Lecture 10:

High-Throughput World Simulation for Agent Training

Visual Computing Systems
Stanford CS348K, Spring 2025

Today

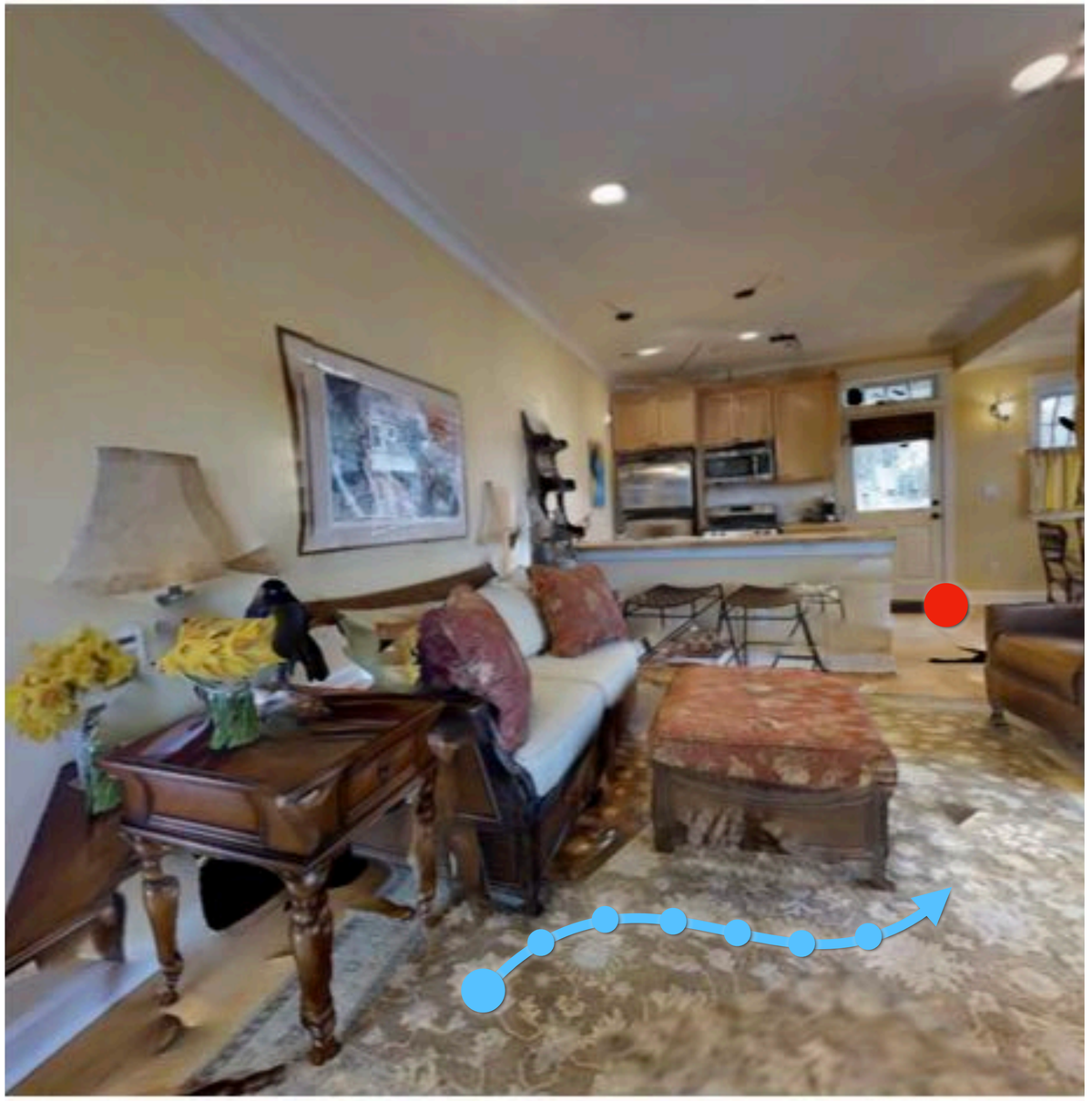
- Slides on the landscape of high-performance world simulation efforts designed for improving the efficiency of training embodied AI agents
- Discussion of the Madrona system
- Discussion: do we even need to simulate from a traditional world model at all?
  - Can't generative AI just make our training data?
  - Setup for tonight's reading: Genie

Background:
Why learning via trial and error requires a lot of simulated experience (reinforcement learning example)

RL in 30 seconds

Model Inference

environment observation e.g. RGB image → π_θ → agent action



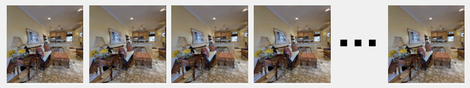
RL in 30 seconds

Model Inference

environment observation e.g. RGB image → π_θ → agent action

Model Training

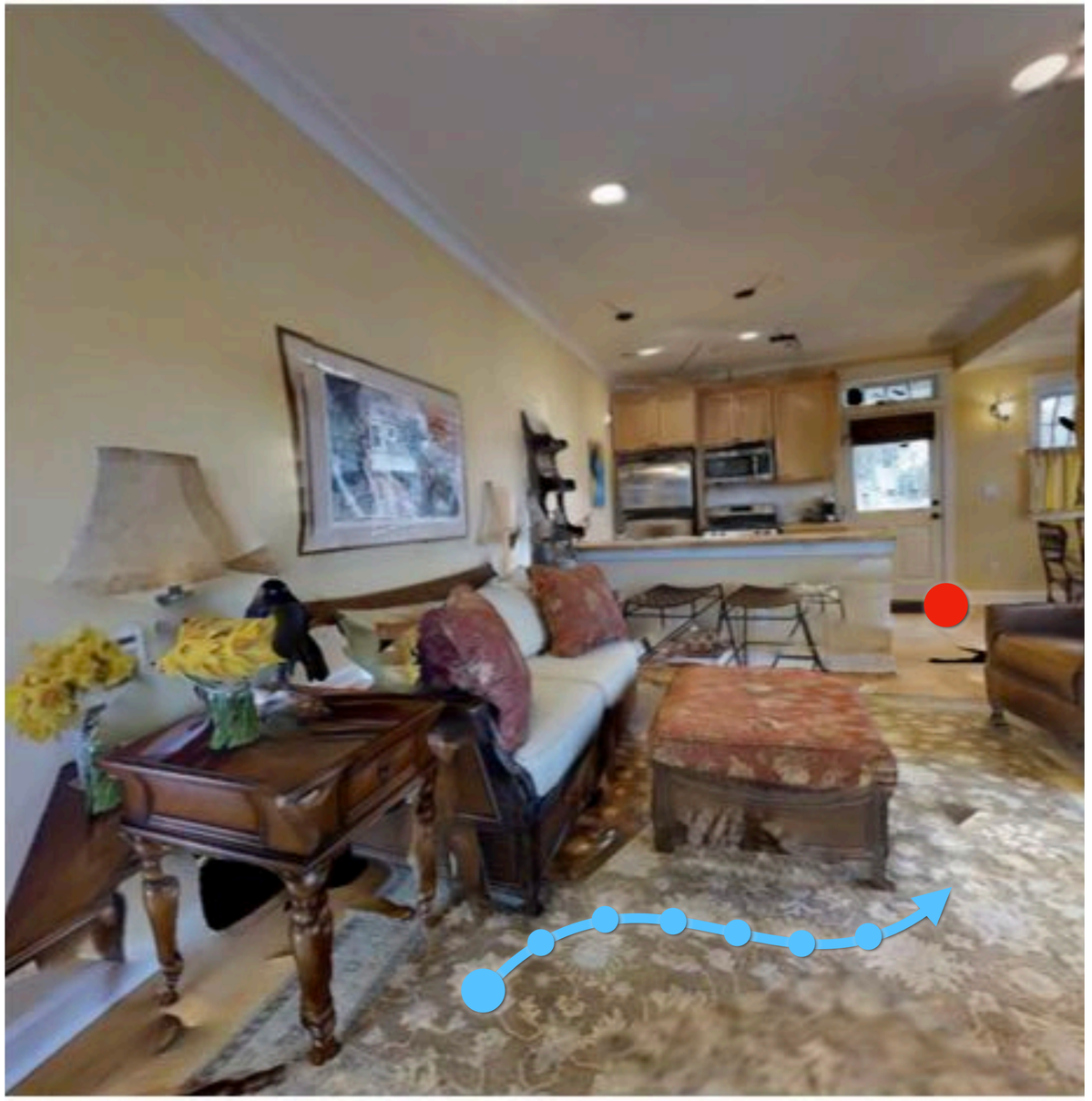
sequence of observations



sequence of agent actions

compute loss gradients π_θ → update model via SGD

Reward: change in distance from goal



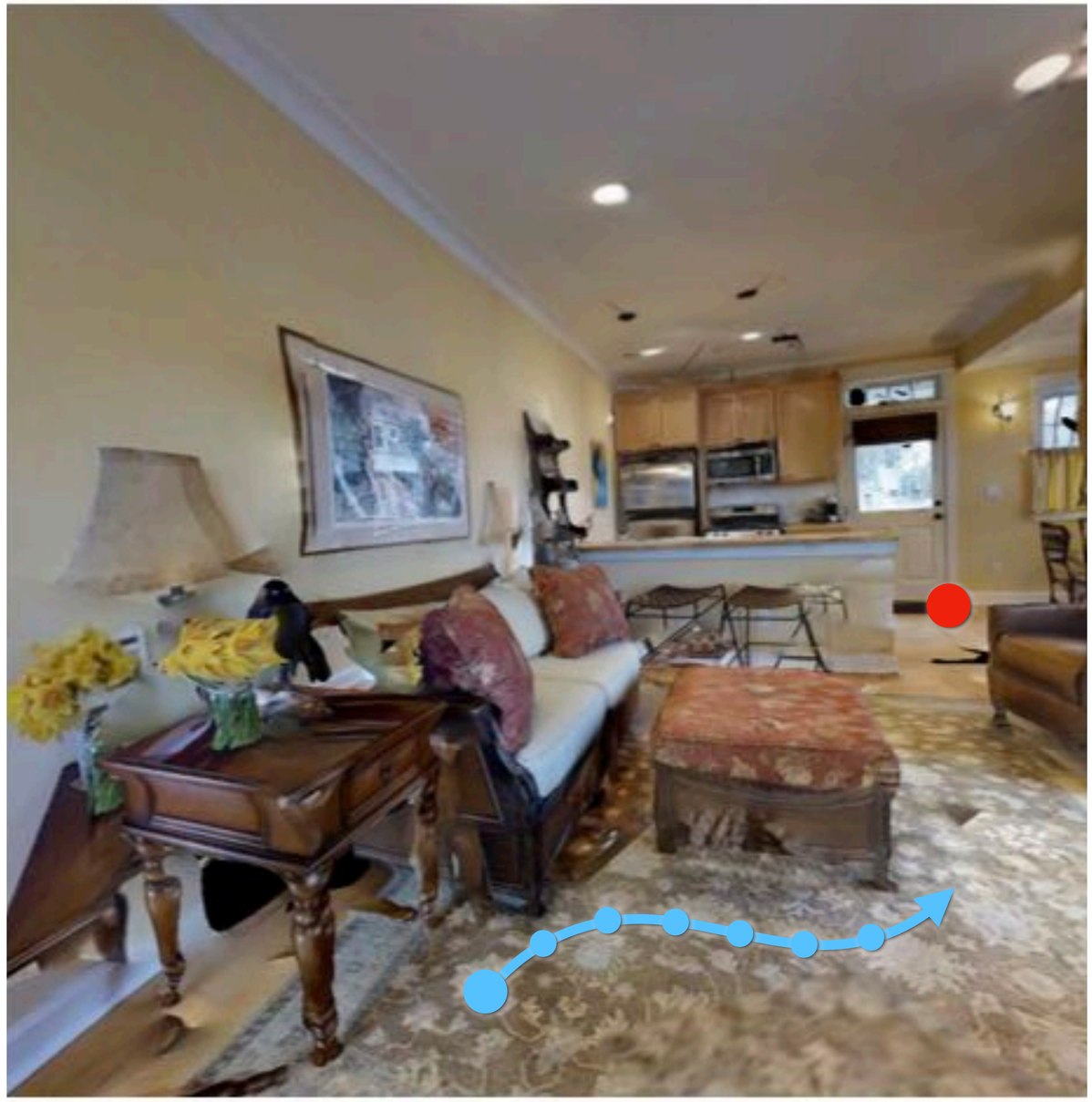
RL in 30 seconds

Model Inference

environment observation e.g. RGB image → π_θ → agent action

Model Training

Rollout → compute loss gradients π_θ → update model via SGD

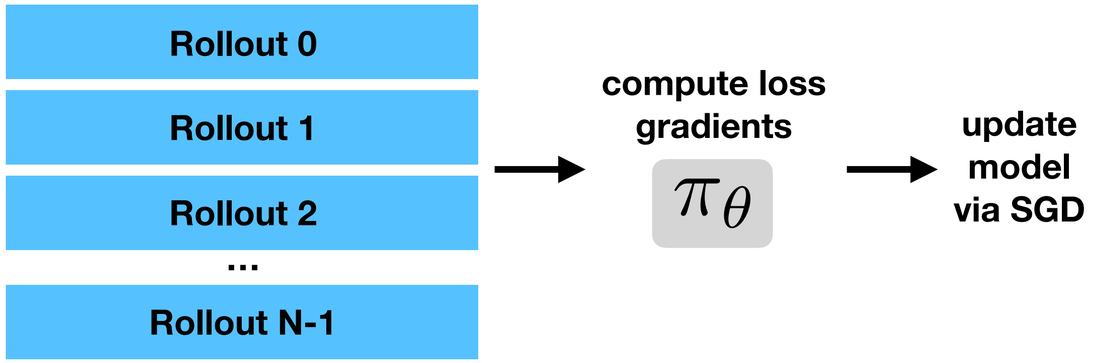


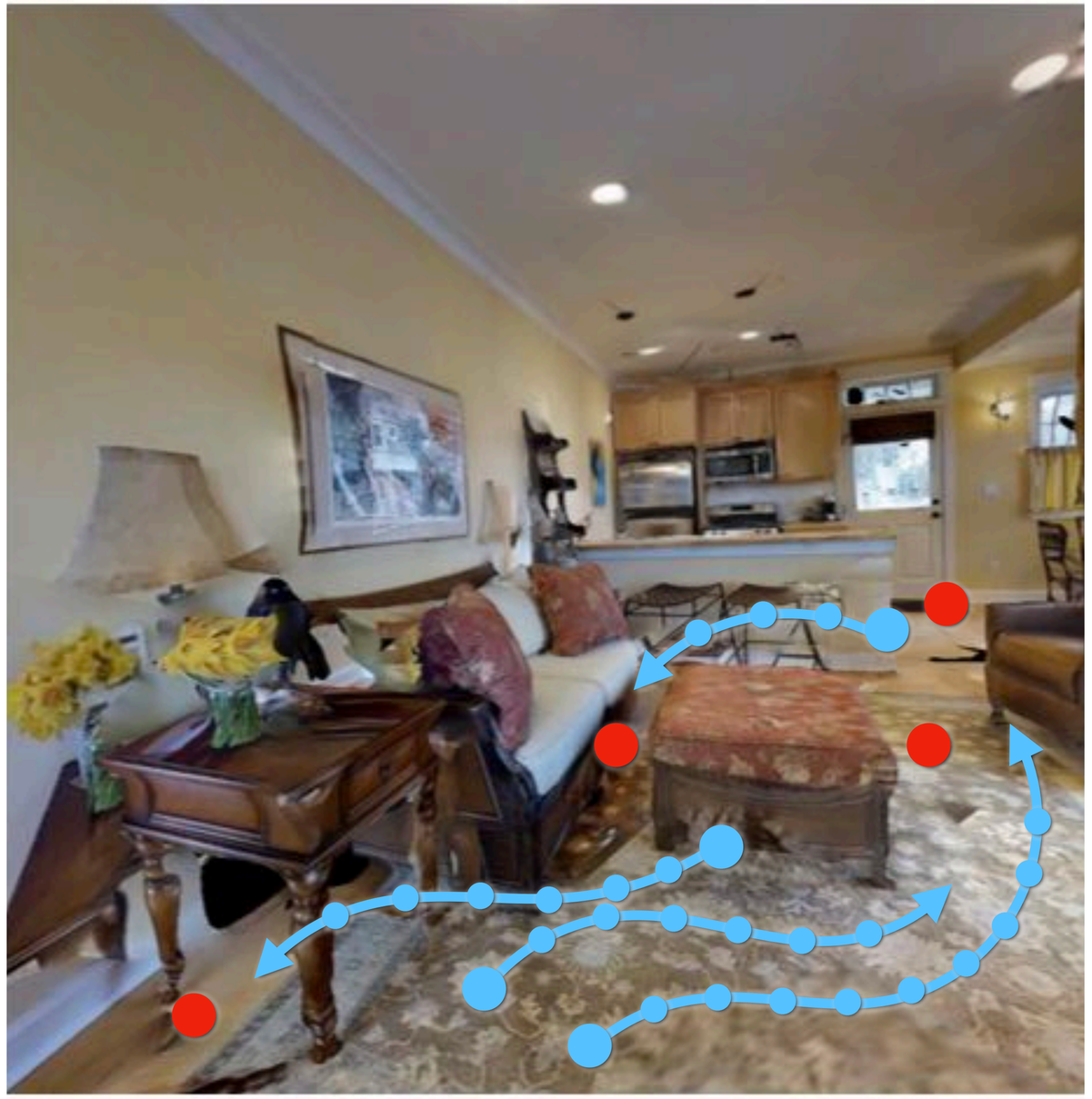
RL in 30 seconds

Many rollouts:

- **Agents independently navigating same environments**

Batch Model Training

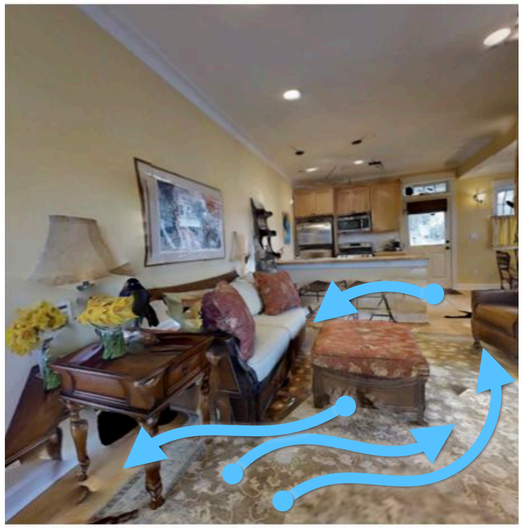




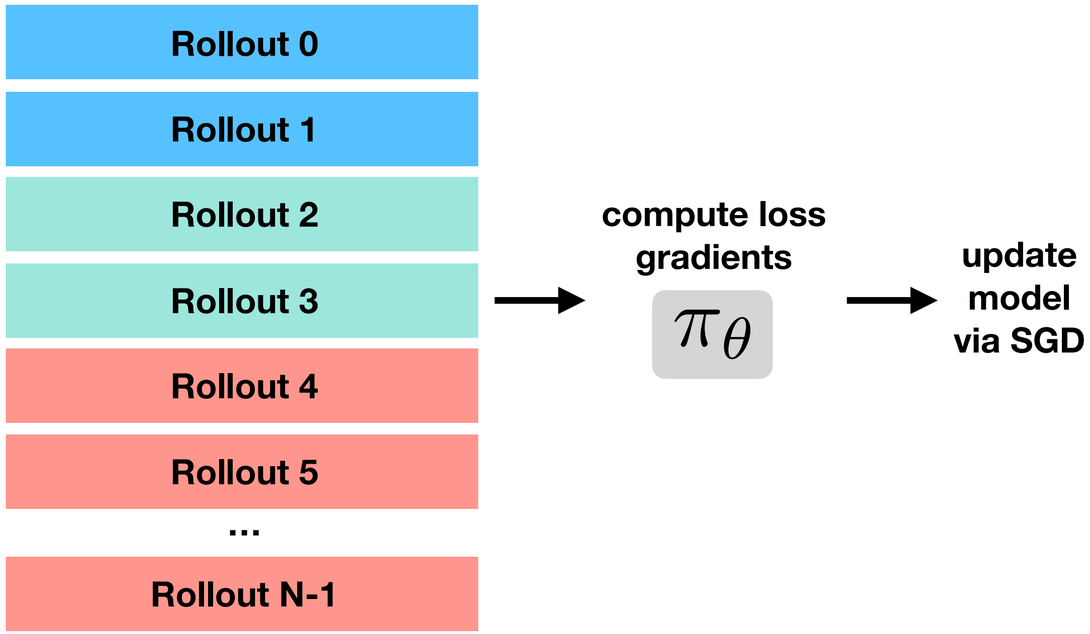
RL in 30 seconds

Many rollouts:

- **Agents independently navigating same environments**
- **Or different environments**

