rather than rasterization based solutions, are a preferred rendering solution when executing on widely used datacenter-class GPUs.

In this paper we study the problem of efficiently rendering images for embodied AI training workloads, where agent training involves rendering millions to billions of independent, low-resolution frames, often with simple lighting and shading, that serve as the agent's observations of the world. To enable high-throughput training from images, we design a flexible, batch-mode rendering interface that allows state-of-the-art GPU-accelerated batch world simulators to efficiently communicate with high-performance rendering backends. Using this interface we architect and compare two high-performance renderers: one based on the GPU hardware-accelerated graphics pipeline and a second based on a GPU software implementation of ray tracing. To evaluate these renderers and encourage further research by the graphics community in this area, we build a rendering benchmark for this under-explored regime. We find that the ray tracing renderer outperforms the rasterization-based solution across the benchmark on a datacenter-class GPU, while also performing competitively in geometrically complex environments on a high-end consumer GPU. When tasked to render large batches of independent 128×128 images, the ray tracer can exceed 100,000 frames per second per GPU for simple scenes, and exceed 10,000 frames per second per GPU on geometrically complex scenes from the HSSD dataset.

CCS Concepts: • **Computing methodologies → Ray tracing**; *Rasterization.*

Authors' Contact Information: Luc Guy Rosenzweig, lrosenz@stanford.edu, Stanford University, USA; Brennan Shacklett, bps@cs.stanford.edu, Stanford University, USA; Warren Xia, waxia@stanford.edu, Stanford University, USA; Kayvon Fatahalian, kayvonf@cs.stanford.edu, Stanford University, Stanford, California, USA.

Additional Key Words and Phrases: high-performance rendering, ray casting, artificial intelligence

ACM Reference Format:
Luc Guy Rosenzweig, Brennan Shacklett, Warren Xia, and Kayvon Fatahalian. 2024. High-Throughput Batch Rendering for Embodied AI. In *Proceedings of SIGGRAPH Asia 2024 Conference Papers (SA Conference Papers '24).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3680528.3687629

1 INTRODUCTION

In modern embodied AI research, it is common to train AI agents in simulated 3D environments. In "pixels-to-actions" agent designs, the AI agent observes the state of the environment using a virtual camera (e.g., an RGB image, or a depth image) and must learn the best action to take given a rendered image. Mastering complex skills requires learning from large amounts of simulated experience drawn from a diverse set of training environments, so conducting a single training experiment can require a simulator to render billions of images across thousands of different 3D scenes.

The rendering scenario described above presents a unique high-performance rendering challenge. Instead of rendering a stream of high-quality images of a single scene at real-time rates for human visual perception, a renderer for embodied AI training is tasked to render many independent streams of low-fidelity images (low resolution, simple lighting and shading), depicting many different environments, at the *highest aggregate throughput possible*—ideally several orders of magnitude faster than real-time rates.

Recently, the desire to reduce the cost of training agents has led to the design of high-performance, batch environment simulators that execute many environment instances concurrently on the GPU. However, due to the perceived high cost of rendering, most of these highly optimized systems do not attempt to render image observations for agents. Therefore, they do not support pixels-to-actions agents, only agents that operate directly on world state (e.g., the positions of scene objects).

In this paper, we conduct the first systematic study of the question: how do we design a high-performance rendering architecture for the needs of pixels-to-actions AI training experiments? We explore how to architect high-performance interfaces between a renderer and existing high-performance, GPU-accelerated, batch world simulation engines, and investigate how rasterization and ray tracing-based renderers perform under this workload. Specifically we make the following contributions:

(1) We design and implement a GPU-driven interface for batch rendering that allows batched, GPU-accelerated world simulators to communicate environment state (object positions etc) efficiently through GPU memory to high-throughput batch renderer backends that render many independent environments and many views of each environment.
(2) We implement two reference batch renderers using this interface, one rasterization-based renderer patterned after prior work, and a batch ray tracer. We demonstrate these renderers deliver state-of-the-art performance in this rendering domain.
(3) We create a rendering benchmark representative of modern embodied AI workloads by assembling a representative set of embodied AI learning environments (and corresponding rendering configurations) from recent research efforts.
(4) We report key findings from experiments studying the performance of batch rendering systems running on high-end consumer and server-class GPUs. Notably, we find that on a high-end H100 GPU (a flagship platform for modern AI), a software ray tracer can significantly outperform a prior state-of-the-art system based on hardware-accelerated rasterization. Given these results and the rendering characteristics of modern embodied AI research, we argue that contrary to existing practice, practitioners looking to build high-performance rendering systems for this workload should consider architecting around ray tracing solutions going forward.

2 PRELIMINARIES: BATCH SIMULATION FOR EFFICIENT TRAINING

2.1 The Agent Training Loop

To establish basic system requirements, we first describe key components of an end-to-end AI agent training system. Agents are trained in simulated 3D worlds, often referred to as *learning environments* in AI literature. In an environment, an agent observes the state of the world, and takes actions, determined by its action *policy* – a deep neural network (DNN) – as it attempts to complete a task. An agent's actions modify the environment's state. For example, in an object manipulation task where the goal is for an agent to move an object to a particular location, an agent's actions might involve deciding where to move its grasping arm next. In this example, an environment simulator would execute basic collision detection operations with 3D geometry and update the positions and velocities of the agent's arm and scene objects accordingly.

The system records the results of environment simulation over a short period of time (often called an *episode* or *rollout*), and uses this information as feedback to a learning algorithm, which modifies the agent's action policy to improve its task performance. In practice, it is common to collect experience from many independent episodes using the same policy before performing a policy update step (potential inter-episode parallelism). In each episode the environment may have different initial conditions or even be a completely different 3D scene. Then, the updated policy is used to attempt the task again in new episodes. Experience is collected again, and the results are used to make new policy updates. To learn complex skills, this process must repeat for many episodes. For example, it is common for modern reinforcement learning based learning methods [Sutton and Barto 2018] to require hundreds of millions of episodes (billions of simulated time steps) to converge to capable policies [Baker et al. 2020; Handa et al. 2023; OpenAI et al. 2019b; Peng et al. 2022; Wijmans et al. 2020].