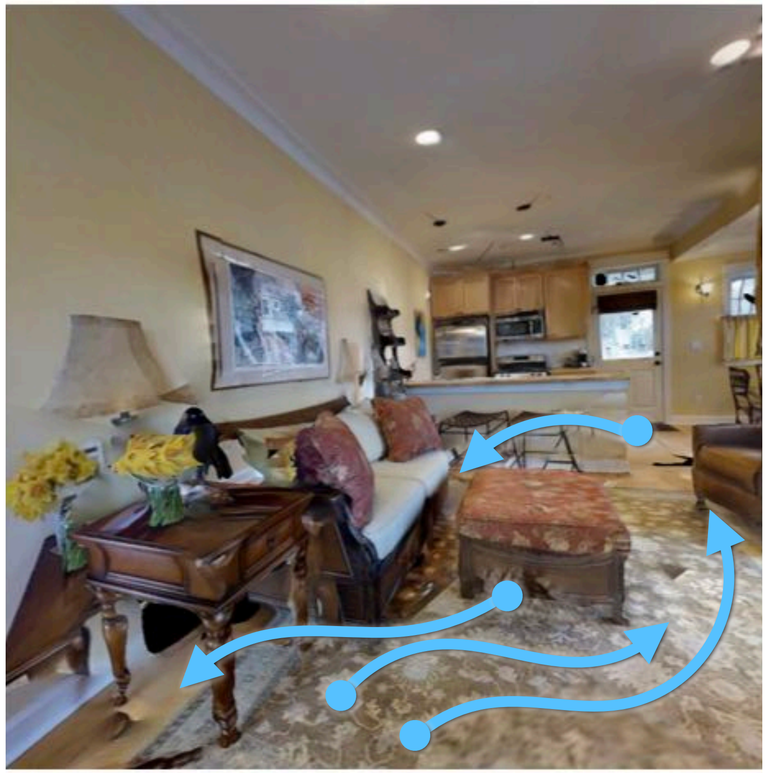


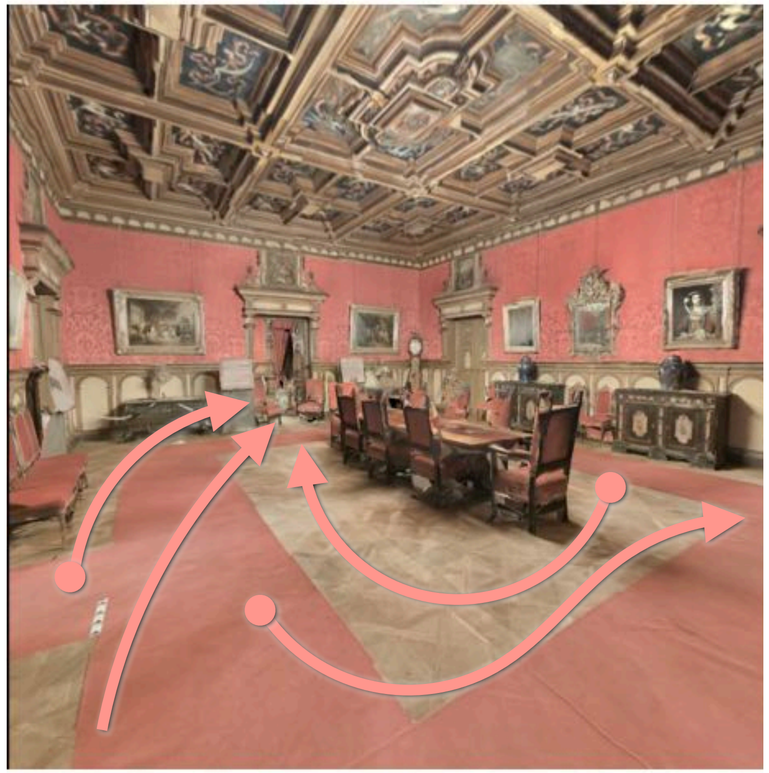
Batch Model Training



Learning skills can require many trials (billions) of learning experience

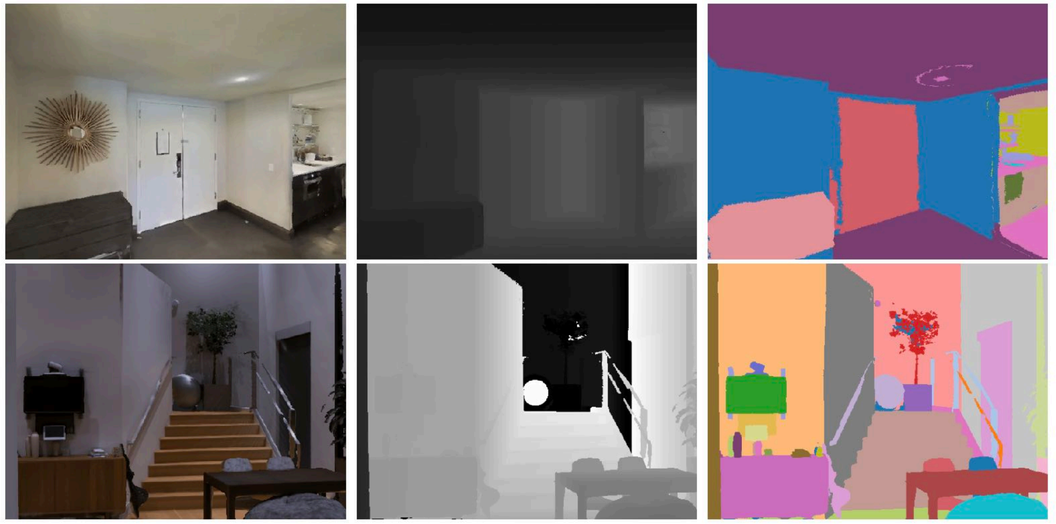
- Training in diverse set of virtual environments
- Many training trials in each environment

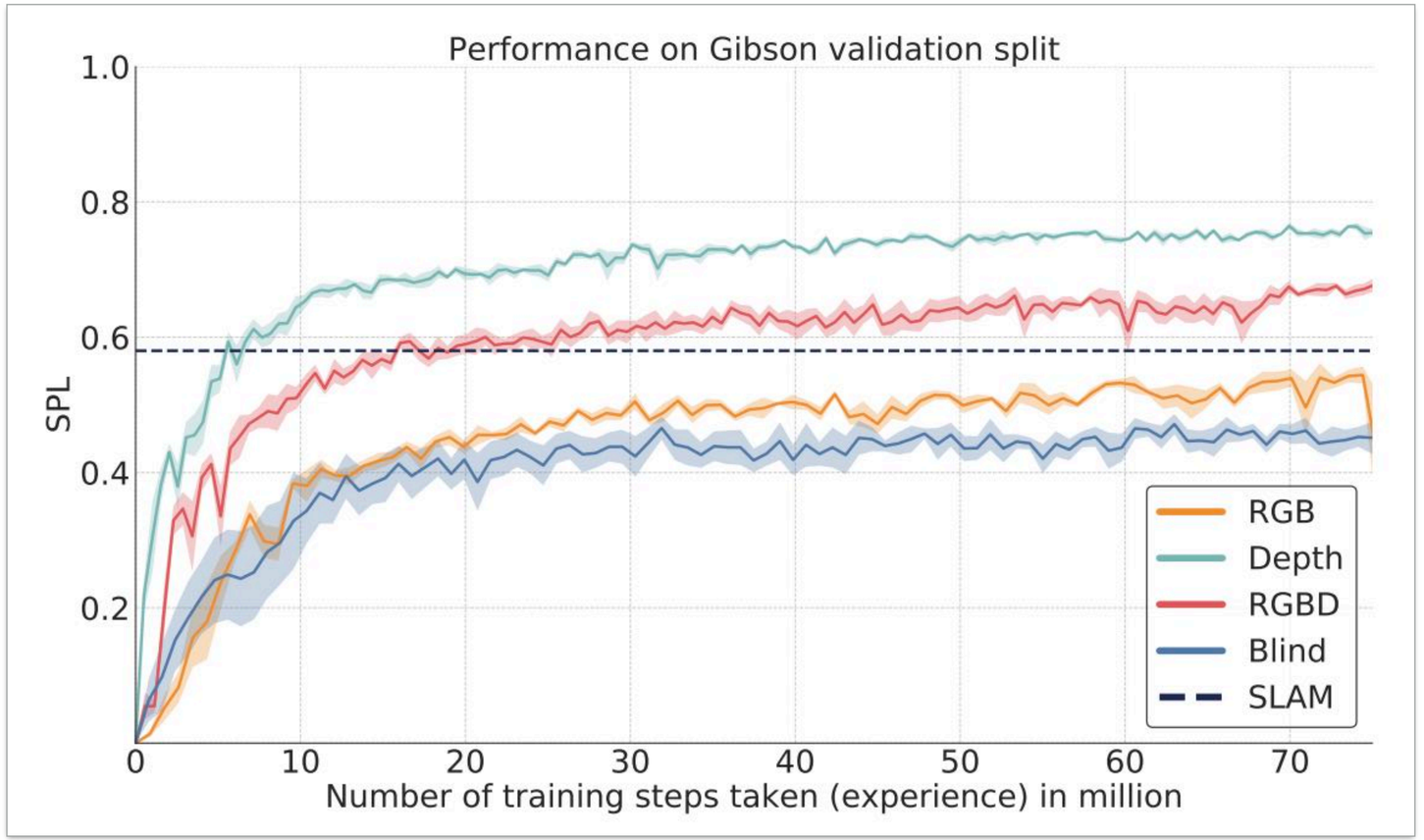




Need significant amounts of simulated experience to learn skills

Example: even for simple PointGoal navigation task: need billions of steps of "experience" to exceed traditional non-learned approaches





Training agents to perform different tasks requires implementation of virtual environments to carry out the required simulations

Many interactive virtual home environments

Navigate to a location

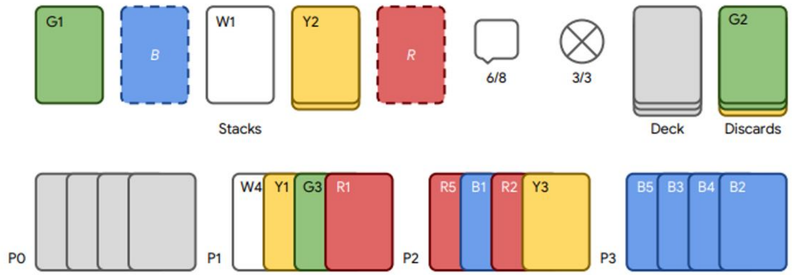
Find an object

Rearrange the room so objects are in desired locations

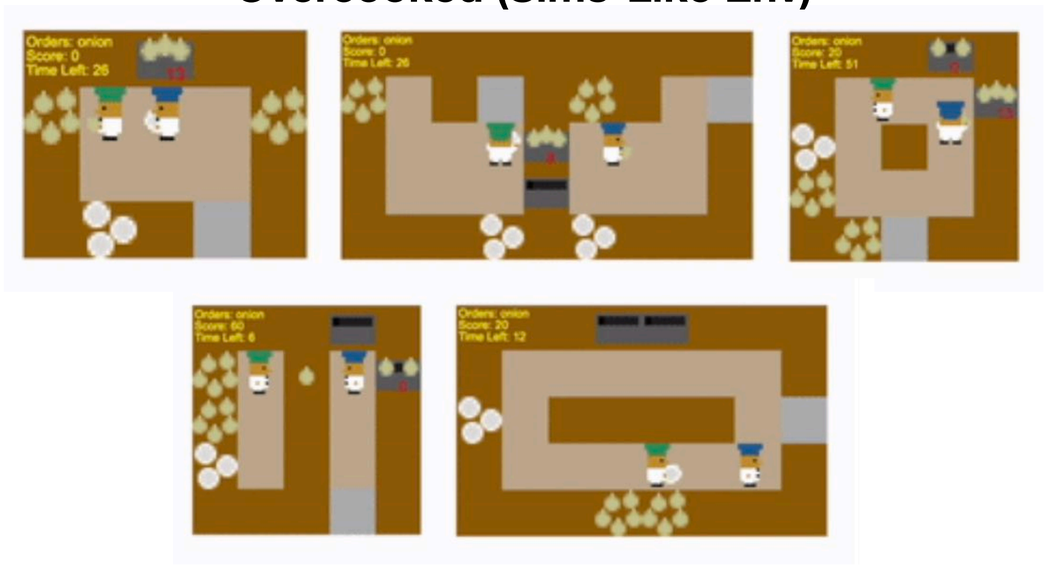
Pour oneself a glass of milk

Multi-agent games

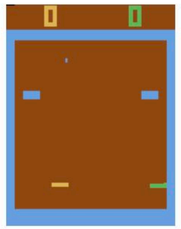
Hanabi (Card Game)

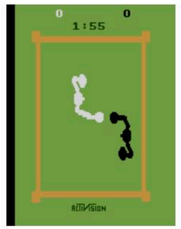


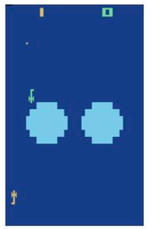
Overcooked (Sims-Like Env)

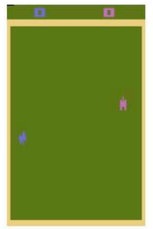


Atari Games

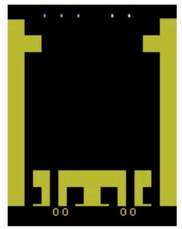

Basketball Pong

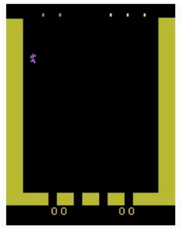

Boxing

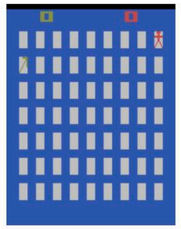

Combat Plane

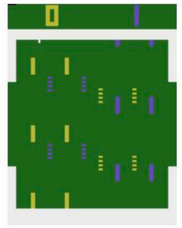

Combat Tank


Double Dunk


Entombed Competitive


Entombed Cooperative


Flag Capture

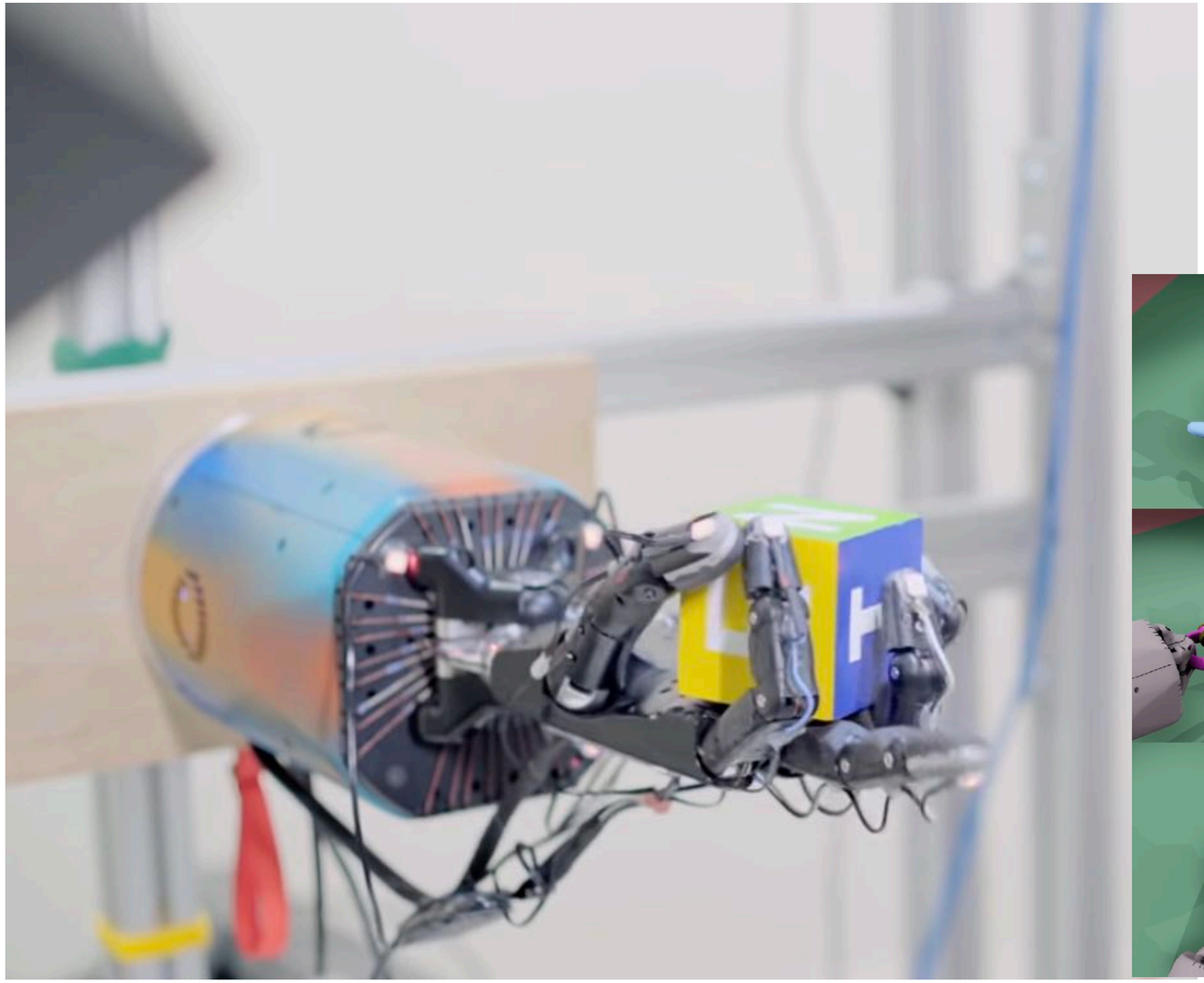

Foozpong

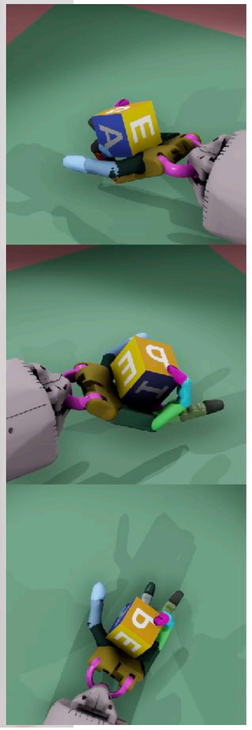

Ice Hockey

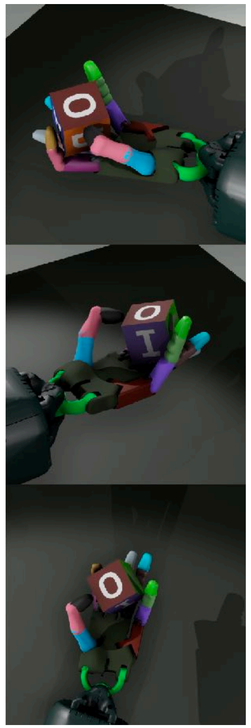

Joust

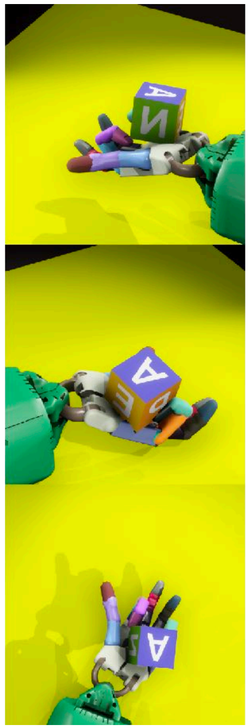

Mario Bros

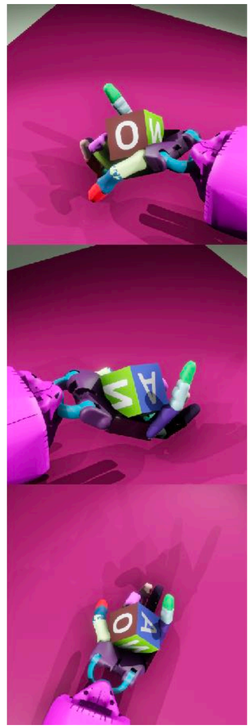
Robot hand manipulation









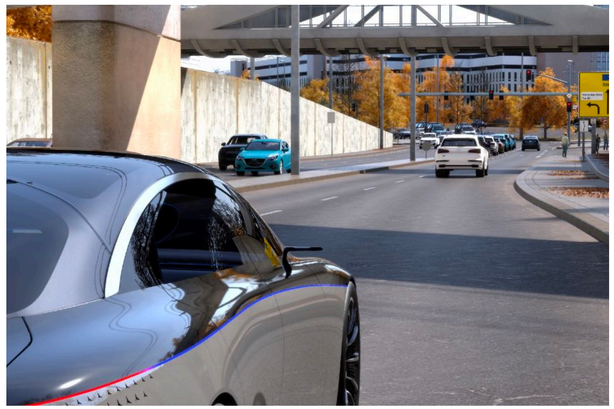
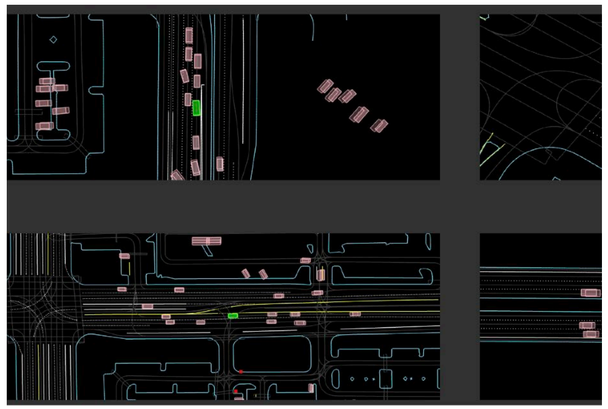
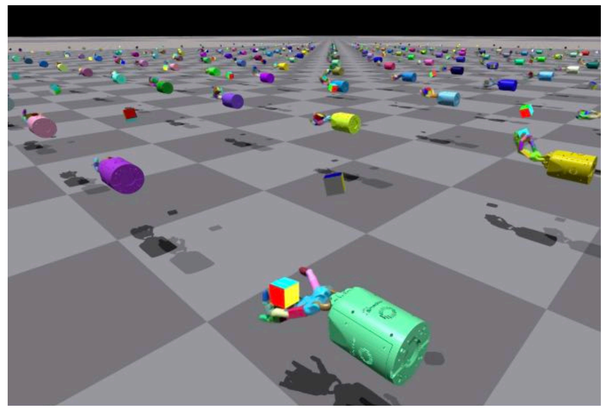



Random incoming ball + Random target
Practice in simulation…

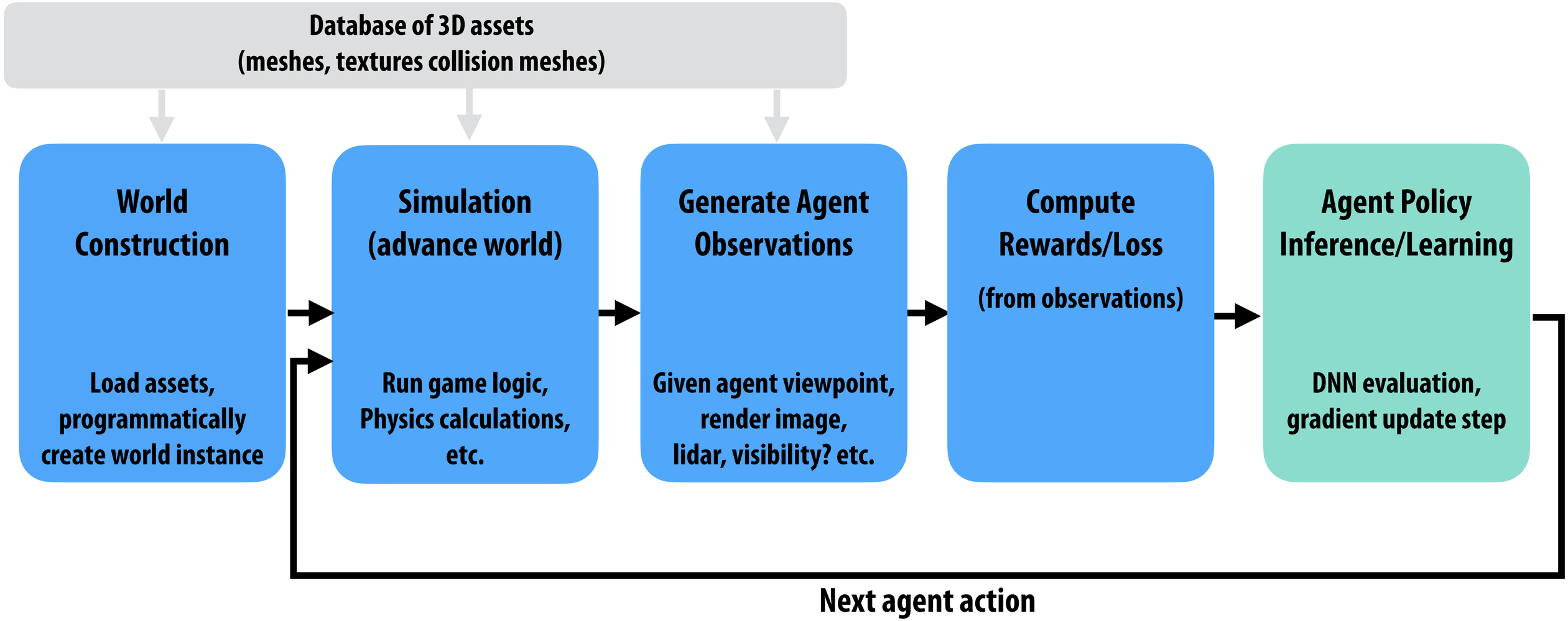
The story so far

1. Training and agent to learn complex skills can require many (millions, even billions) of trial-and-error steps (aka large amounts of "training experience")

2. Researchers create virtual environments to simulate all this experience.

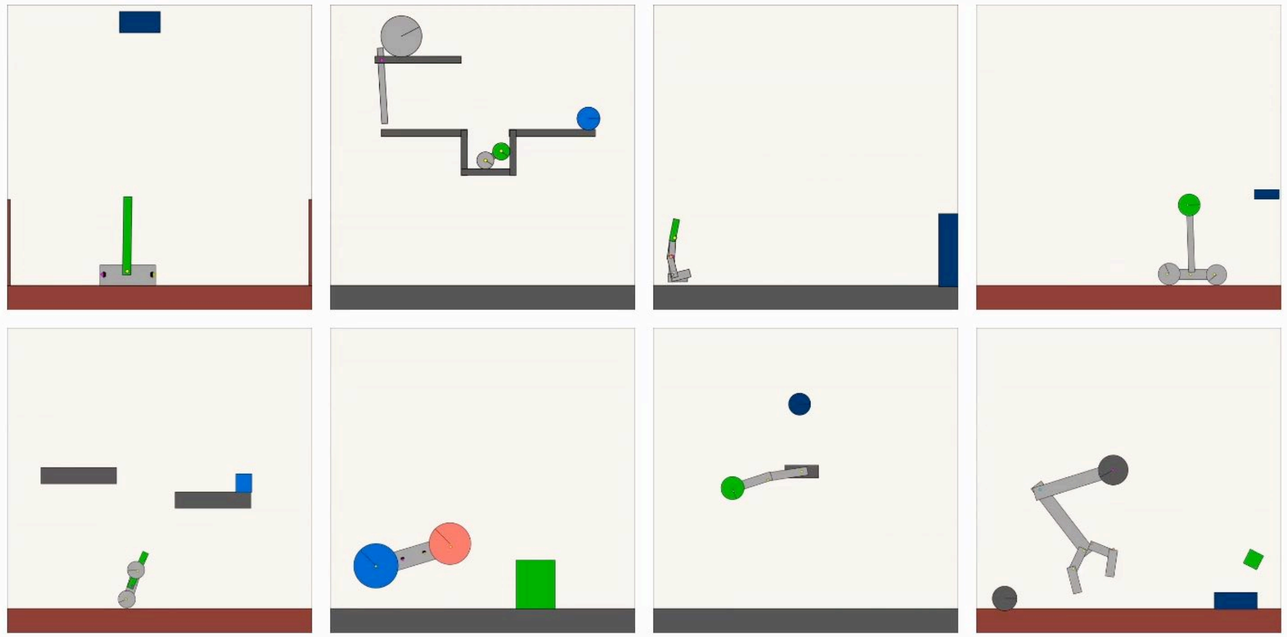




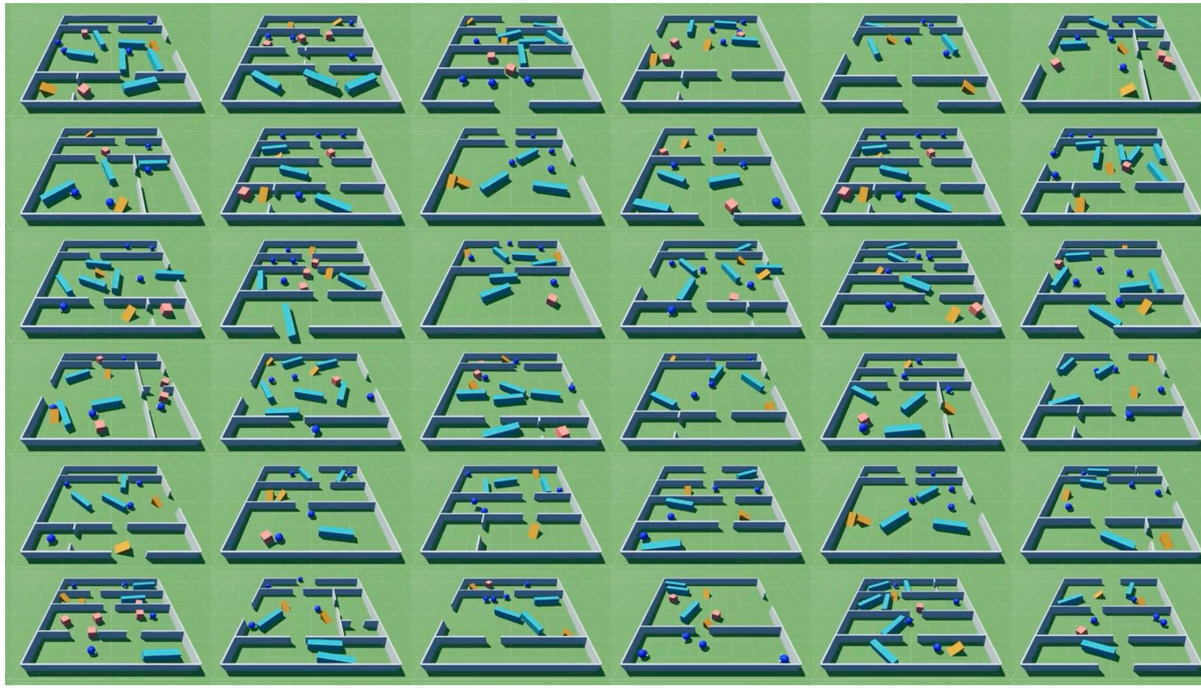

Basic training system components



Procedural world generation

- Many examples of procedurally creating novel environments per “training episode”.
- Interesting questions about how to pick examples to train on

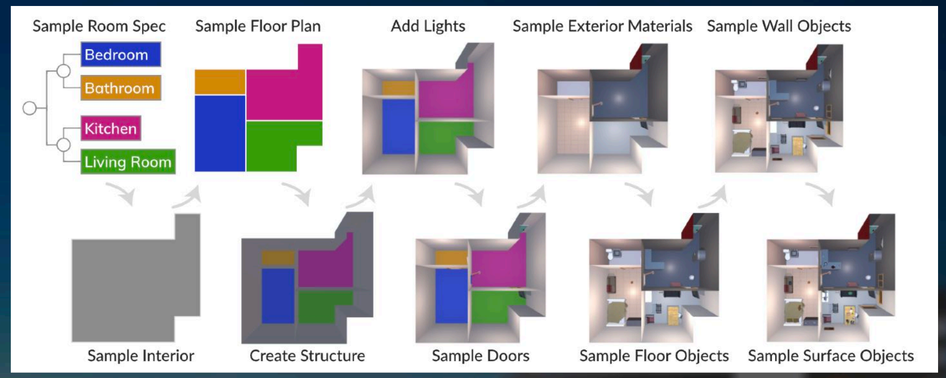




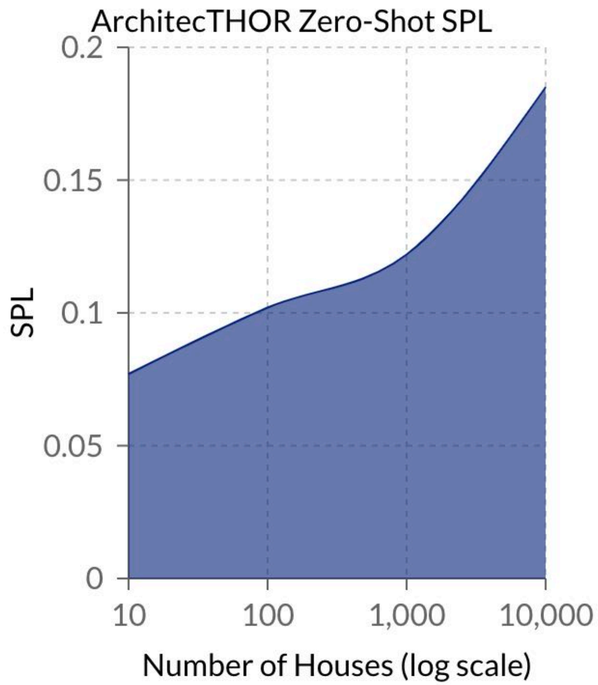
Value of diversity of scenes

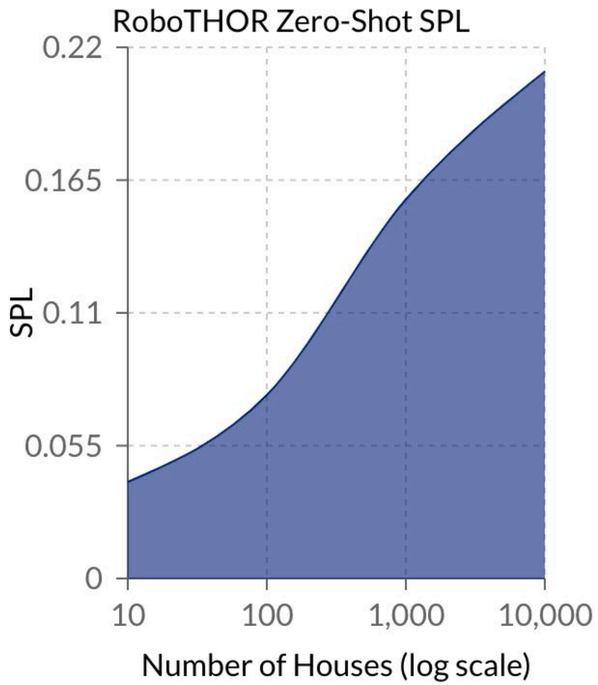
1. Many environments, many variants of task

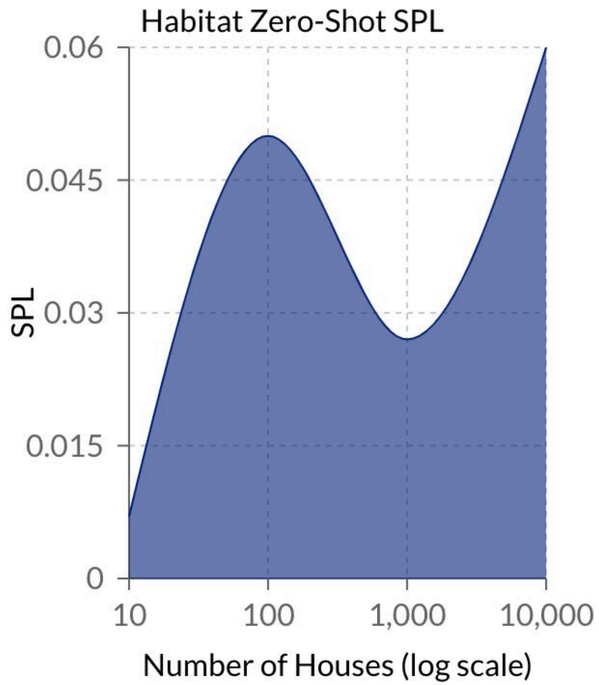
Procedurally generated floorpans, furniture arrangements, random material assigmments, etc.

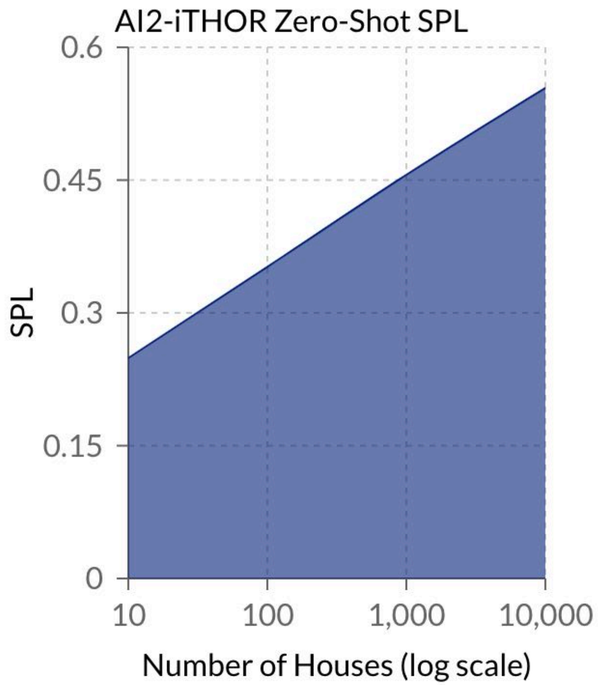


Greater diversity of scenes wins







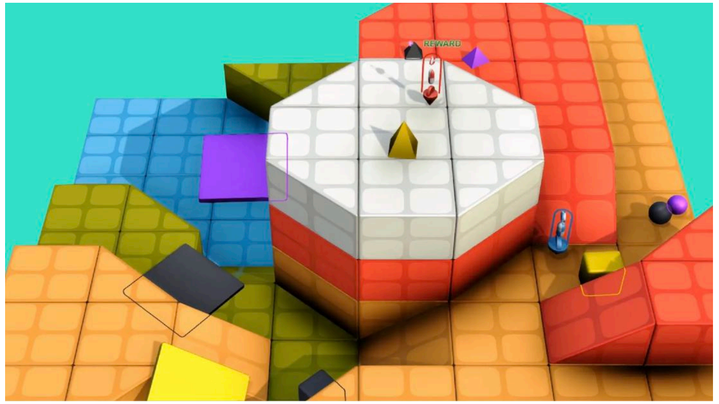


Better off training on a large number of highly diverse scenes than a small number of photorealistic ones

Value of diversity of tasks

DeepMind's XLand: procedurally generate terrains and the rules of the game.

Wide range of "games" requiring different strategies and skills can be defined by specifying goals for agents



Example 1: agent 1 must find a yellow sphere and hold it, while agent 2 must stand by a yellow pyramid

g_1 := `hold(me, yellow sphere)`

g_2 := `near(me, yellow pyramid)`

Example 2: goals for hide and seek: Agent 1 should move to see its opponent. Agent 2's goal is to not be seen by agent 1.

g_1 := `see(me, opponent)`

g_2 := `not(see(opponent, me))`

Example 3: encourage teamwork. Give both agents the same goal of moving one object by another

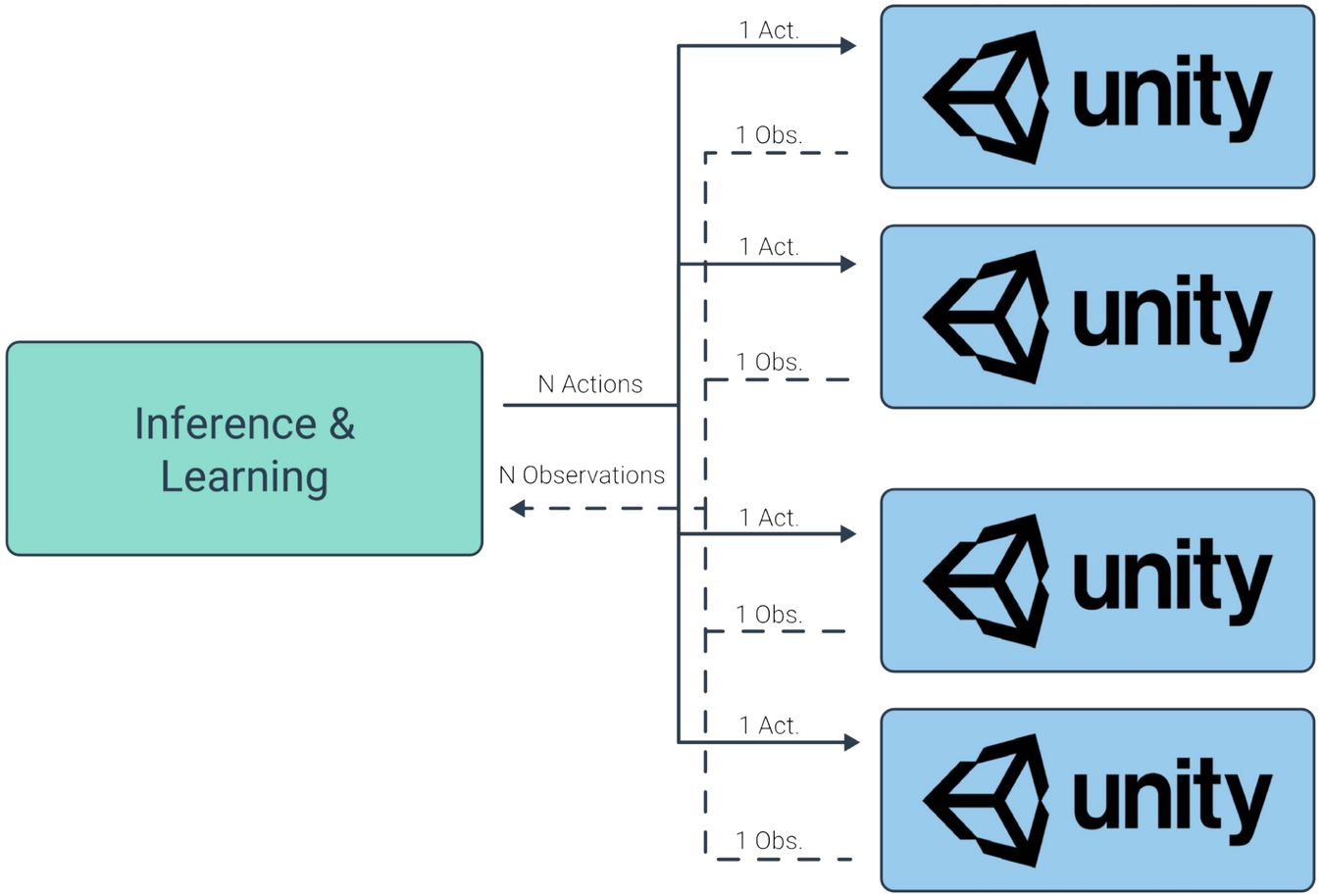
g_1 := `near(yellow pyramid, yellow sphere)`

g_2 := `near(yellow pyramid, yellow sphere)`

The result of this training process is an agent that is generally capable across the held-out evaluation space. Qualitatively, we observe the agent exhibiting behaviours that are generally applicable, rather than optimal for any specific task. Examples of such behaviours include: experimentation through directed exploration until the agent recognises a rewarding state has been achieved; seeking another player out to gather information of its state irrespective of its goal; and tagging another player if it is holding an object that is related to the agent's goal irrespective of that player's intention. We also probe quantitatively the behaviour of agents in test-time multi-agent situations and see evidence of cooperation emerging with training. In addition to the agent exhibiting zero-shot capabilities across a wide evaluation space, we show that finetuning on a new task for just 100 million steps (around 30 minutes of compute in our setup) can lead to drastic increases in performance relative to zero-shot, and relative to training from scratch which often fails completely.