The process described above features three potentially costly components: *Policy inference/update*, which involves DNN inference and training. *Environment simulation*, which may require complex logic like 3D physics calculations and time stepping the environment. And *generating agent observations*, which involves producing a representation of the environment's current state for the agent's policy. When the agent directly observes the environment (e.g., it receives 3D coordinates of all scene objects as input), the cost of generating agent observations is low. However, when training pixels-to-actions agents, which make decisions based on what a virtual sensor "sees", generating observations involves rendering an image of the 3D scene from the perspective of the sensor. This can be costly, and reducing this cost is the focus of our work.

2.2 High Throughput via Batched Execution

The need for efficient rendering is now acute because both policy inference/training and environment simulation have undergone substantial optimization in recent years. DNN design improvements and specialized tensor-processing accelerator hardware have steadily reduced the cost of policy evaluation and training. Similarly, the cost of environment simulation has been reduced by orders of magnitude using *batch simulator* designs that execute thousands of independent environments concurrently on the GPU using coherent parallel execution [Dalton et al. 2020; Freeman et al. 2021; Gulino et al. 2023; Makoviychuk et al. 2021; Rutherford et al. 2023; Shacklett et al. 2023]. By leveraging the massive compute and memory bandwidth of modern GPUs, state-of-the-art batch simulators are capable of executing environments at millions of frames per second, even in 3D environments driven by rigid body physics simulations.

To our knowledge, the only prior work focused on improving the performance of rendering specifically for AI training workloads is the BPS3D system [Shacklett et al. 2021], which applied batched simulation ideas to the renderer itself. BPS3D simultaneously rasterizes thousands of independent scenes by packing rendering commands into a single GPU command buffer, which reduced CPU-GPU communication overhead and provided the GPU graphics pipeline with large amounts of rendering work despite individual scene renderings being too low of resolution to effectively utilize a modern GPU. In our work, we extend the batch-rasterizer design of [Shacklett et al. 2021] to interface efficiently with high-performance, GPU-accelerated batch environment simulators, and reapply the idea of batch rendering in the context of a ray tracing-based renderer.

3 EMBODIED AI RENDERING WORKLOADS

High-performance batch rendering for embodied AI poses significantly different requirements than those placed on typical real-time rendering systems. At a high level, a batch renderer must integrate with learning environments driven by high-throughput GPU batch simulators to render billions of frames of agent observations during training. In this section we enumerate the resulting requirements for the high-performance graphics community.

Many environment, many view rendering. Batch simulation training systems simultaneously execute thousands of independent episodes, in environments that may involve different 3D scenes. Generating agent observations for each environment may also involve rendering the same scene from multiple viewpoints. For example, each agent might feature multiple virtual sensors (e.g, a forward and backward camera), or a single environment might involve multiple agents cooperating to perform a task. Therefore, after each step of batch environment simulation, the rendering system is presented with thousands of independent rendering jobs to complete.

Throughput-maximized rendering. The goal of a rendering system for agent training is to maximize aggregate frame throughput under the load of potentially billions of rendering jobs, which arrive in batches of thousands of independent rendering requests. Common rendering concerns such as reducing the latency of rendering a single frame, or ensuring predictable frame rate to avoid visible stutter, are not requirements of a high-performance batch renderer for agent training. Instead we seek a rendering architecture that can receive environment updates from modern high-performance, GPU-accelerated batch simulators and render these dynamic environments at tens of thousands, to millions, of frames per second.

Wide range of geometric complexity. Learning environments can range in geometric complexity from scenes with many simple geometric primitives (cubes, spheres, etc.) to detailed 3D scans of real-world environments. For example, scenes in our experiments range from seven thousand to over seven million triangles. As a result, renderers must accommodate a wide range of geometry complexity.

Low-resolution output. Many pixels-to-actions AI training experiments operate using low-resolution images: approximately 64×64 to 256×256 in size [OpenAI et al. 2019a; Shacklett et al. 2021; Szot et al. 2021]. Researchers have observed that many challenging tasks can still be solved at lower resolutions, and researchers seeking to generate large amounts of training experience reduce the cost of image generation and expensive DNN processing (policy evaluation) on these images by reducing image size. As a result, rendered triangles have small screen-space area (often sub-pixel).

Low-visual-fidelity rendering. Many state-of-the-art training results depend only on visibility information such as depth, or the semantic object ID of what is visible in each pixel [Deitke et al. 2020; Wijmans et al. 2020; Yadav et al. 2023]. Even when RGB rendering is enabled, the vast majority of agent training experiments do not depend on high-fidelity shading and lighting (e.g., simple Blinn-Phong shading, disabled shadows) [Berges et al. 2023; Petrenko et al. 2021; Savva et al. 2019]. While more realistic appearance simulation is likely to be desirable in fields like robotics where the ultimate goal is to use simulation to train agents that operate successfully in the real world, the exact requirements and benefits of realistic rendering remain an open research question. For example, techniques such as domain randomization [Sadeghi and Levine 2017; Tobin et al. 2017] of low-fidelity images are used as alternatives to high-fidelity rendering to close the sim-to-real gap. *In summary, high-quality shading and lighting is not a requirement for a rendering system to be useful for a wide range of agent training workloads.*

Dynamic scene generation and modification. While realistic appearances are not a requirement for modern training workloads, environment diversity is [Deitke et al. 2022]. Each training episode might take place in a different 3D scene, or in a scene that is procedurally generated on the fly at the start of an episode. These different scenes may feature different scene assets, and different numbers and locations of objects. When simulating thousands of parallel episodes, potentially of different lengths, the rendering structures for new environment instances must be constructed (and cleaned up) frequently and efficiently during training.