Need for high-throughput simulation

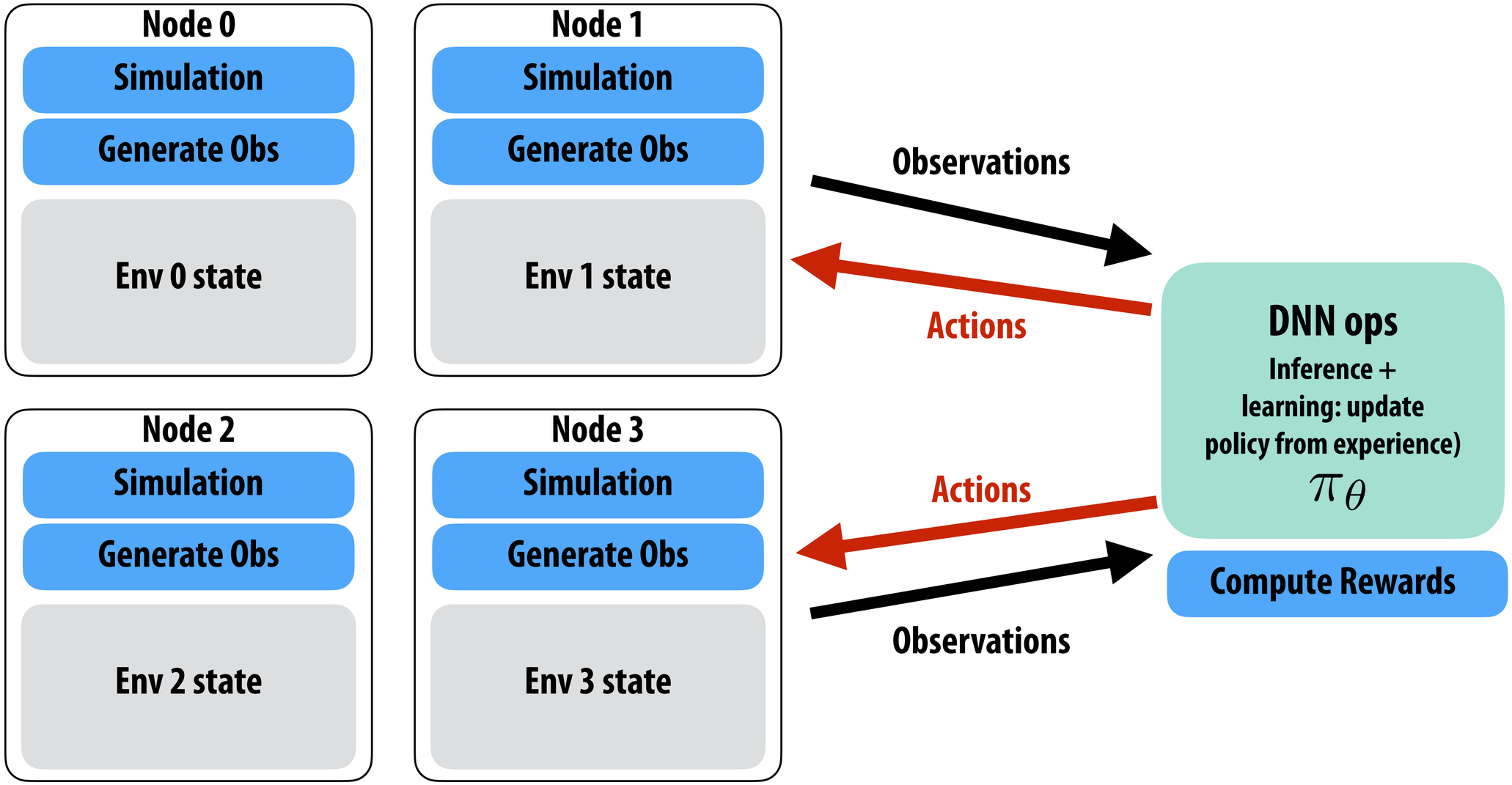
Common parallel simulation approach: treat simulator as a black box, gain high throughput via scale-out parallelization



Treat existing simulation engines as a black box.

Run many copies of the black box in parallel.

A basic design: parallelize over workers



One example of this design: EnvPool (one multi-core node)

- **Pros:**
  - Use any existing simulator, unmodified
  - Collects observations from environments, provides them to Python as a Tensor
- **Cons:**
  - See upcoming slides (simulator-learning code sync costs, running many independent simulators is not optimal on high throughput machines)

README | Code of conduct | Apache-2.0 license

pypi v0.8.4 | PyPI Download 119k | arXiv 2206.10558 | docs passing | Bazel Build and Test passing | issues 49 open | Stars 1k | Forks 90 | license Apache-2.0

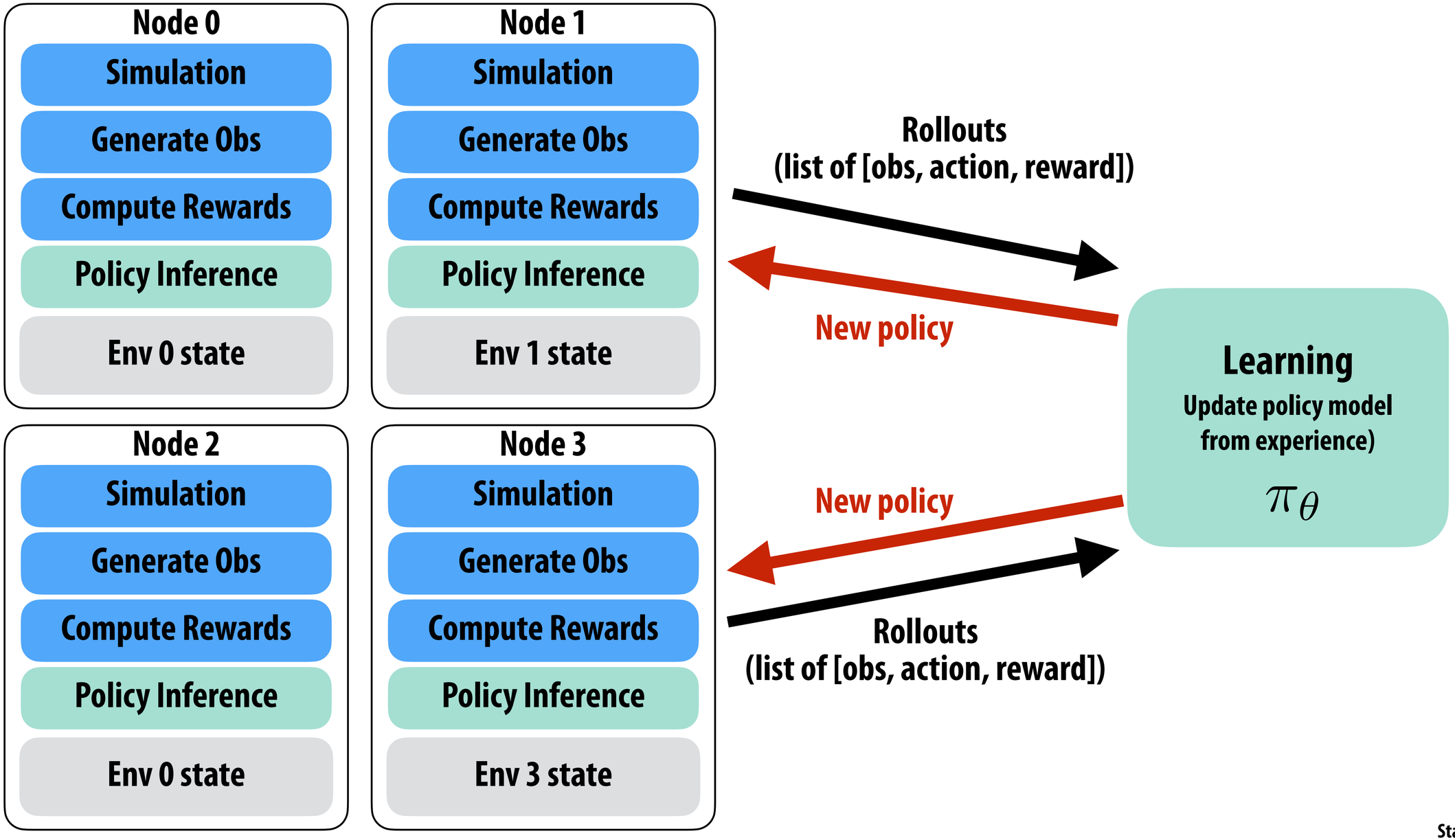
EnvPool is a C++-based batched environment pool with pybind11 and thread pool. It has high performance (~1M raw FPS with Atari games, ~3M raw FPS with Mujoco simulator on DGX-A100) and compatible APIs (supports both gym and dm_env, both sync and async, both single and multi player environment). Currently it supports:

- [x] Atari games
- [x] Mujoco (gym)
- [x] Classic control RL envs: CartPole, MountainCar, Pendulum, Acrobot
- [x] Toy text RL envs: Catch, FrozenLake, Taxi, NChain, CliffWalking, Blackjack
- [x] ViZDoom single player
- [x] DeepMind Control Suite
- [x] Box2D
- [x] Procgen
- [x] Minigrid

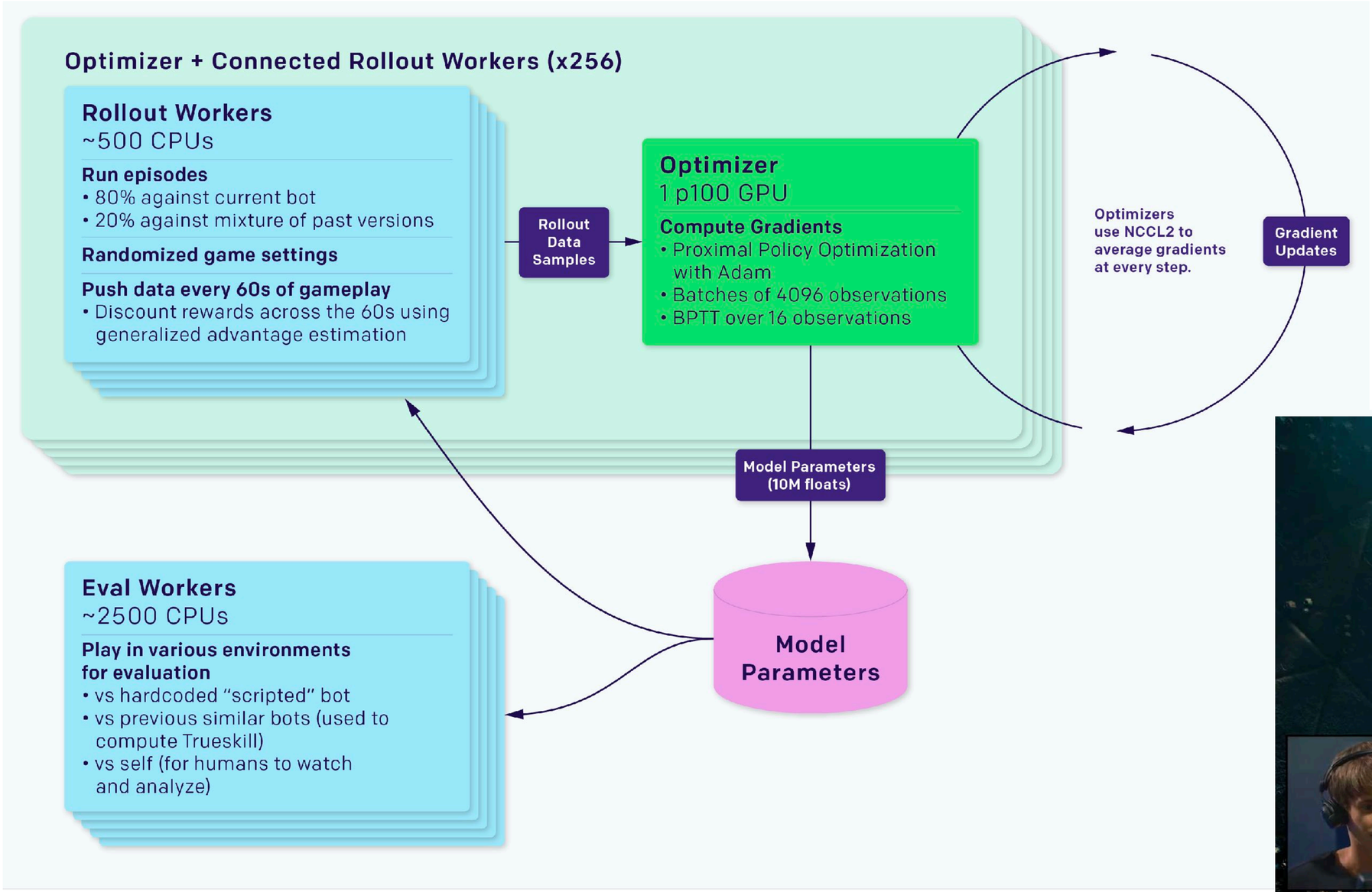
Here are EnvPool's several highlights:

- Compatible with OpenAI `gym` APIs, DeepMind `dm_env` APIs, and `gymnasium` APIs;
- Manage a pool of envs, interact with the envs in batched APIs by default;
- Support both synchronous execution and asynchronous execution;
- Support both single player and multi-player environment;
- Easy C++ developer API to add new envs: Customized C++ environment integration;
- Free **~2x** speedup with only single environment;
- **1 Million** Atari frames / **3 Million** Mujoco steps per second simulation with 256 CPU cores, **~20x** throughput of Python subprocess-based vector env;
- **~3x** throughput of Python subprocess-based vector env on low resource setup like 12 CPU cores;
- Comparing with existing GPU-based solution (Brax / Isaac-gym), EnvPool is a **general** solution for various kinds of speeding-up RL environment parallelization;

Similar design for distributed system: parallelize over workers



Example: Rapid by (OpenAI)



OpenAI Five stats (Dota 2)

	OPENAI FIVE
CPUs	128,000 preemptible CPU cores on GCP
GPUs	256 P100 GPUs on GCP
Experience collected	~180 years per day (~900 years per day counting each hero separately)
Size of observation	~36.8 kB
Observations per second of gameplay	7.5
Batch size	1,048,576 observations
Batches per minute	~60

Design issues of basic scale-out approach

- Inefficient simulation/rendering: low-complexity worlds do not make good use of a modern parallel processor's resources
  - GPUs won't achieve high-throughput rendering/physics with smaller workloads

- Inefficient communication between simulation and inference/training

- Duplication of computation and memory footprint (for scene data) across environment simulator instances

- Seems wasteful, right?

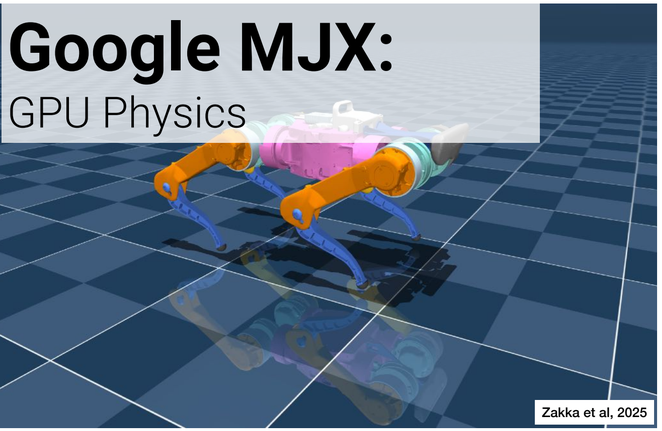
A new visual computing systems research question:

Can we execute embodied AI training more efficiently if we architect a world simulation engine from the ground up to process many independent worlds at once?

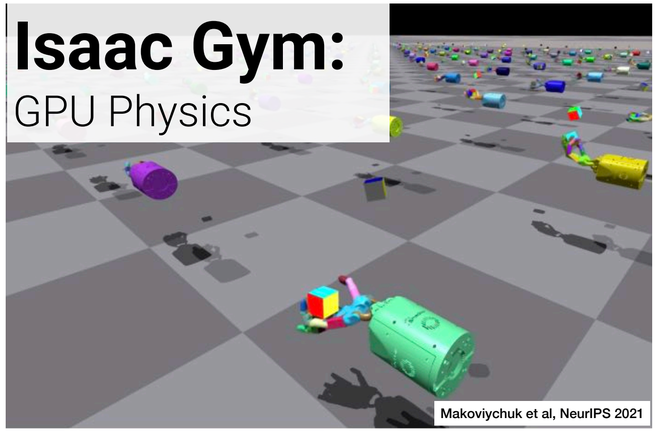
More recent design: Batch Simulators: achieve millions of steps/sec by executing thousands of environments in parallel on a single GPU

But… requires a simulator rewrite for the GPU

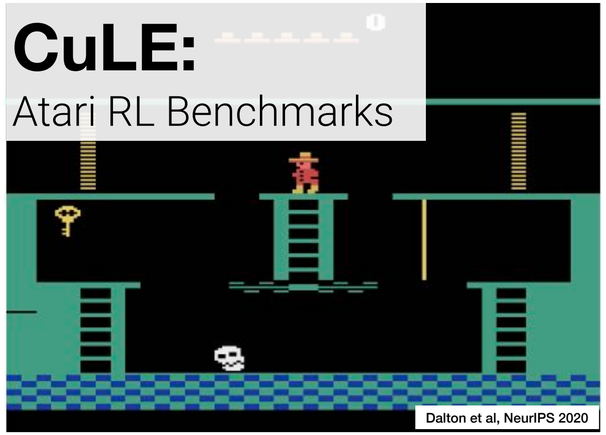
Google MJX:
GPU Physics



Isaac Gym:
GPU Physics

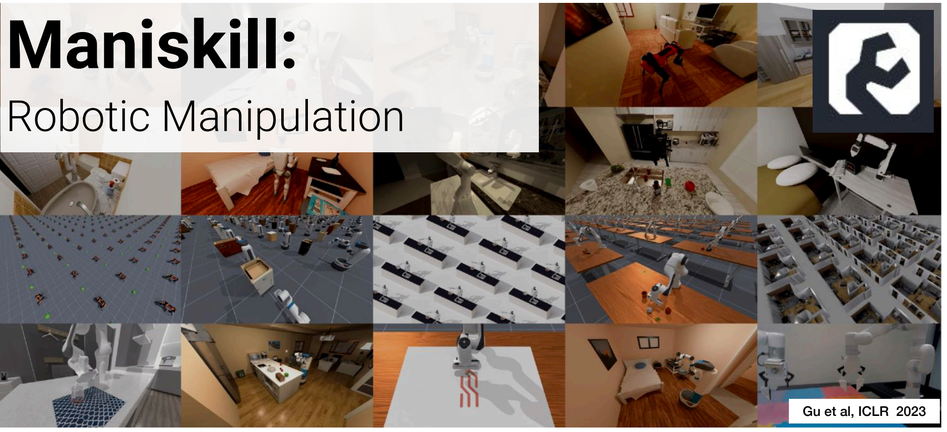


CuLE:
Atari RL Benchmarks

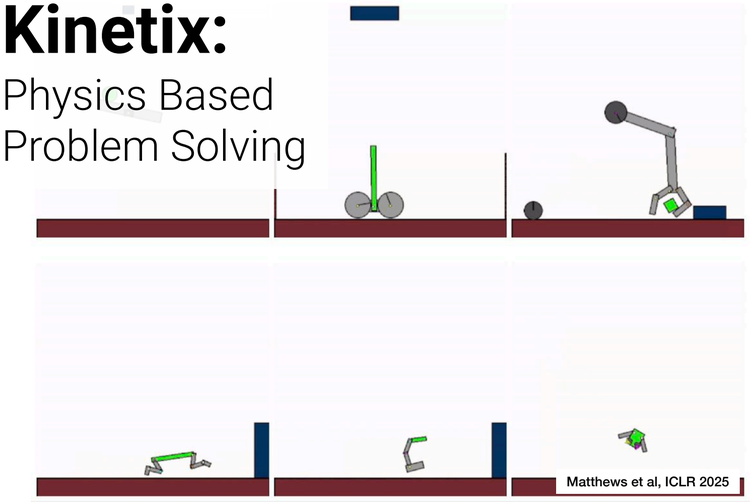


BPS3D:
Home Navigation

Maniskill:
Robotic Manipulation

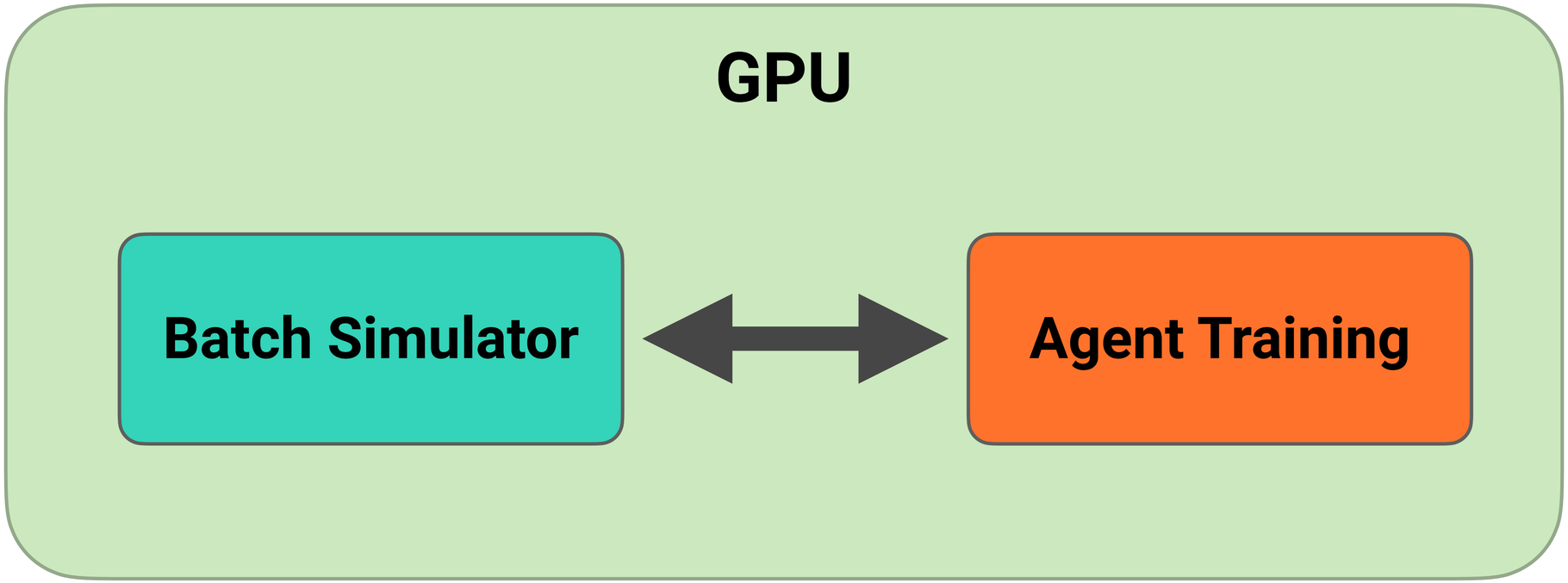


Kinetix:
Physics Based
Problem Solving



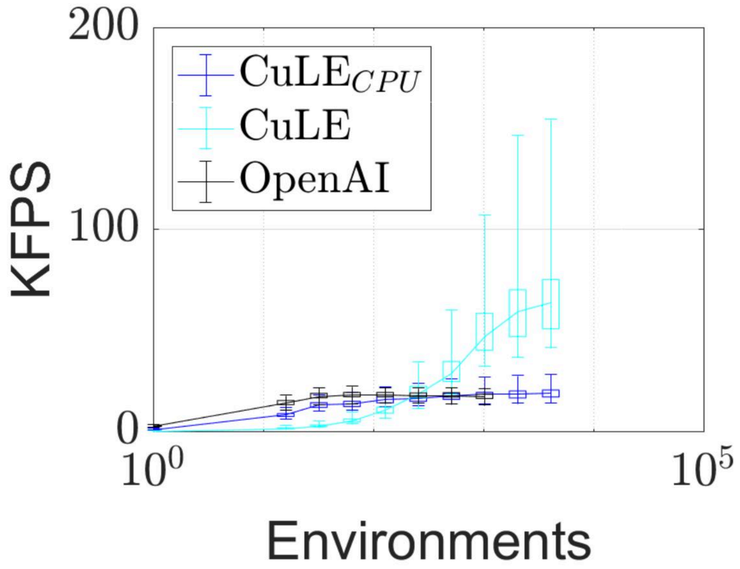
Craftax:
Open-Ended 2D Exploration

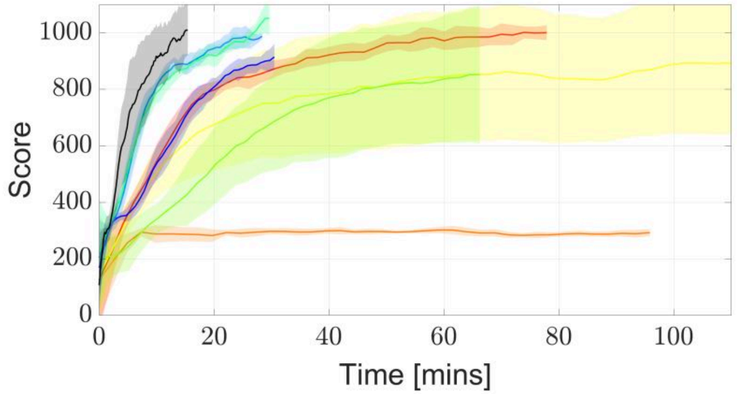
Batch simulator's global view of a batch of environments allows much higher training efficiency



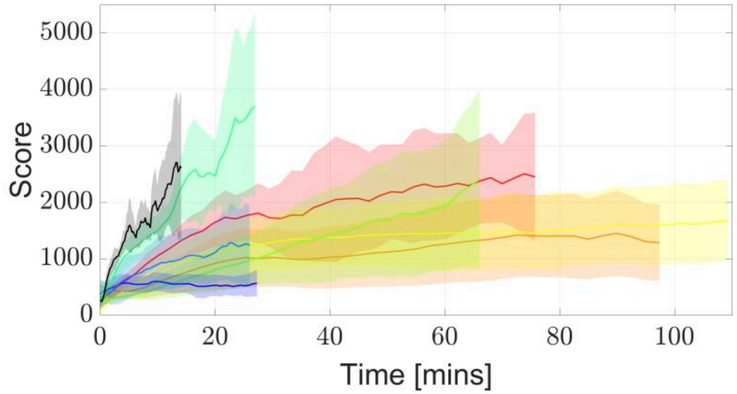
CuLE: Rewriting an Atari emulator in CUDA

- One CUDA thread = work for one Atari simulator instance
- Large numbers of threads execute

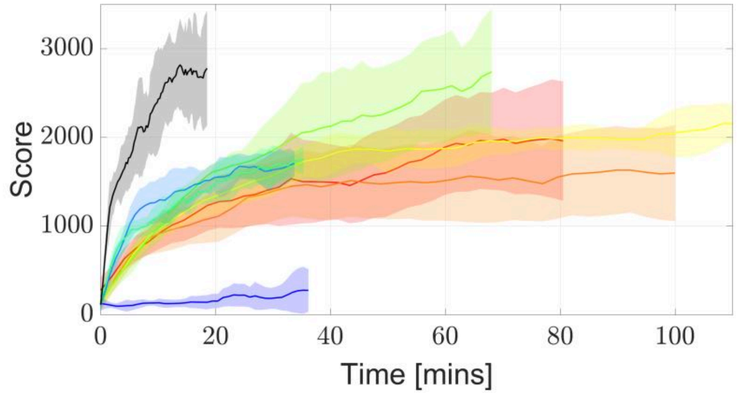




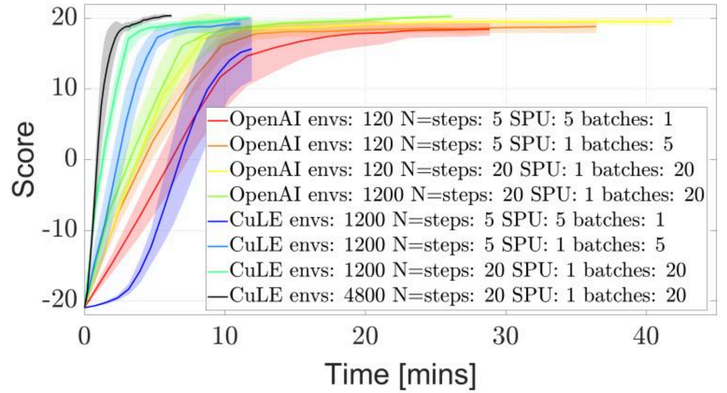
(a) Assault, 20M training frames



(b) Asterix, 20M training frames



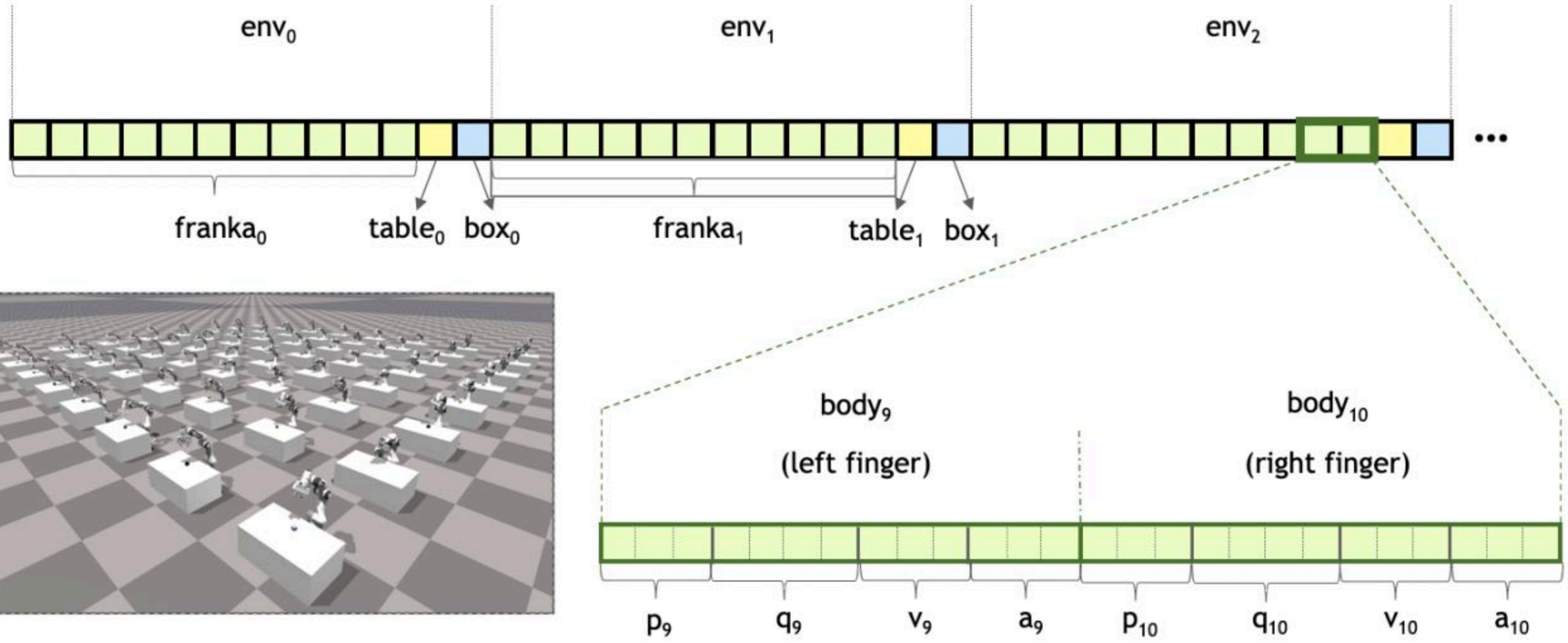
(c) Ms-Pacman, 20M training frames



(d) Pong, 8M training frames

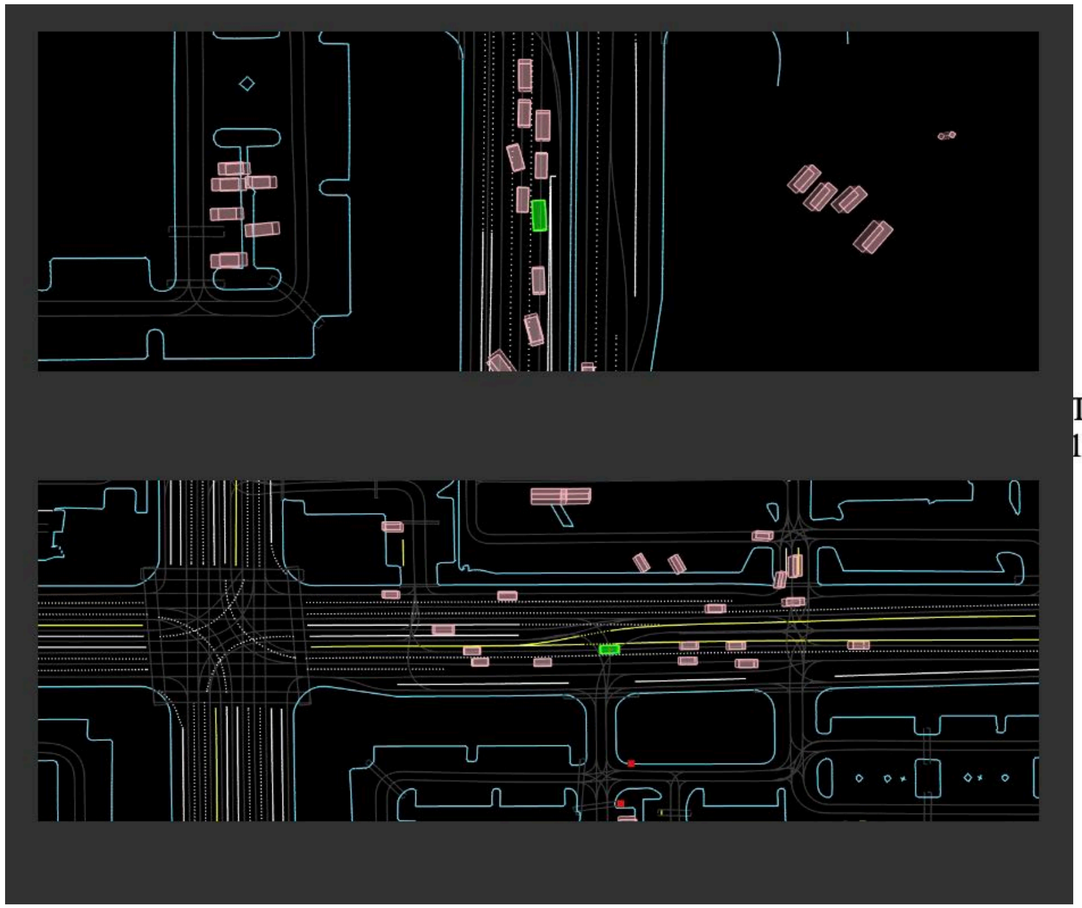
NVIDIA Issac Gym

- Batched many-environment execution applied to rigid body physics sim
- Simulate 100's to 1000's of world environments simultaneously on the GPU
- Current state for all environments packaged in a single PyTorch tensor
- User can write GPU-accelerated loss/reward functions in PyTorch on this tensor
- Result: tight loop of simulate/infer/train



Waymax

- Self-driving car simulator built using Jax programming environment
- Environment state stored in JAX tensors (reserve space for max number of objects across all environments in a batch)

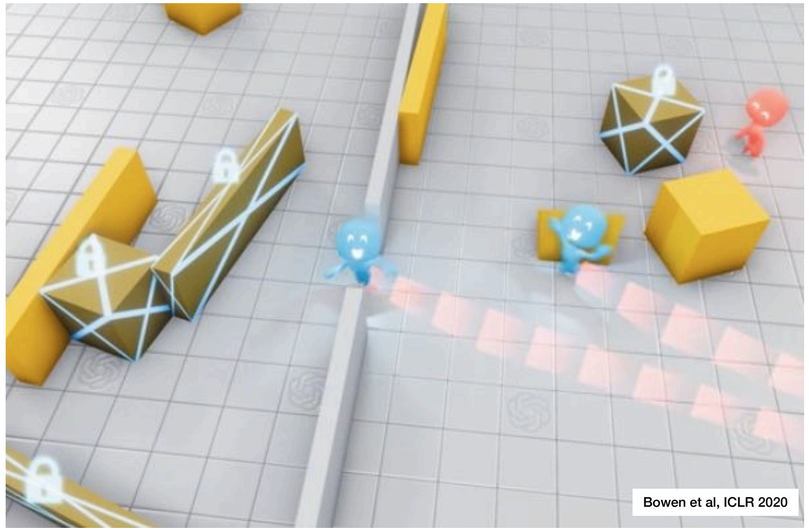


	Device	BS-1	BS-16	Reset	Step	Transition	Metrics	RolloutExpert
Single-Agent Env	CPU	✓		1.09	131	0.90	112	1.0×10^4
	CPU		✓	12.2	1.7×10^3	10.9	1.69×10^3	1.4×10^5
	GPU-v100	✓		0.58	0.75	0.47	0.21	56.2
	GPU-v100		✓	0.67	2.48	0.52	2.27	279
Multi-Agent Env	CPU	✓		6.23	129	1.01	112	1.1×10^4
	CPU		✓	49.8	1.1×10^3	14.3	1.72×10^3	1.6×10^5
	GPU-v100	✓		0.64	0.92	0.53	0.19	73.3
	GPU-v100		✓	0.81	2.86	0.51	2.24	OOM

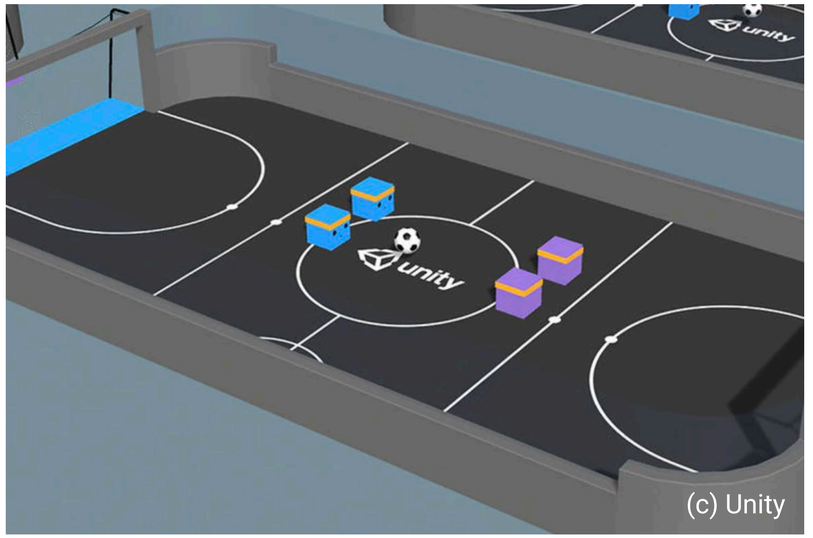
Table 2: Runtime benchmark in milliseconds: the environment controls all objects in the scene (up to 128 as defined in WOD).

What about building new batch simulators for training new kinds of agents?

Support Novel ML Research



Train Agents for New Game

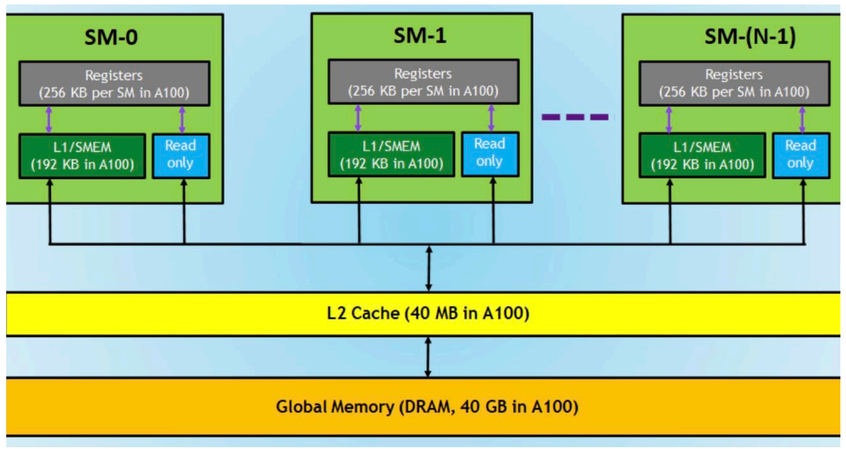


Tons of engineering to build new GPU-accelerated batch simulator from scratch!