4 BATCH RENDERER IMPLEMENTATION

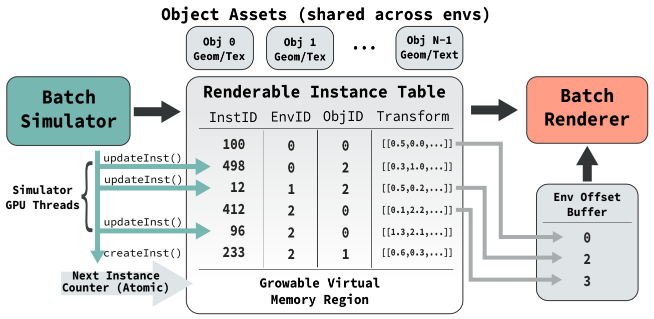
In this section, we describe BLINK, a high-throughput interface for communicating environment state updates (object movement, creation and deletion, etc) from GPU batch simulators to high-performance batch renderer backends. Using BLINK, we then present the implementation of two batch renderers, BRAST and BTRACE, that meet the feature and performance requirements in Section 3.

4.1 GPU Batch Simulator – Batch Renderer Interface

A key design challenge in any high-performance graphics system involves architecting how the simulator communicates environment state, and state changes, to the renderer [He et al. 2017]. Unfortunately, the vast majority of graphics systems make a fundamental assumption: environment simulation logic is controlled by the host CPU with environment state residing in CPU memory. As a result, their renderers only provide a CPU-side interface to communicate environment state updates via standard graphics hardware APIs.

CPU-side renderer interfaces are inefficient for state-of-the-art GPU-accelerated batch simulators because these systems execute environment simulation logic on the GPU. These simulators must employ expensive synchronization and data read back operations to copy environment state from the GPU to the CPU, and then pass it through CPU-side APIs that ultimately copy it *back to GPU memory* for rendering. Moreover, it can be challenging for CPU code to efficiently parallelize the large number of renderer state updates needed for thousands of batch-simulated environments.

To address these inefficiencies we present BLINK (Batch renderer/simulator Linkage), a high-performance interface between GPU batch environment simulation and batch rendering that allows GPU simulation logic to communicate environment state updates to the renderer in parallel and entirely through GPU memory.



InstID	EnvID	ObjID	Transform
100	0	0	[[0.5,0.0,...]]
498	0	2	[[0.3,1.0,...]]
12	1	2	[[0.5,0.2,...]]
412	2	0	[[0.1,2.2,...]]
96	2	0	[[1.3,2.1,...]]
233	2	1	[[0.6,0.3,...]]

Fig. 2. BLINK allows a GPU batch simulator to communicate environment state changes to the renderer in a high-performance and memory-efficient manner. BLINK stores all instance data, across all environments, in a column-major table in GPU memory ("Renderable Instance Table") that permits parallel allocation, deallocation, and update by the batch simulator. After the batch simulator has completed all updates, BLINK sorts this table by environment id to enable fast data access by the renderer. (Not shown: a similar design is used to store view information.) Object assets are stored separately from the instance table, allowing mesh and texture data to be allocated once and shared across all environments.

4.1.1 *Batch Renderer Interface Concepts.*

BLINK extends standard instanced rendering APIs to the multi-environment batch setting by providing simulators an interface for allocating, deallocating, and updating the following entities:

1. *Environments.* From the perspective of a renderer, an environment serves as a namespace to encapsulate a set of object instances and views.
2. *Objects* correspond to 3D assets (made up of triangle meshes and optionally textures). Objects are referenced by a unique, global ID and are not specific to any one environment.
3. Object *instances*, which are associated with a specific environment, and maintain position and rotation transforms that define their position in that environment. A single object may be instanced many times, across many environments.
4. *Views* define camera parameters and are also associated with a specific environment. Each environment can have multiple views. The batch renderer is responsible for rendering output frames for all views across all environments. Currently, views share the same image parameters (height, width, etc.)

The challenge when implementing this interface on the GPU is how to allow efficient parallel creation, deletion, and updates of instance and view data while ultimately aggregating data in a format suitable for efficient batch rendering.

4.1.2 *Implementation.*

BLINK addresses this challenge by adopting recent low-footprint, parallelism-friendly GPU memory management strategies that favor throughput over latency [Shacklett et al. 2023] and storing data for all instances (and all views) across all environments in large, contiguous GPU memory buffers that enable efficient access and lookup by renderer backends.

Fig. 2 illustrates how instance data for all environments is organized into a single table and updated by the batch simulator. At the end of each environment simulation step, the batch simulator updates instance information for all instances in all environments by using the renderer API to map instance ids to an offset in the table, and then updates the table's contents (e.g., position and rotation transforms) in parallel using GPU threads. Parallel instance allocation is implemented by atomically incrementing a pointer to append new instances to the table. Parallel deletion is implemented by marking a row in a table as invalid (the environment ID is set to -1). Similar schemes are used to modify view information.

After the simulator has updated instance and view data, BLINK prepares the data for efficient rendering. The parallel allocation scheme described above can lead to instances (or views) from different environments being interleaved in the table, presenting a problem for renderer backends that require efficient access to data for *each* environment. To solve this problem, BLINK globally synchronizes across all environments before rendering begins, and sorts the rows of the instance and view data tables by environment ID [Adinets and Merrill 2022]. After the sort, control is transferred to the renderer backend along with pointers to each environment's instance and view data, which is now continuous in memory (Fig. 2, "Env Offset Buffer").

BLINK's synchronization and sort step is pragmatic in a batch setting since maximizing overall throughput is prioritized over minimizing per-view rendering latency. In the subsequent sections we discuss how the sorted table layout aids common rendering tasks such as GPU command buffer construction and BVH construction. The sort also serves to bring deleted rows (with environment ID -1) to the end of the table, so memory reclamation efficiently implemented at the end of each simulation step by truncating the table. Row allocation is simplified by mapping the table to a large region of virtual memory that can be backed by physical pages as necessary [Shacklett et al. 2023].

As illustrated in Fig. 2, object data is allocated once in the system, and reused across all instances from all environments.