Task Knowledge

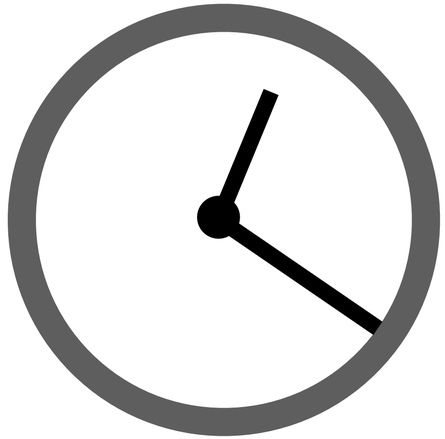
+

GPU Programming Skill

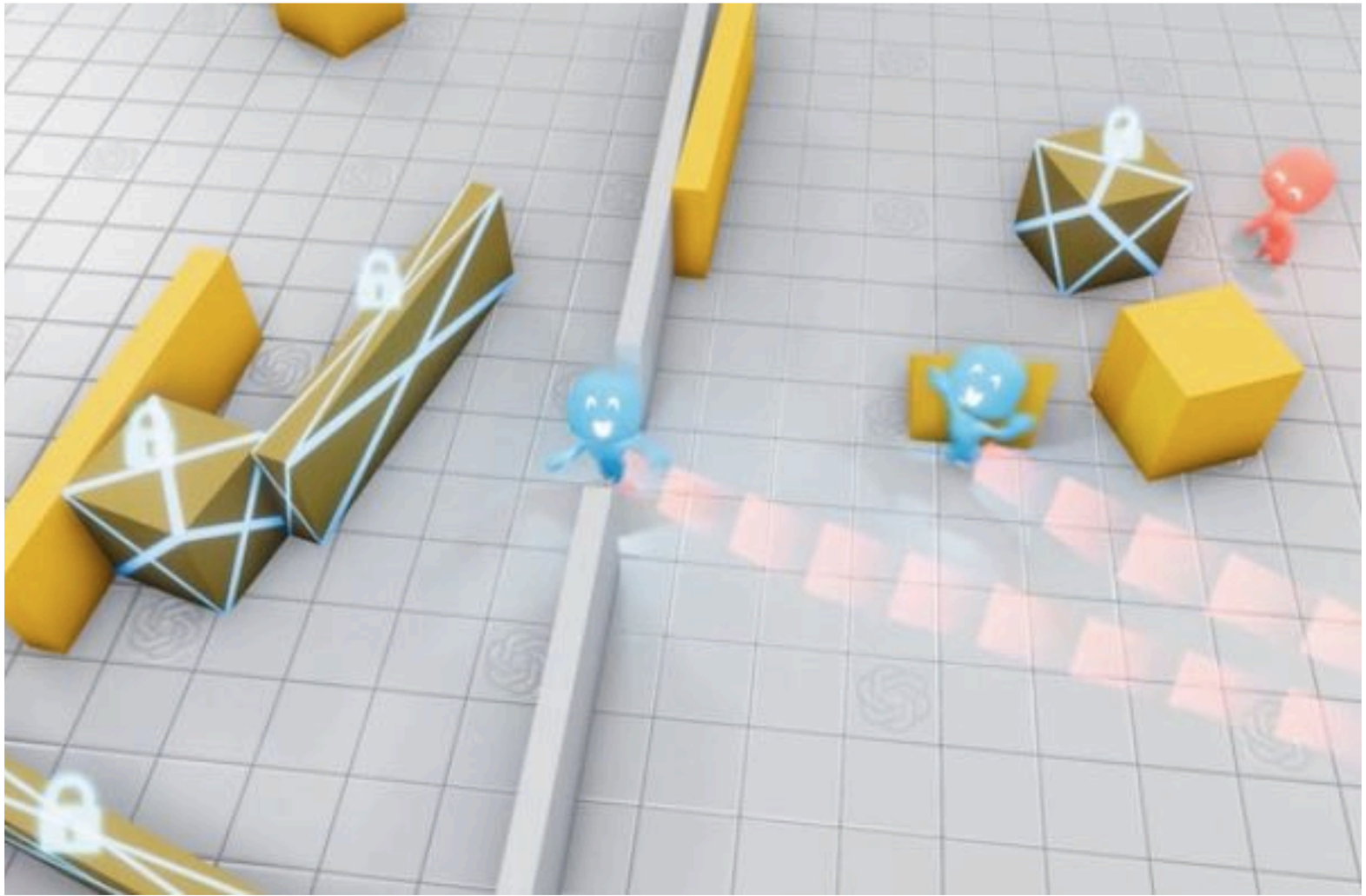


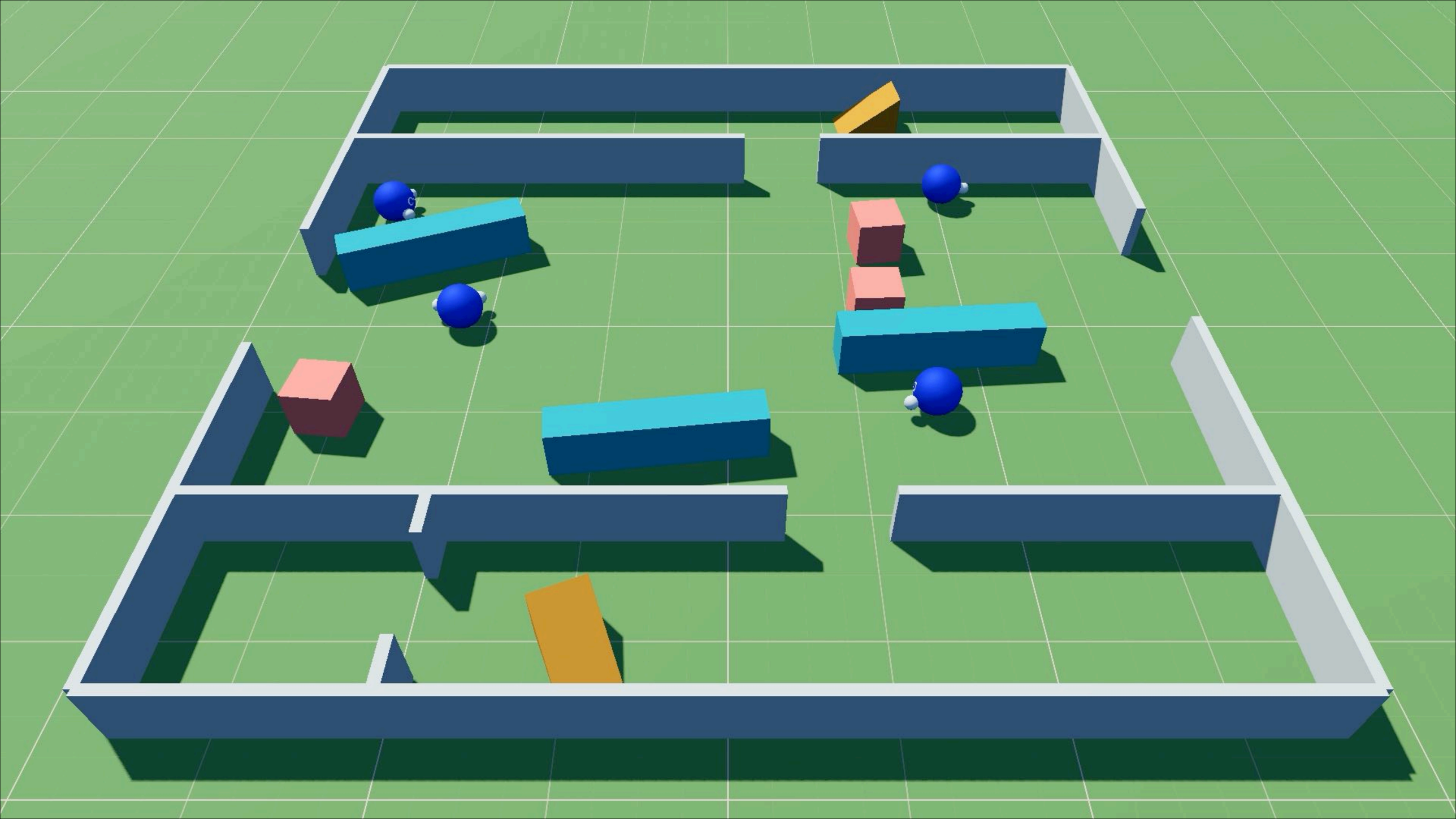
+

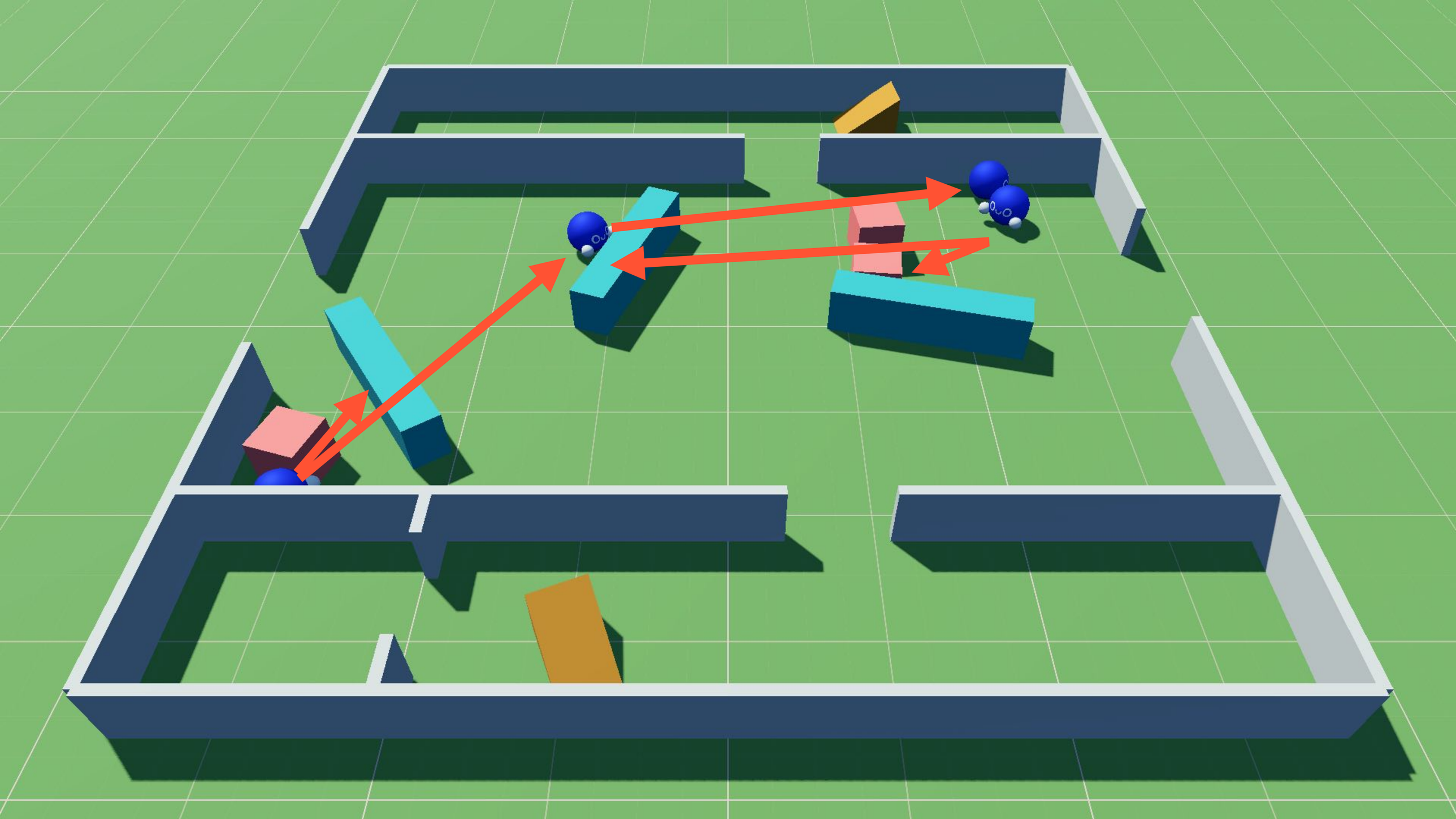
Engineering Time

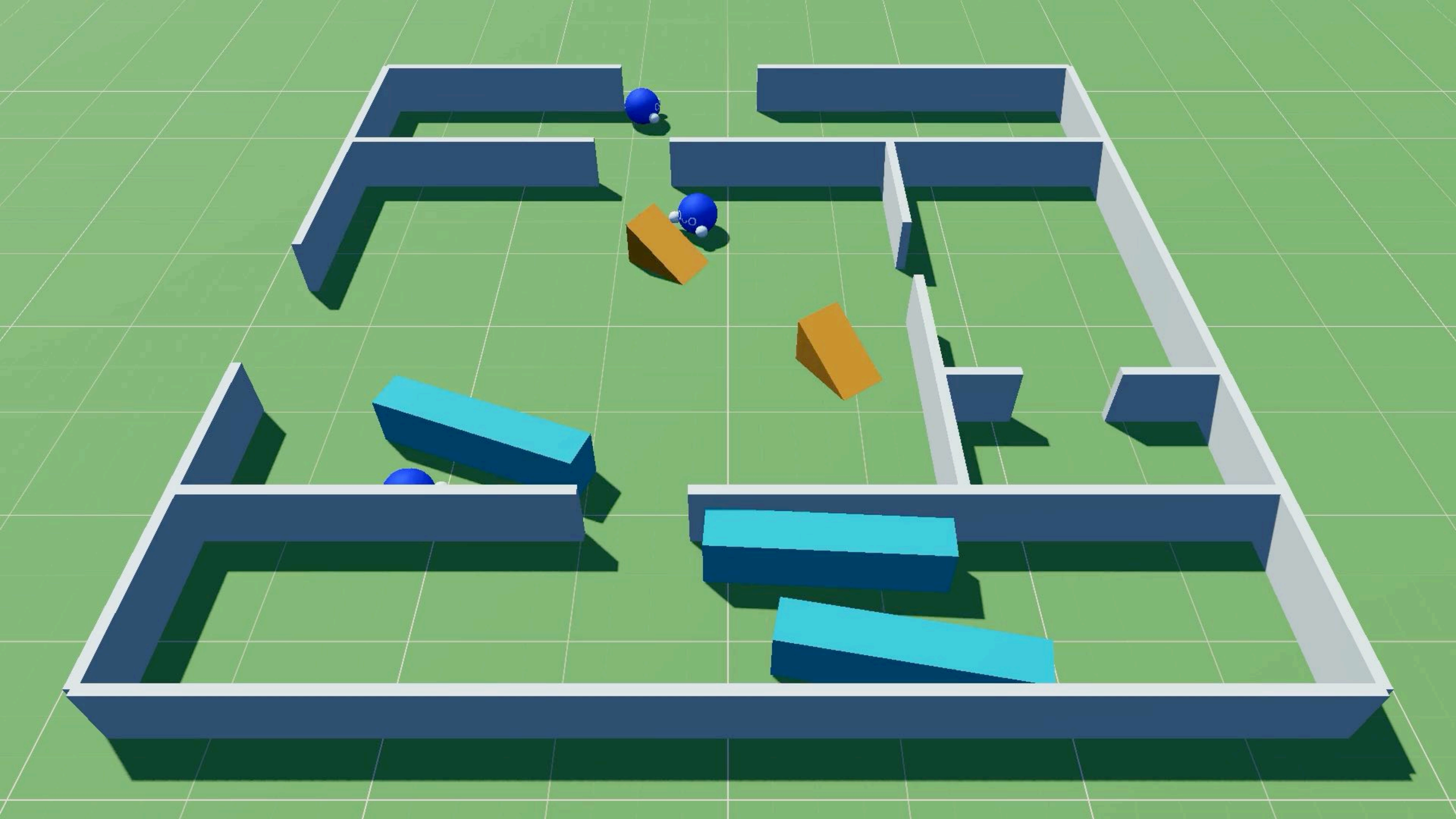


Running example: Simulating OpenAI's 3D "Hide and Seek" learning environment



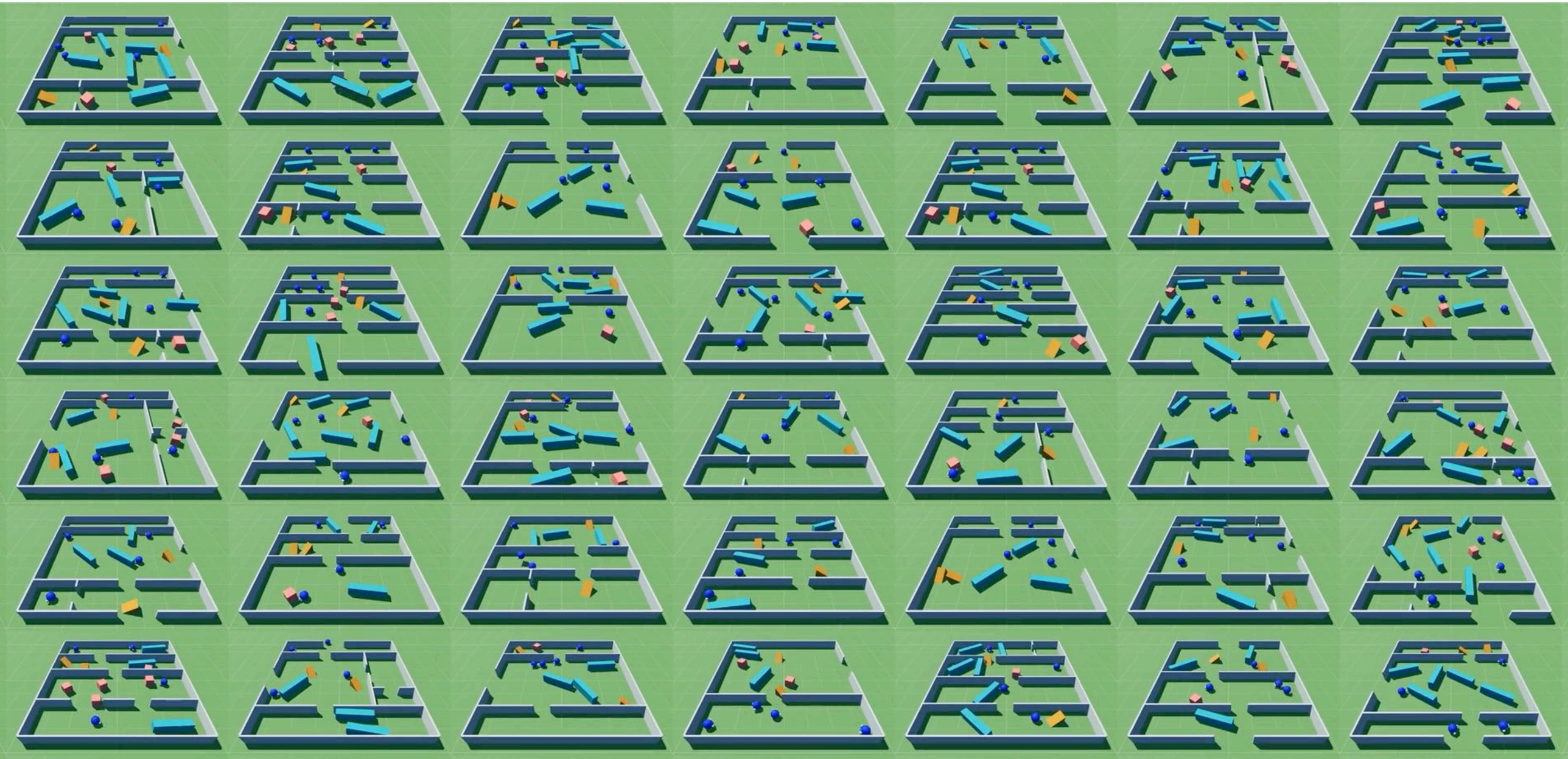




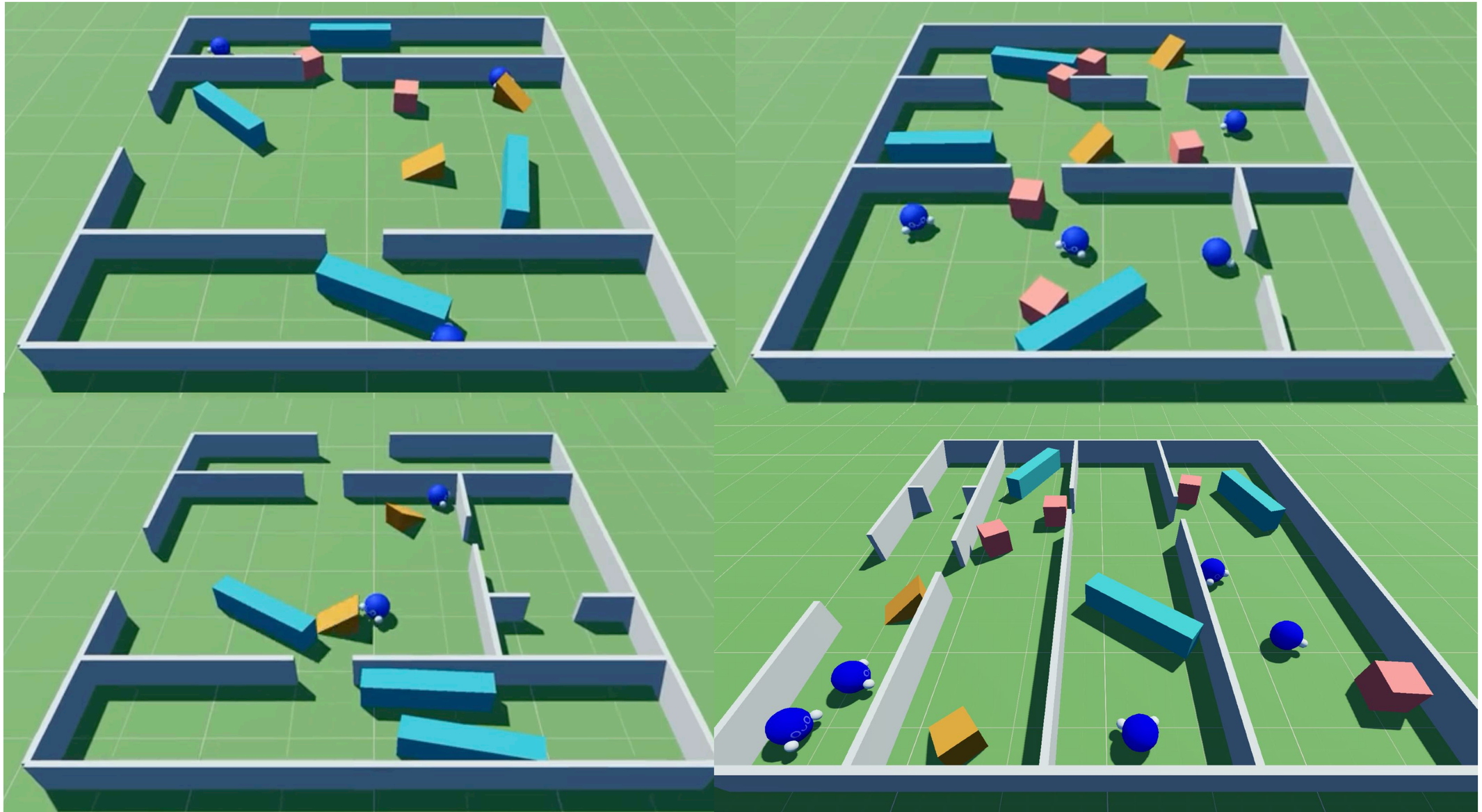


Parallel Systems Programming Requirements

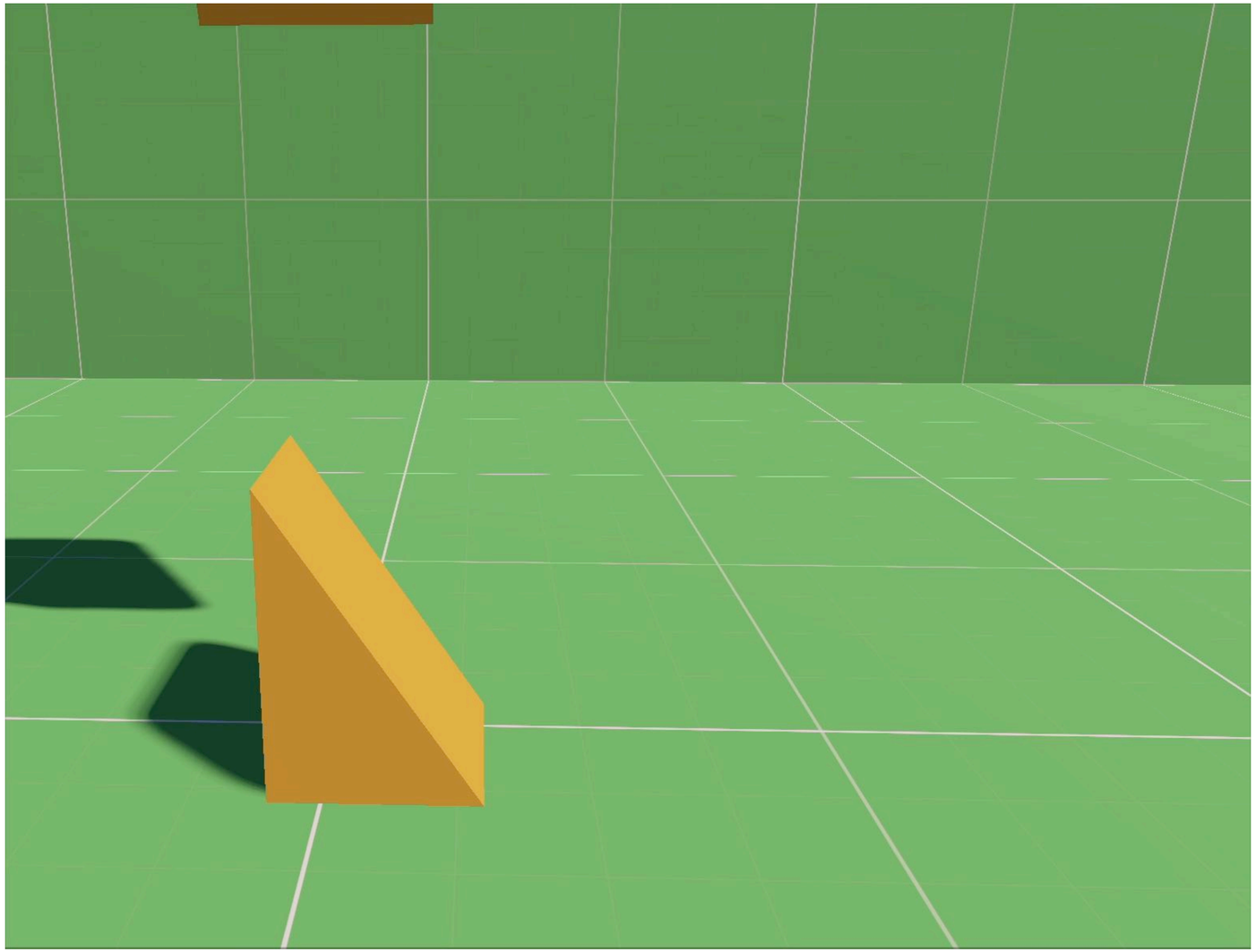
1. Nested Parallelism: Task logic for each object in each world



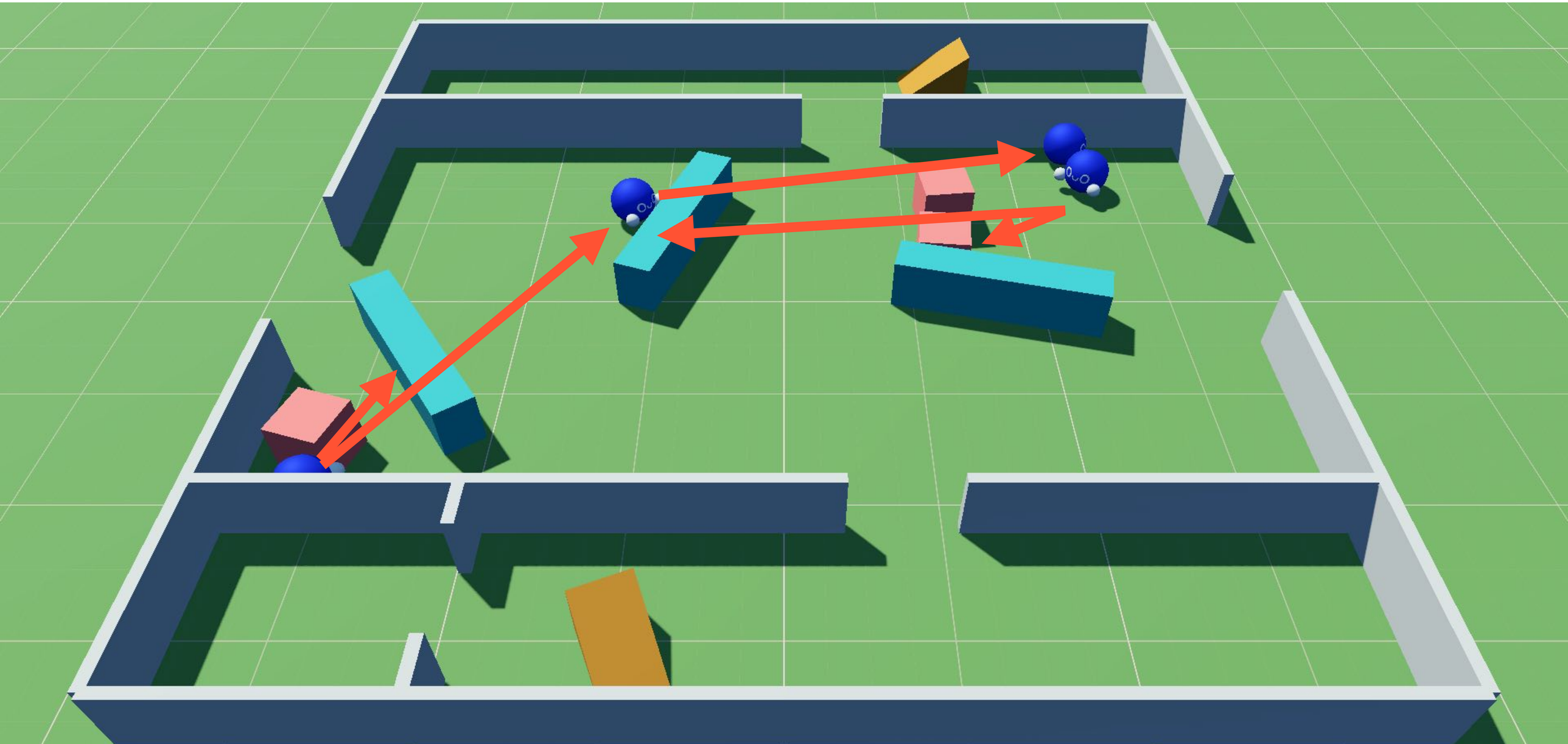
2. Irregular Nested Parallelism & Irregular Collection Sizes: Worlds with varying numbers of objects



3. Dynamic GPU-controlled memory allocation & parallelism: Physics contacts, sparse events



4. Complex Spatial Joins: Ray casting, 3D collisions



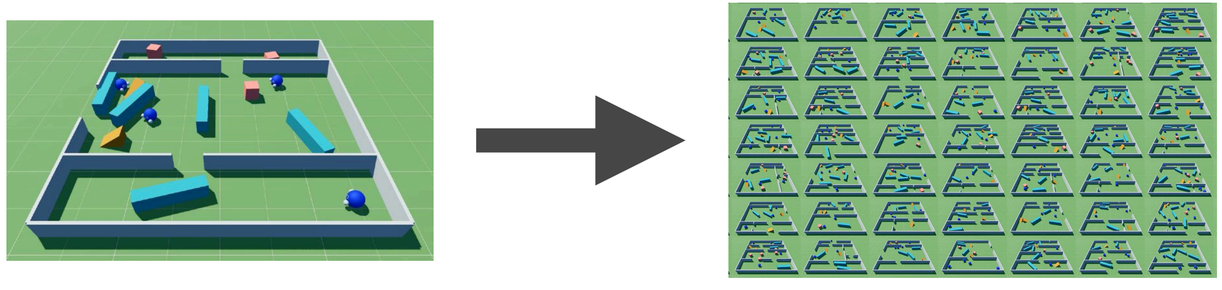
Usability: Easy scripting of task-specific logic

```
def process_action(agent_position, action):
  if action.type == MOVE_CHARACTER:
      force = computeMovementForce(action.dir)
   if action.type == LOCK_OBJECT_IN_PLACE:
      hit_obj = raycastForward(agent_position)
      if hit_obj:
        lockObject(hit_obj)
 ...
```

Usability: Easy scripting of task-specific logic

```
def process_action(agent_position, action):
  if action.type == MOVE_CHARACTER:
      force = computeMovementForce(action.dir)
   if action.type == LOCK_OBJECT_IN_PLACE:
      hit_obj = raycastForward(agent_position)
      if hit_obj:
        lockObject(hit_obj)
...
```

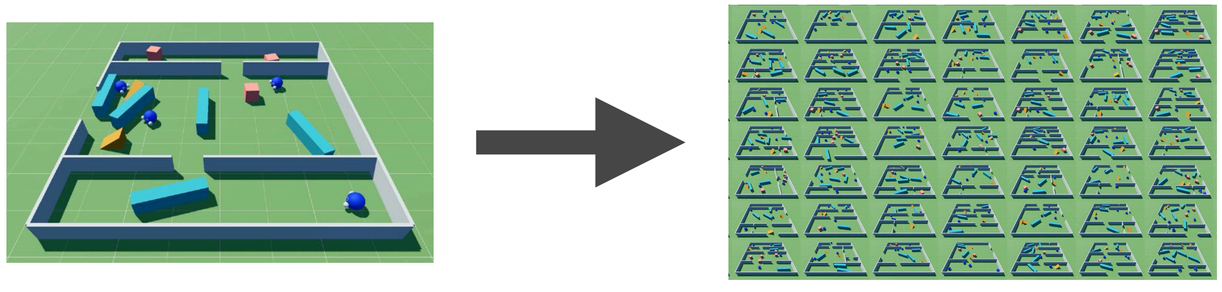
5. Implicit Parallelism: Logic written in terms of 1 environment, automatically batched



Usability: Easy scripting of task-specific logic

```
def process_action(agent_position, action):
  if action.type == MOVE_CHARACTER:
      force = computeMovementForce(action.dir)
   if action.type == LOCK_OBJECT_IN_PLACE:
      hit_obj = raycastForward(agent_position)
      if hit_obj:
        lockObject(hit_obj)
...
```

6. Convenience of standard SPMD control flow



Claim:

We need to create a "game engine" for building batch simulators that meets these requirements!

1. Nested Parallelism
2. Irregular Parallelism & Collection Sizes
3. Dynamic GPU-Controlled Allocation
4. Complex Spatial Joins
5. Implicit Parallelism
6. SPMD-Style Control Flow

What About Using Existing Systems / Frameworks to Help to Build Complex Batch Simulators?

Lowest-level option: Where does CUDA C++ fall short for our needs?

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Writing a batch simulator in raw CUDA requires custom parallel memory management & scheduling

	CUDA C++
Nested Parallelism	✓
Irregular Parallelism & Collection Sizes	X
Dynamic GPU-Controlled Allocation	X
Complex Spatial Joins	X
Implicit Parallelism	X
SPMD-style control flow	✓

Higher-level GPU kernel languages: friendlier syntax, but same abstraction as CUDA C++

NVIDIA Warp


```python
import warp as wp

@wp.kernel
def integrate(p: wp.array(dtype=wp.vec3),
              v: wp.array(dtype=wp.vec3),
              f: wp.array(dtype=wp.vec3),
              m: wp.array(dtype=float)):

    # thread id
    tid = wp.tid()

    # Semi-implicit Euler step
    v[tid] = v[tid] + (f[tid] * m[tid] + wp.vec3(0.0, -9.8, 0.0)) * dt
    x[tid] = x[tid] + v[tid] * dt

# kernel launch
wp.launch(integrate, dim=1024, inputs=[x, v, f, …], device="cuda:0")
```

Array-based programming: Describe simulation in terms of bulk operations on fixed-size tensors

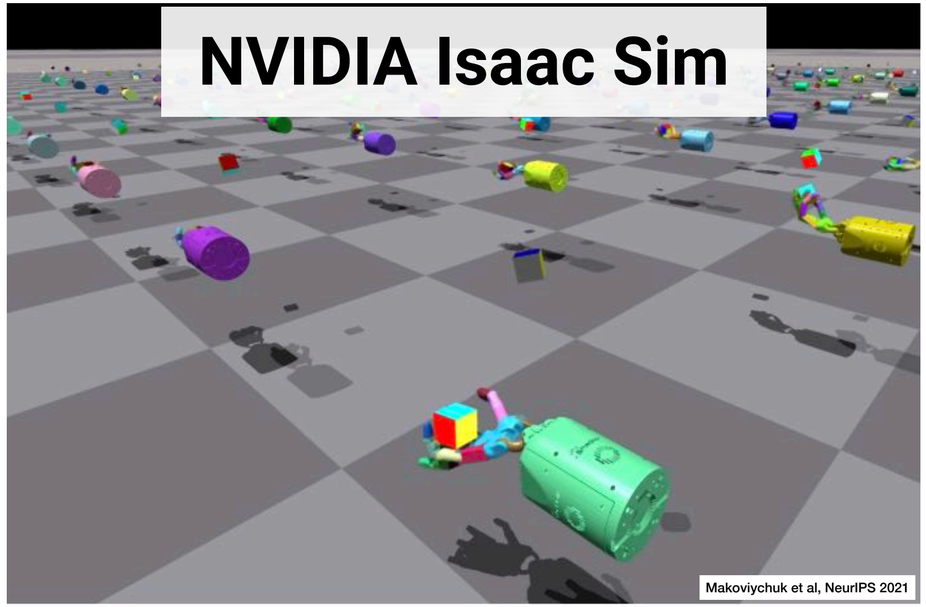
PyTorch

Array-based programming: Variable environment structures, procedural generation are challenging

	PyTorch	
Nested Parallelism	✓	✓
Irregular Parallelism & Collection Sizes	~	X
Dynamic GPU-Controlled Allocation	X	X
Complex Spatial Joins	X	X
Implicit Parallelism	~	✓
SPMD-style control flow	X	X

What about building batch simulators by reusing existing GPU simulation libraries?

NVIDIA PHYSX



Existing GPU simulation libraries (PhysX) are designed to accelerate a CPU-controlled simulation

	GPU PhysX
Nested Parallelism	✓
Irregular Parallelism & Collection Sizes	~
Dynamic GPU-Controlled Allocation	X
Complex Spatial Joins	✓
Implicit Parallelism	X
SPMD-style control flow	X

Existing Systems Do Not Meet the Requirements!

	PyTorch	JAX	CUDA C++	Warp / NUMBA	GPU PhysX
Nested Parallelism	✓	✓	✓	✓	✓
Irregular Parallelism & Collection Sizes	~	X	X	X	~
Dynamic GPU-Controlled Allocation	X	X	X	X	X
Complex Spatial Joins	X	X	X	X	✓
Implicit Parallelism	~	✓	X	X	X
SPMD-style control flow	X	X	✓	✓	X

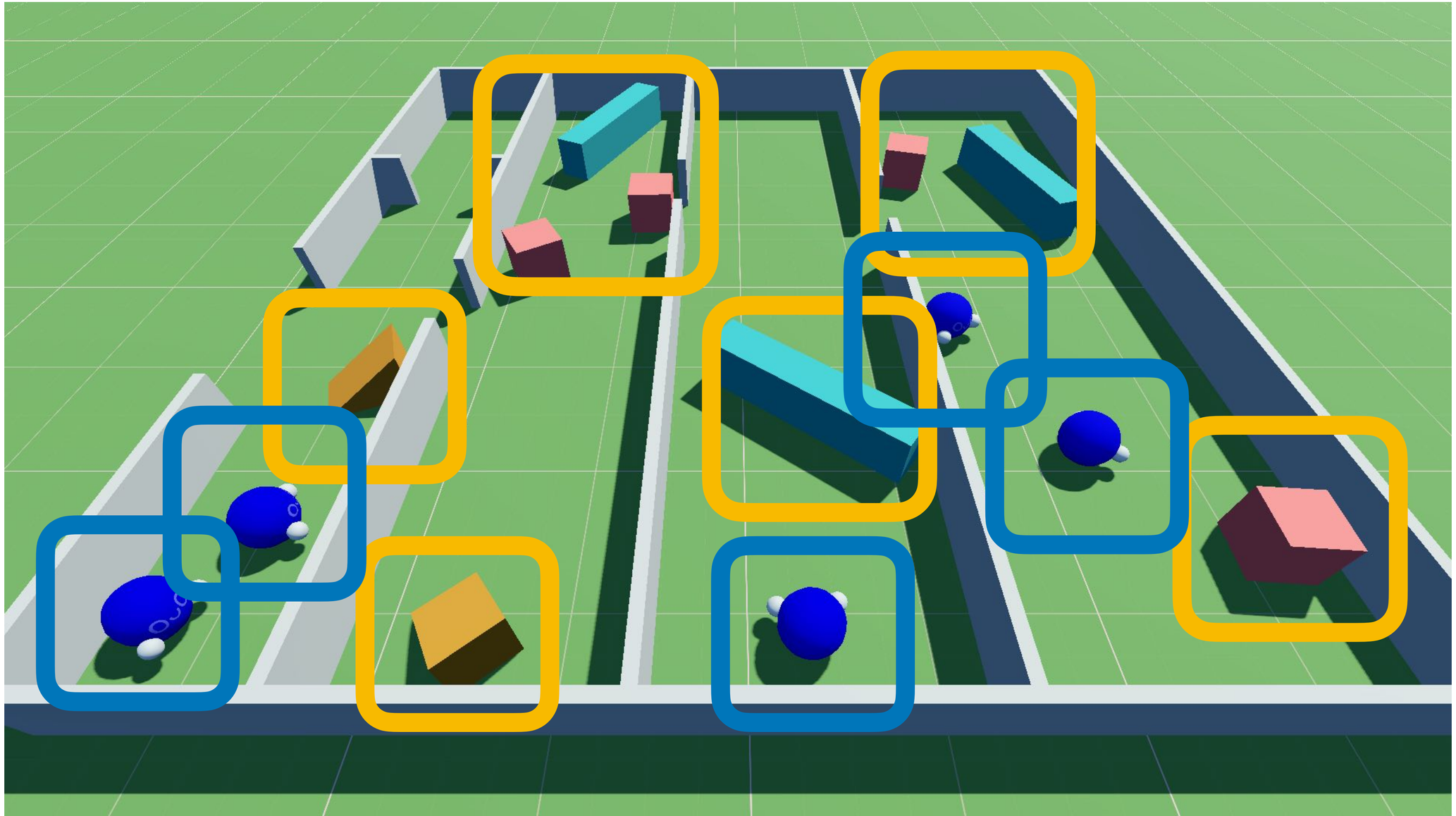
Tonight's reading

- Madrona engine project (Stanford project)