4.1.3 *BLINK Simulator Integration.*

BLINK defines the binary format of the aforementioned in-memory table structure, but it does not stipulate how a batch simulator creates and populates the structure. A batch simulator seeking to communicate with a BLINK renderer need only export contiguous GPU memory buffers containing instance and view data tagged by environment ID in the struct-of-arrays table format described above. Given this data (along with optional additional renderer-backend data provided as additional table columns), BLINK will sort the data internally and then invoke the batch renderer backend. Providing renderer state updates in the correct tabular format is the lowest-level option for using BLINK. For convenience, we also provide a CUDA C++ API that GPU batch simulators can access to create and populate buffers. This API provides methods for atomically assigning table rows and virtual memory management. We use the C++ wrapper to implement integrations with two GPU batch simulators as described in Section 5.

4.2 BRAST: A Batch Rasterizer

Our batch rasterizer (BRAST) is an evolution of BPS3D, the Vulkan-based batch rasterizer that achieved orders of magnitude speedups over prior non-batch renderers when rendering 3D scans of real-world scenes [Shacklett et al. 2021]. The original BPS3D implementation has two major limitations in the context of this work: 1) BPS3D only provides a CPU simulator interface, with no efficient pathway for high-performance GPU batch simulators, and 2) multi-agent training experiments are not possible due to an internal limitation of one view per environment. We address both these limitations by replacing BPS3D's front-end simulator interface and CPU-side state management with BLINK, and then reimplement BPS3D's GPU-driven culling and batch rendering strategy using BLINK structures.

Specifically, BRAST accesses the instance and view data buffers exposed by BLINK directly in a GPU-compute pass that performs frustum culling in parallel across instances. As a result, BRAST avoids per-instance CPU-GPU transfer costs incurred by BPS3D. Additionally, since BLINK exposes per-environment instance and view data for fast GPU lookup, BRAST, unlike BPS3D, supports supports multi-view rendering. BPS3D parallelizes frustum culling using one warp (32 threads) per view. Warp threads cooperatively loop over instances in the view's environment, yielding high execution coherence even when different environments in a batch have different numbers of views (e.g. varying numbers of agents per environment).

4.3 BTRACE: A Batch Raytracer

The combination of low output image resolution and the relatively high geometric complexity of many learning environments yields a rendering workload featuring many small-area triangles (often subpixel) that is known to map poorly to modern GPU-accelerated graphics pipelines [Fatahalian et al. 2010; Karis 2021]. Acting on this observation, we create a second batch renderer implementation based on ray tracing (BTRACE).

BTRACE is a software-based batch ray tracer written in CUDA using persistent threads [Aila and Laine 2009]. Its design follows many best practices in high-performance GPU ray tracing, such as a two-level BVH to handle dynamic scenes [Parker et al. 2010], and the use of compressed, wide BVHs to reduce memory bandwidth and dependent memory accesses [Ylitie et al. 2017]. The bottom level BVH for each object is constructed off-line, and both construction costs and the resulting GPU memory storage are amortized across all environments in the batch. The top-level BVH for each environment holds object instances, and is constructed from scratch each frame using the LBVH algorithm [Lauterbach et al. 2009].

BLINK enables two important optimizations for the LBVH implementation. A standard LBVH build begins by spatially sorting instances by morton code. Rather than introducing the overhead of a separate parallel sort within BTRACE, we extend the instance sort already performed by BLINK to support an optional secondary sort key that BTRACE uses to pre-sort instances by morton code within each environment before the LBVH build begins. (We hypothesize that BLINK's support for multiple sort keys is generally useful; e.g., a rasterizer might pre-sort environment instances by depth.) After the sort, BTRACE leverages the contiguous data layout of per-environment instances to efficiently construct all per-environment LBVHs in parallel via a single compute kernel dispatch. Specifically, BTRACE combines the instance data across all environments into a single highly parallel LBVH build [Karras 2012] that keeps the BVH tree structure for each environment separate using bounds checks against the environment offsets provided by BLINK.

One key aspect of our solution is that it *does not* utilize hardware-accelerated ray tracing APIs (e.g., Vulkan, Optix). Our initial prototypes found these platforms demonstrate poor throughput under the load of constructing many small top-level BVHs. In addition, flagship datacenter AI GPUs, a primary hardware target for our work, currently lack ray tracing acceleration hardware.

5 EVALUATION

Using our BRAST and BTRACE batch-renderers, we perform a systematic study of the efficiency of batch rendering performance across several popular embodied AI datasets and two GPU device classes commonly used for embodied AI research. We use this study to evaluate our design decisions, understand the current performance of our state-of-the-art implementations, and also to identify opportunities for performance optimization going forward.

5.1 Experimental Setup

5.1.1 Benchmark Learning Environments. We focus our study on dynamic, interactable environments created by active embodied AI research communities interested in pixels-to-actions training. We select four example datasets that span a gamut of scene complexity and AI task diversity.

- **Madrona Hide and Seek** is a multi-agent learning environment built in Madrona [Shacklett et al. 2023], a GPU-accelerated batch simulation framework. Prior versions of Hide and Seek provide internal engine state to agents. We modify the environment to support vision-based agents by attaching a RGB camera to each agent. The environment contains five agents, each represented as spheres (several hundred triangles each). All other objects are simple geometric primitives with less than ten faces per object. Objects are either solid colored or use simple textures. This is the simplest environment in the benchmark from a rendering perspective and is typical of modern environments used to learn multi-agent teamwork [Baker et al. 2020].
- **MuJoCo MJX Barkour** is a quadruped robot simulated in MuJoCo MJX (a GPU-accelerated batch physics engine) [Limited 2024]. This environment includes one depth-only camera viewing the Barkour robot from top-down as it moves on a plane. Rendered geometry is taken from MJX collision assets for the robot (30 instances, ~230K total triangles). This environment is a proxy for typical of robotics research: small-scale environments with a few detailed geometries.
- **ProcTHOR** is a collection of 64 procedurally generated home interiors randomly drawn from the thousands of scenes in the ProcTHOR dataset [Deitke et al. 2022]. We load these environments into the Madrona engine and place a single RGB camera in a random location in each environment to represent the agent. The camera follows a random path through each environment. On average these scenes feature ~244K triangles and ~70 object instances.
- **HSSD** is a collection of 64 detailed, human-authored building interiors (homes and apartments) randomly drawn from the Habitat Synthetic Scenes Dataset [Khanna et al. 2023]. Like

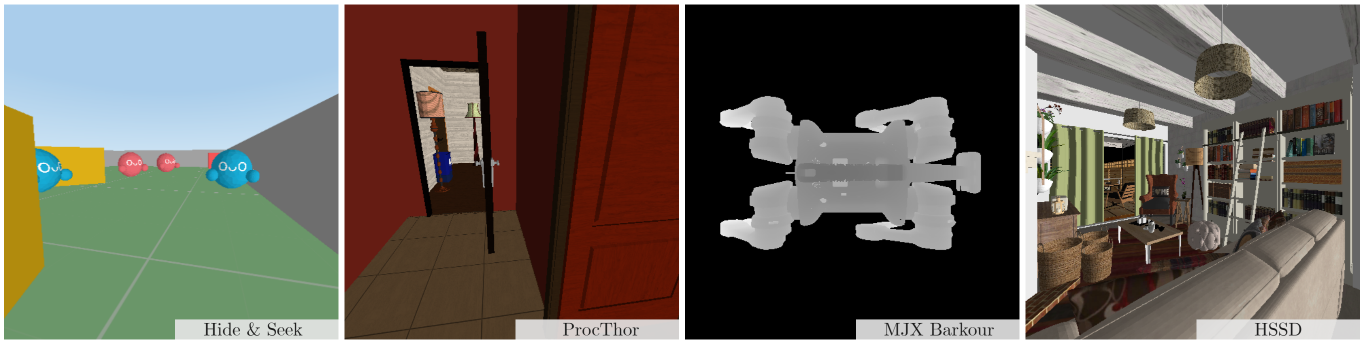


Fig. 3. Example agent views from environments from the four benchmarks. Scene complexity ranges from simple geometric primitives (Hide and Seek) to millions of triangles per environment in apartment-scale scenes (HSSD). All environments generate RGB images, except MJX Barkour, which only requires depth rendering. Hide and Seek is a multi-agent environment where the same environment is rendered multiple times to produce observations from the perspective of each agent. All other benchmarks involve single-view environments.

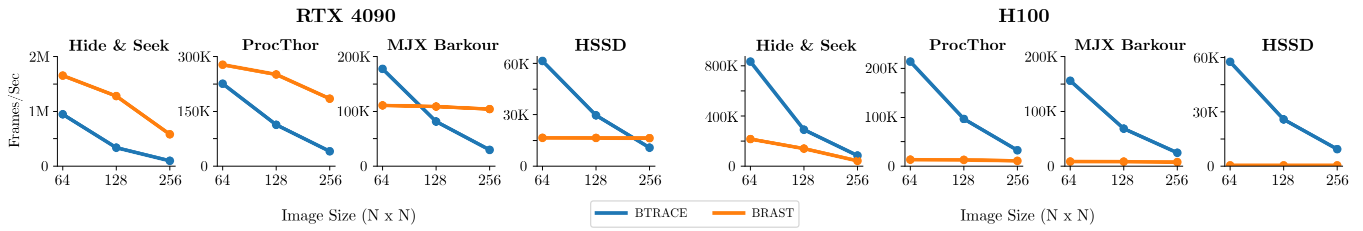


Fig. 4. Throughput of the BRAST and BTRACE batch renderers across all environments on RTX 4090 and H100 GPUs. BTRACE significantly outperforms BRAST in all configurations on the H100, since the H100 is a datacenter class GPU with lower amounts of rasterization hardware. Even on the RTX 4090, the prototype BTRACE renderer is competitive with BRAST on the more geometrically complex HSSD and MJX Barkour scenes.

ProcTHOR, HSSD is loaded into the Madrona engine and benchmarked with one RGB camera. With high resolution textures and on average 7.6M triangles (~330 instances) per environment, HSSD exhibits significantly greater visual detail than prior synthetic embodied AI datasets. It serves as a proxy for increasingly complex future embodied AI environments.

Figure 3 shows example renderings from all benchmarks.

5.1.2 Benchmarking Setup. We measure renderer throughput on two important classes of GPUs for AI training: a NVIDIA GeForce RTX 4090 (a high-end consumer-grade GPU, often used for smaller scale AI experiments) and a NVIDIA H100 Tensor Core GPU (a high-end datacenter GPU for AI, used for large-scale agent training). The RTX 4090 benchmark machine uses a Intel Core i9 13900K CPU. The H100 is located in a Google Compute Engine A3 instance.

The resolution of images provided to agents varies significantly in the literature depending on the task being learned. To evaluate a range of different performance regimes, we measure throughput for each dataset at resolutions typical for agent training: 64×64, 128×128 and 256×256 pixels per view [Shacklett et al. 2021; Stooke et al. 2021; Szot et al. 2021; Wijmans et al. 2020].

Following prior work we fix batch size in all experiments to 1024 environments, a practical size for training vision-based agents [Shacklett et al. 2021]. Larger batch sizes would increase renderer throughput, but potentially hit GPU memory limitations during training. Hide and Seek is procedurally generated, so all environments in a batch have unique, randomly generated layouts. When rendering HSSD and ProcTHOR, we sample 16 unique scenes from the dataset and generate a batch of 1024 environments using these scenes (different agent initial conditions). This setup mimics end-to-end training, where due to memory limitations, scene assets will typically be swapped on and off the GPU during training. Environments where agents receive RGB observations use Phong lighting.

5.2 Renderer Throughput Comparison

Figure 4 plots the throughput (in aggregate frames/sec) of BRAST and BTRACE, and illustrates that the relative performance between the two renderers depends heavily on output image resolution and the GPU hardware being used. On the RTX 4090 (a consumer gaming GPU), high fixed-function rasterizer throughput results in BRAST outperforming BTRACE when screen-space triangle area is larger (higher image resolutions, lower-geometric complexity scenes). On the higher geometric complexity scenes (MJX Barkour and HSSD), BRAST throughput is bound by the performance of triangle-processing hardware. Profiling tools on the RTX 4090 show that the Primitive Distributor bottlenecks the performance in higher geometry environments. Since the H100 includes less fixed-function triangle processing hardware than the RTX 4090 (it is designed for AI workloads, not 3D graphics performance), the H100 BTRACE achieves significantly higher throughput than BRAST on nearly all benchmark conditions. In the extreme case, even when rendering HSSD at 256×256 resolution, BTRACE is still 22× faster than BRAST.

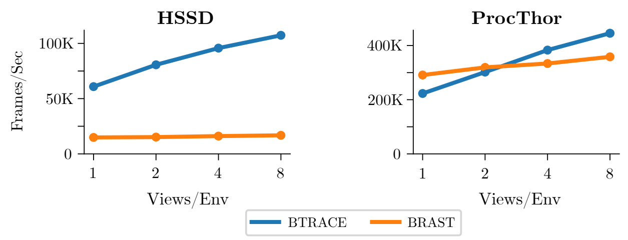


Fig. 5. BRAST and BTRACE throughput on the RTX 4090 as the number of views per environment increases (64×64 pixels per view). The performance of BTRACE improves relative to BRAST with increasing view count because BTRACE amortizes BVH build costs across views. This efficiency results in BTRACE outperforming BRAST on ProcTHOR when rendering more than two views per environment.

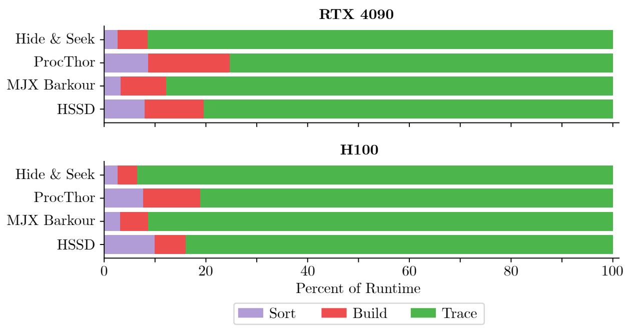


Fig. 6. A breakdown of the performance of BTRACE while rendering a batch of 1024 environments at 128×128 pixels. Both the radix sort used by BLINK and the LBVH build consume a small fraction of per-batch costs relative to ray tracing time. This suggests future work to improve BVH build quality would likely improve performance.

Table 1. Rendering throughput at 64×64 resolution of BRAST compared against the prior state-of-the-art batch renderer, BPS3D [Shacklett et al. 2021]. BRAST leverages the high-throughput simulator interface provided by BLINK to eliminate CPU-side setup costs and outperforms BPS3D, especially when rendering low-cost environments where CPU costs can limit overall renderer throughput.

Renderer	Hide and Seek (FPS)	MJX Barkour (FPS)
BRAST	1056K	111K
BPS3D	589K	76K

While more advanced GPU-driven culling [Karis 2021] and geometric level-of-detail solutions could improve BRAST's performance by reducing the number of triangles the hardware rasterizer must process, these efforts would require significant renderer engineering, and still may not close the performance gap. *These results suggest that to achieve high end-to-end system throughput for embodied AI training, either architects of datacenter AI GPUs should consider provisioning more fixed-function triangle processing hardware in their designs, or batch renderers for embodied AI should be built on ray tracing based methods.* We also note that BTRACE currently features only basic GPU ray tracing optimizations. We believe further performance engineering would not only substantially increase the BTRACE-BRAST performance gap on the H100, but likely close most of the BTRACE-BRAST performance gap on the RTX 4090.

A second trend is that on all benchmarks but HSSD, overall batch renderer throughput is high. For example, when rendering at 128×128 on the RTX 4090, aggregate BTRACE performance is nearly 100K FPS. At these rates, significant performance engineering of DNN policy inference/training is required to keep up with the batch renderer. This suggests that for the first time, performance headroom unexpectedly exists to explore how more sophisticated rendering techniques (e.g., more advanced lighting simulation, or simulating artifacts of real sensors) might benefit training [Chattopadhyay et al. 2021]. BTRACE is a significantly more versatile renderer for exploring the potential of more advanced rendering effects.

5.3 Multi-Agent Environment Scaling

Multi-agent training workloads are an exciting area of embodied AI, and an interesting domain for BTRACE, because rendering costs can be amortized across multiple views of the same environment. Fig. 5 compares the performance of BRAST and BTRACE on the RTX 4090 in a hypothetical multi-robot task that scales the views per environment from 1 to 8 in ProcTHOR and HSSD. Increasing the views per environment provides opportunities for BTRACE to amortize the cost of building an environment's top-level BVH across all views, unlike BRAST, which processes geometry for each view independently. As a result, the performance of BTRACE improves relative to BRAST on both datasets as view count increases. In the case of ProcTHOR, BTRACE begins to outperform BRAST after two views.

5.4 BTRACE Performance Analysis

BTRACE represents a prototype-quality implementation of a software batch ray tracer on the GPU. To understand current bottlenecks and inform future optimization, Fig. 6 illustrates the fraction of total time spent in stages of BTRACE execution when rendering a batch of 1024 environments at 128×128 pixels. Note that BLINK only consumes a small fraction of total time each batch (purple bar) to perform sorting of instances by environment and Morton code. Even for HSSD, where each environment has hundreds of instances, less than 10% of total time per batch is spent sorting.

The dominant cost in all configurations is tracing rays. This suggests that replacing the fast LBVH build step (which occupies less than 10% of runtime) with a slower, but higher quality BVH build algorithm could reduce overall render time by making ray traversal more efficient. In a batch renderer, top-down BVH builds could be parallelized across environments, an axis of parallelism that does not exist in real-time GPU rendering workloads. In addition, many lower-level optimizations are possible to lower ray traversal costs, such as further compression of BVH nodes and triangle indices, and leveraging on-chip shared memory for ray traversal optimizations [Vaidyanathan et al. 2019; Ylitie et al. 2017].

5.5 BLINK Renderer Interface Performance

Finally, we directly evaluate the benefits of BLINK on renderer throughput. Table 1 ablates our design by comparing the performance of BRAST (which uses BLINK to interface with the batch simulator) against BPS3D for the Hide and Seek and MJX Barkour datasets. We constrain Hide and Seek to have one agent per environment since BPS3D does not support multi-view environments.

Recall BPS3D was a prior state-of-the-art batch rasterizer that uses a CPU-side interface to communicate with the batch simulator. This requires copying camera data and instance transforms from the GPU batch simulator back to the CPU each frame. BPS3D compacts the instance data in CPU memory and uploads it back to the GPU. The resulting copy and synchronization overheads limit the throughput achieved by BPS3D. Overall BRAST outperforms BPS3D by 1.8× on Hide and Seek and 1.5× on MJX Barkour. BLINK also allows BRAST and BTRACE to support multiple views per environment, and the above results show this additional flexibility does not come at a performance cost. BLINK allows the two GPU-driven subsystems to interoperate efficiently without CPU intervention, and provides benefits both in terms of improving throughput but also enabling new multi-agent training workloads.

6 DISCUSSION

Training embodied computer vision based agents in simulated environments is an emerging area that stands to be a major consumer of future computer graphics systems. Our work contributes new rendering systems optimized for this workload. Perhaps more importantly, we hope that by characterizing the field's rendering requirements, and identifying a set of benchmark scenes for evaluation of future systems, our work leads to further progress in this interesting area of high-performance graphics.

Our investigations, although preliminary, suggest interesting opportunities going forward. For example, although nearly every prior system used for fast rendering in this domain is based on rasterization, the widespread use of datacenter GPUs for training, as well as the growing geometric complexity of training scenes, suggests that software ray tracing-based renderers that leverage the full programmable capability of these GPUs, are likely a more performant, and pragmatic design going forward.

Our results also suggest that a well-architected batch renderer can achieve exceptionally high throughput on scenes in frequent use in embodied AI research today. This suggests new possibilities for AI reseachers to consider how learning algorithms could best take advantage of much greater amounts of collected experience, or how to utilize more advanced rendering algorithms to help train agents in simulation that exhibit new capabilities, such as more advanced skills, or more robust transfer of the skills learned in simulation to AI agents operating in the real world.

ACKNOWLEDGMENTS

We thank Erwin Coumans, Erik Frey, Joshua Schnarr and Chris Lentini for valuable conversations, insights and hardware. Support for this project was provided by Meta, Activision, Andreessen Horowitz, and the US ARL under No. W911NF-21-2-0251. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

REFERENCES

Andy Adinets and Duane Merrill. 2022. Onesweep: a faster least significant digit radix sort for gpus. *arXiv preprint arXiv:2206.01784* (2022).

Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009* (New Orleans, Louisiana) *(HPG '09)*. Association for Computing Machinery, New York, NY, USA, 145–149. https://doi.org/10.1145/1572769.1572792

Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. 2020. Emergent Tool Use From Multi-Agent Autocurricula. In *International Conference on Learning Representations*. https://openreview.net/forum?id=SkxpxJBKwS

Vincent-Pierre Berges, Andrew Szot, Devendra Singh Chaplot, Aaron Gokaslan, Roozbeh Mottaghi, Dhruv Batra, and Eric Undersander. 2023. Galactic: Scaling End-to-End Reinforcement Learning for Rearrangement at 100k Steps-Per-Second. In *CVPR*.

Prithvijit Chattopadhyay, Judy Hoffman, Roozbeh Mottaghi, and Ani Kembhavi. 2021. RobustNav: Towards Benchmarking Robustness in Embodied Navigation. *International Conference in Computer Vision (ICCV)* (2021).

Steven Dalton, Iuri Frosio, and Michael Garland. 2020. Accelerating Reinforcement Learning through GPU Atari Emulation. In *Advances in Neural Information Processing Systems*, Vol. 33.

Matt Deitke, Winson Han, Alvaro Herrasti, Aniruddha Kembhavi, Eric Kolve, Roozbeh Mottaghi, Jordi Salvador, Dustin Schwenk, Eli VanderBilt, Matthew Wallingford, et al. 2020. Robothor: An open simulation-to-real embodied ai platform. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 3164–3174.

Matt Deitke, Eli VanderBilt, Alvaro Herrasti, Luca Weihs, Kiana Ehsani, Jordi Salvador, Winson Han, Eric Kolve, Aniruddha Kembhavi, and Roozbeh Mottaghi. 2022. ProcTHOR: Large-Scale Embodied AI Using Procedural Generation. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.). https://openreview.net/forum?id=4-bV1bi74M

Kayvon Fatahalian, Solomon Boulos, James Hegarty, Kurt Akeley, William R. Mark, Henry Moreton, and Pat Hanrahan. 2010. Reducing shading on GPUs using quad-fragment merging. *ACM Trans. Graph.* 29, 4, Article 67 (jul 2010), 8 pages. https://doi.org/10.1145/1778765.1778804

C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. 2021. Brax - A Differentiable Physics Engine for Large Scale Rigid Body Simulation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, Vol. 1.

Cole Gulino, Justin Fu, Wenjie Luo, George Tucker, Eli Bronstein, Yiren Lu, Jean Harb, Xinlei Pan, Yan Wang, Xiangyu Chen, John D. Co-Reyes, Rishabh Agarwal, Rebecca Roelofs, Yao Lu, Nico Montali, Paul Mougin, Zoey Yang, Brandyn White, Aleksandra Faust, Rowan McAllister, Dragomir Anguelov, and Benjamin Sapp. 2023. Waymax: An Accelerated, Data-Driven Simulator for Large-Scale Autonomous Driving Research. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*.

Ankur Handa, Arthur Allshire, Viktor Makoviychuk, Aleksei Petrenko, Ritvik Singh, Jingzhou Liu, Denys Makoviichuk, Karl Van Wyk, Alexander Zhurkevich, Balakumar Sundaralingam, et al. 2023. Dextreme: Transfer of agile in-hand manipulation from simulation to reality. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 5977–5984.

Yong He, Tim Foley, Teguh Hofstee, Haomin Long, and Kayvon Fatahalian. 2017. Shader components: modular and high performance shader development. *ACM Trans. Graph.* 36, 4, Article 100 (jul 2017), 11 pages. https://doi.org/10.1145/3072959.3073648

Brian Karis. 2021. Nanite: A Deep Dive. In *SIGGRAPH 2021 Courses: Advances in Real-Time Rendering in Games*. https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf.

Tero Karras. 2012. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics* (Paris, France) *(EGGH-HPG'12)*. Eurographics Association, Goslar, DEU, 33–37.

Mukul Khanna, Yongsen Mao, Hanxiao Jiang, Sanjay Haresh, Brennan Shacklett, Dhruv Batra, Alexander Clegg, Eric Undersander, Angel X. Chang, and Manolis Savva. 2023. Habitat Synthetic Scenes Dataset (HSSD-200): An Analysis of 3D Scene Scale and Realism Tradeoffs for ObjectGoal Navigation. arXiv:2306.11290 [cs.CV]

C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. 2009. Fast BVH Construction on GPUs. *Computer Graphics Forum* (2009). https://doi.org/10.1111/j.1467-8659.2009.01377.x

DeepMind Technologies Limited. 2024. MuJoCo XLA Documentation. https://mujoco.readthedocs.io/en/stable/mjx.html.

Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. 2021. Isaac Gym: High Performance GPU Based Physics Simulation

For Robot Learning. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, Vol. 1.

OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. 2019a. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113* (2019).

OpenAI, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. 2019b. Dota 2 with Large Scale Deep Reinforcement Learning. arXiv:1912.06680

Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. 2010. Optix: a general purpose ray tracing engine. *Acm transactions on graphics (tog)* 29, 4 (2010), 1–13.

Xue Bin Peng, Yunrong Guo, Lina Halper, Sergey Levine, and Sanja Fidler. 2022. ASE: Large-scale Reusable Adversarial Skill Embeddings for Physically Simulated Characters. *ACM Trans. Graph.* 41, 4, Article 94 (July 2022).

Aleksei Petrenko, Erik Wijmans, Brennan Shacklett, and Vladlen Koltun. 2021. Megaverse: Simulating Embodied Agents at One Million Experiences per Second. In *ICML*.

Alexander Rutherford, Benjamin Ellis, Matteo Gallici, Jonathan Cook, Andrei Lupu, Gardar Ingvarsson, Timon Willi, Akbir Khan, Christian Schroeder de Witt, Alexandra Souly, Saptarashmi Bandyopadhyay, Mikayel Samvelyan, Minqi Jiang, Robert Tjarko Lange, Shimon Whiteson, Bruno Lacerda, Nick Hawes, Tim Rocktaschel, Chris Lu, and Jakob Nicolaus Foerster. 2023. JaxMARL: Multi-Agent RL Environments in JAX. *arXiv preprint arXiv:2311.10090* (2023).

Fereshteh Sadeghi and Sergey Levine. 2017. CAD2RL: Real Single-Image Flight Without a Single Real Image. In *Robotics: Science and Systems XIII, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, July 12-16, 2017*, Nancy M. Amato, Siddhartha S. Srinivasa, Nora Ayanian, and Scott Kuindersma (Eds.). https://doi.org/10.15607/RSS.2017.XIII.034

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. 2019. Habitat: A Platform for Embodied AI Research. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*.

Brennan Shacklett, Luc Guy Rosenzweig, Zhiqiang Xie, Bidipta Sarkar, Andrew Szot, Erik Wijmans, Vladlen Koltun, Dhruv Batra, and Kayvon Fatahalian. 2023. An Extensible, Data-Oriented Architecture for High-Performance, Many-World Simulation. *ACM Trans. Graph.* 42, 4, Article 90 (jul 2023), 13 pages.

Brennan Shacklett, Erik Wijmans, Aleksei Petrenko, Manolis Savva, Dhruv Batra, Vladlen Koltun, and Kayvon Fatahalian. 2021. Large Batch Simulation for Deep Reinforcement Learning. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Adam Stooke, Anuj Mahajan, Catarina Barros, Charlie Deck, Jakob Bauer, Jakub Sygnowski, Maja Trebacz, Max Jaderberg, Michael Mathieu, Nat McAleese, Nathalie Bradley-Schmieg, Nathaniel Wong, Nicolas Porcel, Roberta Raileanu, Steph Hughes-Fitt, Valentin Dalibard, and Wojciech Marian Czarnecki. 2021. Open-Ended Learning Leads to Generally Capable Agents. arXiv:2107.12808 [cs.LG]

Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (second ed.). The MIT Press. http://incompleteideas.net/book/the-book-2nd.html

Andrew Szot, Alexander Clegg, Eric Undersander, Erik Wijmans, Yili Zhao, John Turner, Noah Maestre, Mustafa Mukadam, Devendra Singh Chaplot, Oleksandr Maksymets, et al. 2021. Habitat 2.0: Training home assistants to rearrange their habitat. *Advances in neural information processing systems* 34 (2021), 251–266.

Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. 2017. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 23–30. https://doi.org/10.1109/IROS.2017.8202133

Karthik Vaidyanathan, Sven Woop, and Carsten Benthin. 2019. Wide BVH Traversal with a Short Stack. In *High-Performance Graphics - Short Papers*, Markus Steinberger and Tim Foley (Eds.). The Eurographics Association. https://doi.org/10.2312/hpg.20191190

Erik Wijmans, Abhishek Kadian, Ari Morcos, Stefan Lee, Irfan Essa, Devi Parikh, Manolis Savva, and Dhruv Batra. 2020. DD-PPO: Learning Near-Perfect PointGoal Navigators from 2.5 Billion Frames. In *International Conference on Learning Representations (ICLR)*.

Karmesh Yadav, Ram Ramrakhya, Santhosh Kumar Ramakrishnan, Theo Gervet, John Turner, Aaron Gokaslan, Noah Maestre, Angel Xuan Chang, Dhruv Batra, Manolis Savva, et al. 2023. Habitat-matterport 3d semantics dataset. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4927–4936.

Henri Ylitie, Tero Karras, and Samuli Laine. 2017. Efficient incoherent ray traversal on GPUs through compressed wide BVHs. In *Proceedings of High Performance Graphics* (Los Angeles, California) *(HPG '17)*. Association for Computing Machinery, New York, NY, USA, Article 4, 13 pages. https://doi.org/10.1145/3105762.3105773

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009