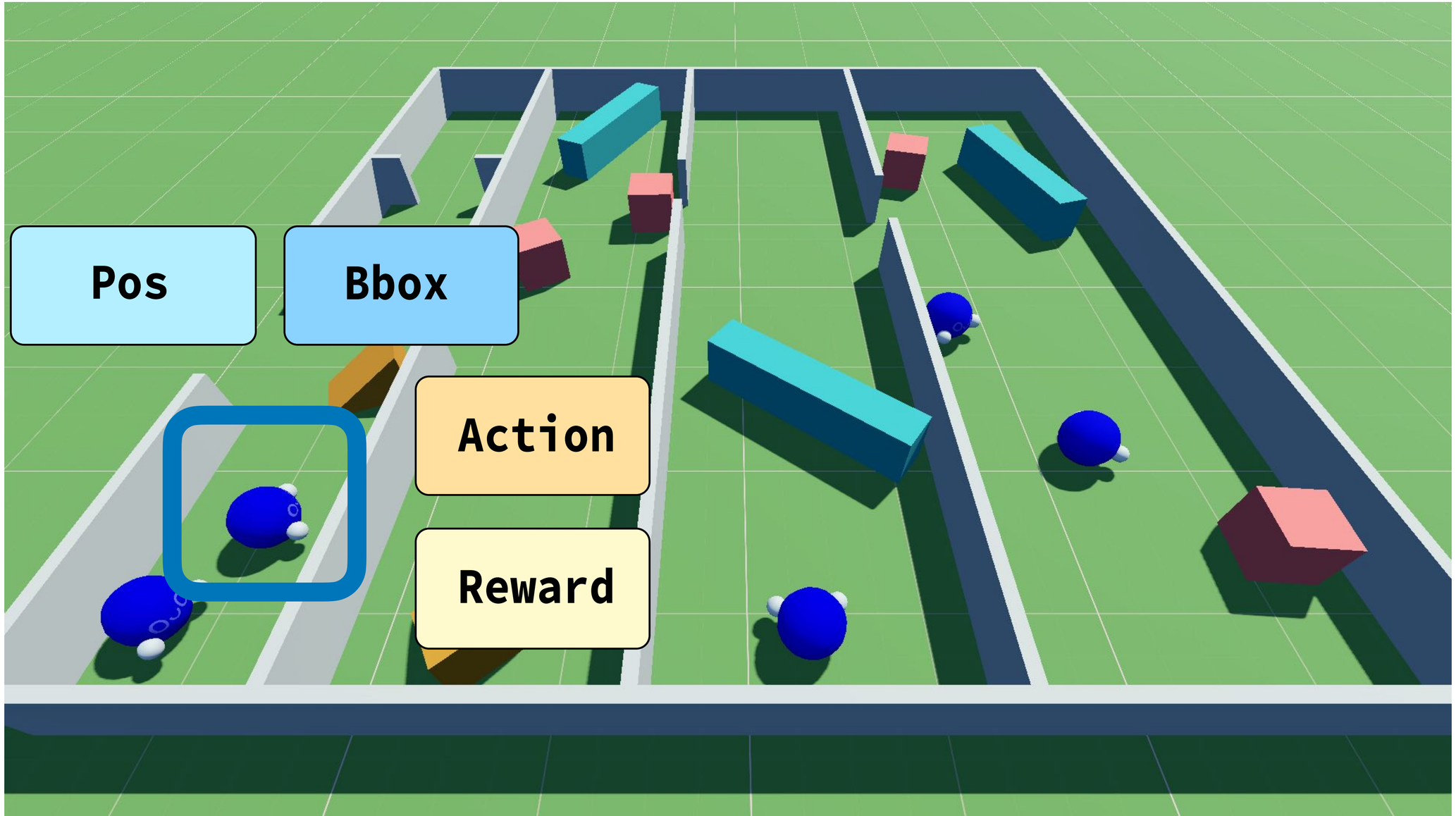
- A framework for building GPU batch simulators using entity component system (ECS) design patterns

Tutorial:
Entity Component System (ECS) Design Patterns on the GPU

Concept 1: Entities (ECS)



Concept 2: Components (ECS)



Concept 2: Components (ECS)

Agents

Id	EnvID	Pos	Bbox	Action	Reward	
12	0	[0,0,.5]	{min...	LEFT	0.1	...
32	0	[2,1,0]	{min...	FWD	-0.1	...
51	0	[1.5,0,1]	{min...	FWD	-2.5	...

Obstacles

Id	EnvID	Pos	Bbox	
13	0	[0.5,0,.5]	{min...	...
72	0	[0,1,3]	{min...	...
61	0	[1,1,2]	{min...	...

Batch simulation ECS: Store data across all environments in unified tables in GPU memory

Agents

Id	EnvID	Pos	Bbox	Action	Reward	
12	0	[0,0,.5]	{min...	LEFT	0.1	...
32	0	[2,1,0]	{min...	FWD	-0.1	...
51	0	[1.5,0,1]	{min...	FWD	-2.5	...
62	1	[1.5,0.5,1]	{min...	RIGHT	0.3	...
65	1	[-0.5,0,0]	{min...	RIGHT	0.5	...
20	2	[0.5,1,1]	{min...	LEFT	1.5	...
23	2	[-1.5,0,0]	{min...	LEFT	10.5	...
41	2	[0.5,1,0]	{min...	BACK	-10	...

Obstacles

Id	EnvID	Pos	Bbox	
13	0	[0.5,0,.5]	{min...	...
72	0	[0,1,3]	{min...	...
61	0	[1,1,2]	{min...	...
49	1	[0.5,1,2]	{min...	...
70	1	[-0.5,1,0]	{min...	...
33	2	[1.5,0,2.5]	{min...	...
81	3	[2.5,0.5,2]	{min...	...
11	3	[1.5,0.5,2]	{min...	...

Unified table storage also enables throughput-oriented dynamic memory allocation

Agents

Id	EnvID	Pos	Bbox	Action	Reward	
12	0	[0,0,.5]	{min...	LEFT	0.1	...
32	0	[2,1,0]	{min...	FWD	-0.1	...
~~51~~	~~X~~	~~[1.5,0,1]~~	~~{min...~~	~~FWD~~	~~-2.5~~	~~...~~
62	1	[1.5,0.5,1]	{min...	RIGHT	0.3	...
65	1	[-0.5,0,0]	{min...	RIGHT	0.5	...
20	2	[0.5,1,1]	{min...	LEFT	1.5	...
23	2	[-1.5,0,0]	{min...	LEFT	10.5	...
41	2	[0.5,1,0]	{min...	BACK	-10	...
51	1	[0.5,1,1]	{min...	BACK	-10	...

Obstacles

Id	EnvID	Pos	Bbox	
13	0	[0.5,0,.5]	{min...	...
72	0	[0,1,3]	{min...	...
61	0	[1,1,2]	{min...	...
~~49~~	~~X~~	~~[0.5,1,2]~~	~~{min...~~	~~...~~
70	1	[-0.5,1,0]	{min...	...
33	2	[1.5,0,2.5]	{min...	...
81	3	[2.5,0.5,2]	{min...	...
~~11~~	~~X~~	~~[1.5,0.5,2]~~	~~{min...~~	~~...~~
15	2	[1.5,1.5,2]	{min...	...
12	0	[2.5,0.5,3]	{min...	...

Concept 3: Systems (ECS)

Agents

Id	EnvID	Pos	Bbox	Action	Reward	
12	0	[0,0,.5]	{min...	LEFT	0.1	...
32	1	[2,1,0]	{min...	FWD	-0.1	...
51	2	[1.5,0,1]	{min...	FWD	2.5	...
22	2	[2.5,0,1.5]	{min...	RIGHT	1.1	...

Obstacles

Id	EnvID	Pos	Bbox	
13	0	[0.5,0,.5]	{min...	...
72	1	[0,1,3]	{min...	...
61	1	[1,1,2]	{min...	...
25	2	[1.5,1,2.5]	{min...	...

ProcessActions

Pos, Action

Collisions

Id, Pos, Bbox

ComputeRewards

Pos, Reward

Concept 3: Systems (ECS)

Agents

Id	EnvID	Pos	Bbox	Action	Reward	
12	0	[0,0,.5]	{min...	LEFT	0.1	...
32	1	[2,1,0]	{min...	FWD	-0.1	...
51	2	[1.5,0,1]	{min...	FWD	2.5	...
22	2	[2.5,0,1.5]	{min...	RIGHT	1.1	...

Obstacles

Id	EnvID	Pos	Bbox	
13	0	[0.5,0,.5]	{min...	...
72	1	[0,1,3]	{min...	...
61	1	[1,1,2]	{min...	...
25	2	[1.5,1,2.5]	{min...	...

ProcessActions

Pos, Action

Collisions

Id, Pos, Bbox

ComputeRewards

Pos, Reward

Systems written as straight-line, per-entity logic

Agents

Id	EnvID	Pos	Bbox	Action	Reward
12	0	[0,0,.5]	{min...	LEFT	0.1
32	1	[2,1,0]	{min...	FWD	-0.1
51	2	[1.5,0,1]	{min...	FWD	2.5
22	2	[2.5,0,1.5]	{min...	RIGHT	1.1

```
def process_action(agent_position, action):
  if action.type == MOVE:
      force = computeMovementForce(action.dir)
   if action.type == LOCK:
      hit_obj = raycastForward(agent_position)
      if hit_obj:
        lockObject(hit_obj)

...
```


ProcessActions

Pos, Action

Collisions

Id, Pos, Bbox

ComputeRewards

Pos, Reward

Parallel GPU threads execute system logic over each table row

Agents

Id	EnvID	Pos	Bbox	Action	Reward
GPU Thread	0	[0,0,.5]	{min...	LEFT	0.1
GPU Thread	1	[2,1,0]	{min...	FWD	-0.1
GPU Thread	2	[1.5,0,1]	{min...	FWD	2.5
GPU Thread	2	[2.5,0,1.5]	{min...	RIGHT	1.1

```
def process_action(agent_position, action):
  if action.type == MOVE:
      force = computeMovementForce(action.dir)
   if action.type == LOCK:
      hit_obj = raycastForward(agent_position)
      if hit_obj:
        lockObject(hit_obj)

...
```


ProcessActions

Pos, Action

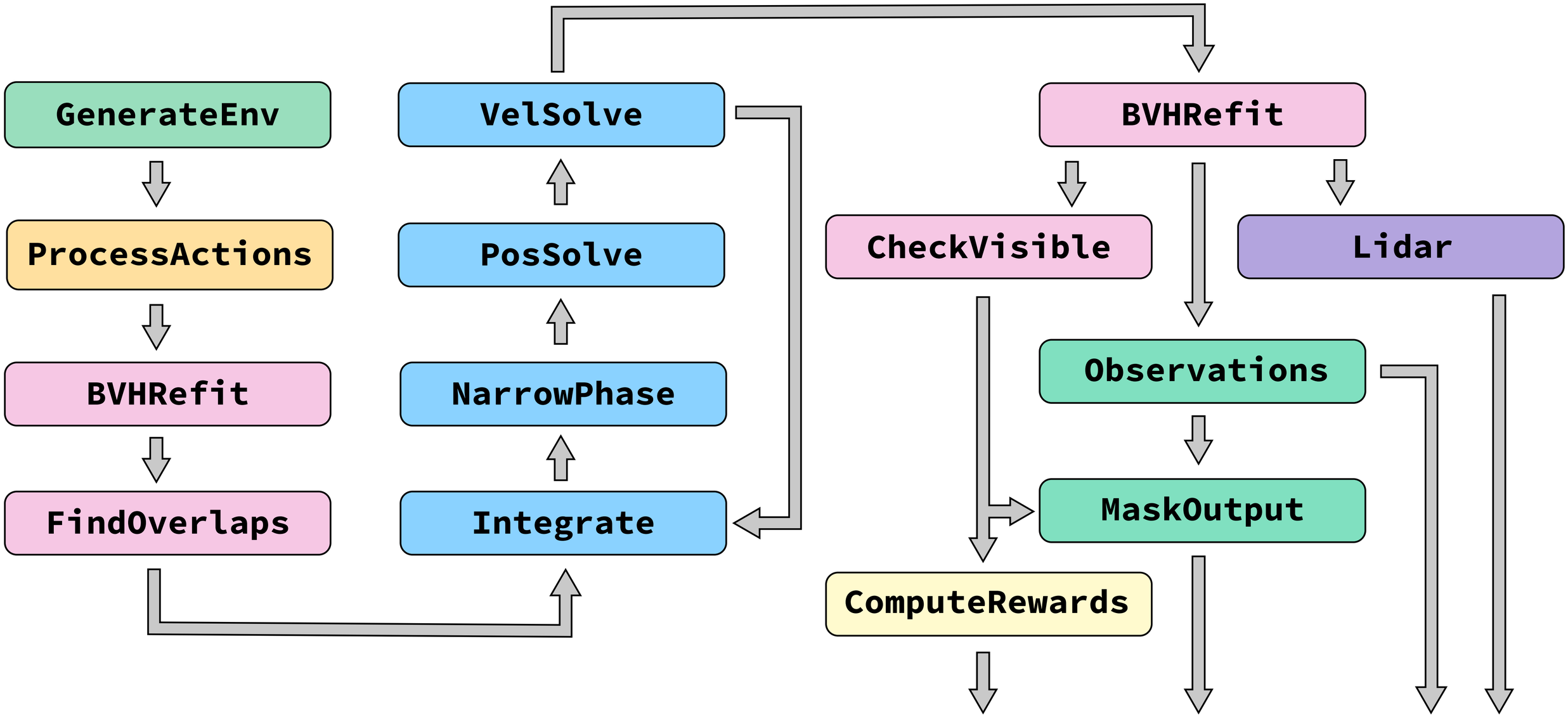
Collisions

Id, Pos, Bbox

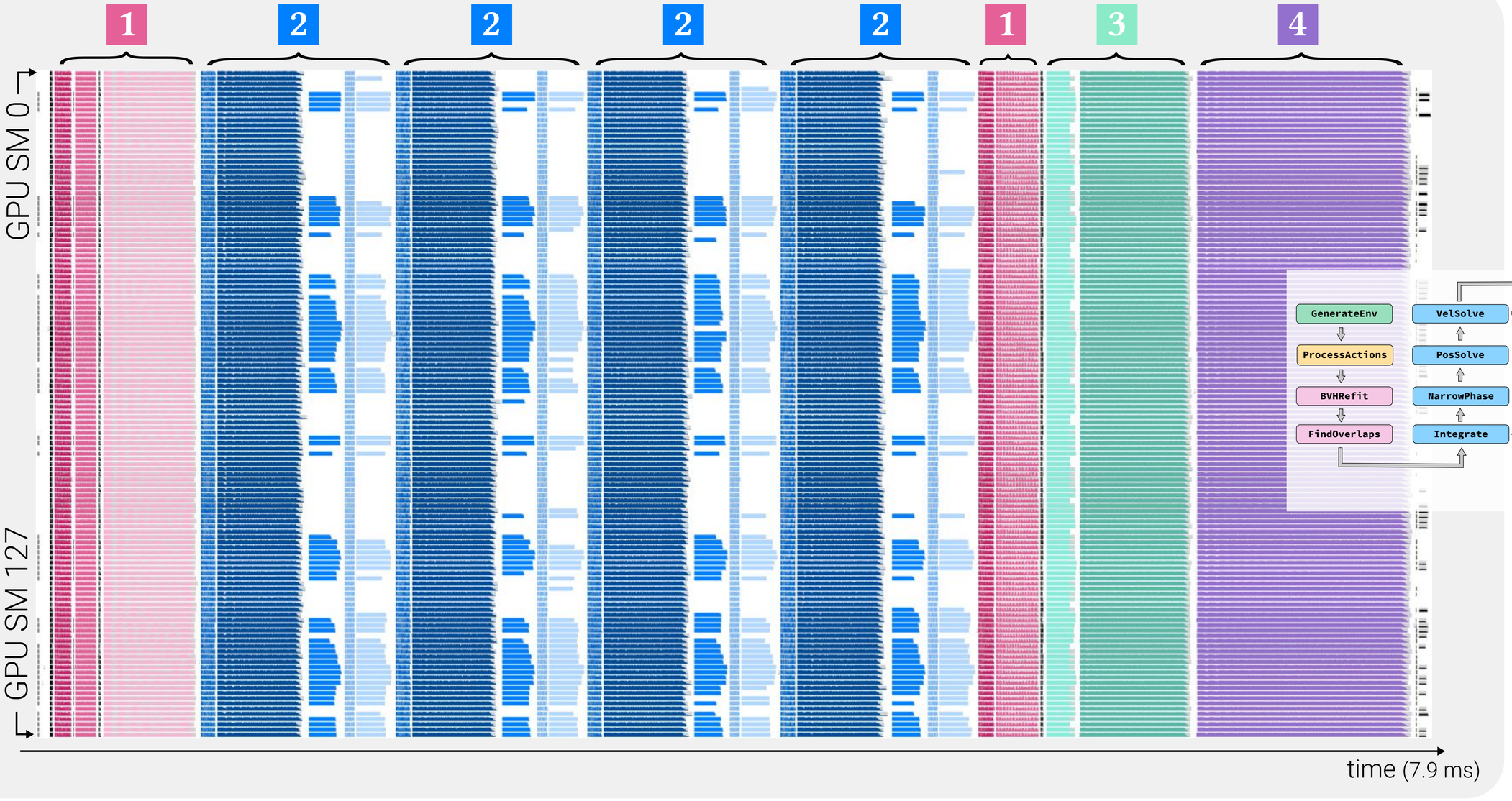
ComputeRewards

Pos, Reward

ECS systems combined into task graph and executed in parallel on the GPU



Scheduling batch of worlds onto GPU



1 BVH & Broad Phase 2 Physics Sub-Step 3 Agent Observations & Rewards 4 LIDAR

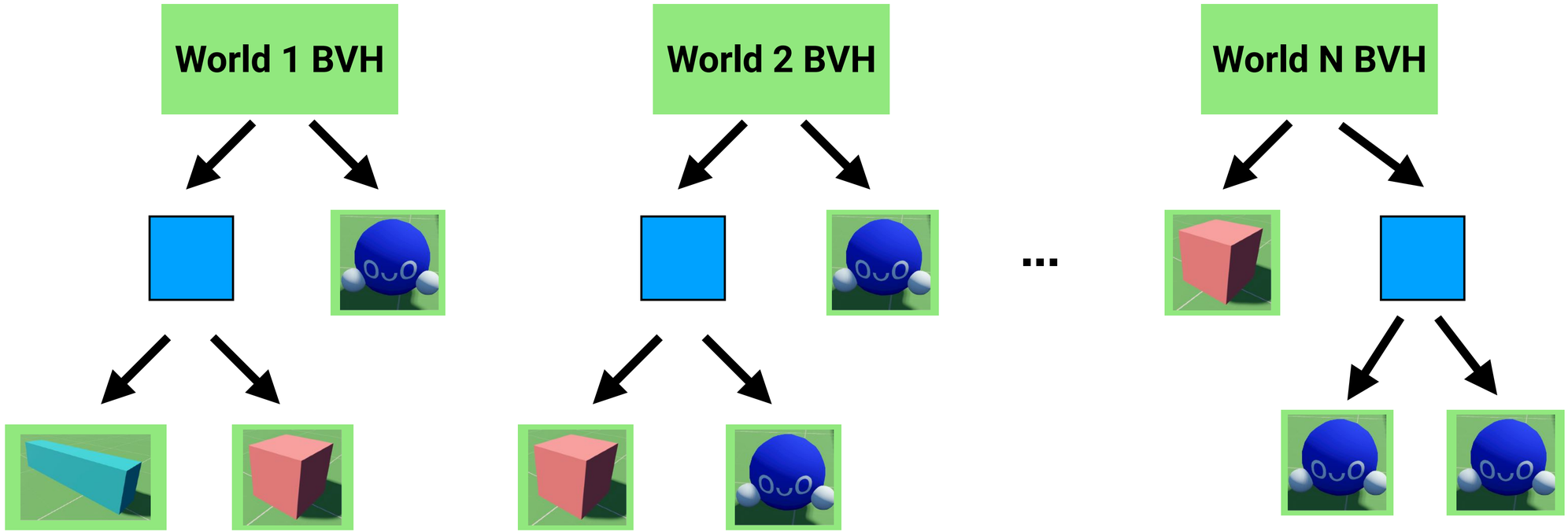
What about spatial queries?

```
def process_action(agent_position, action):
  if action.type == MOVE:
        force = computeMovementForce(action.dir)
    if action.type == LOCK:
        hit_obj = raycastForward(agent_position)
        if hit_obj:
          lockObject(hit_obj)

  ...
```

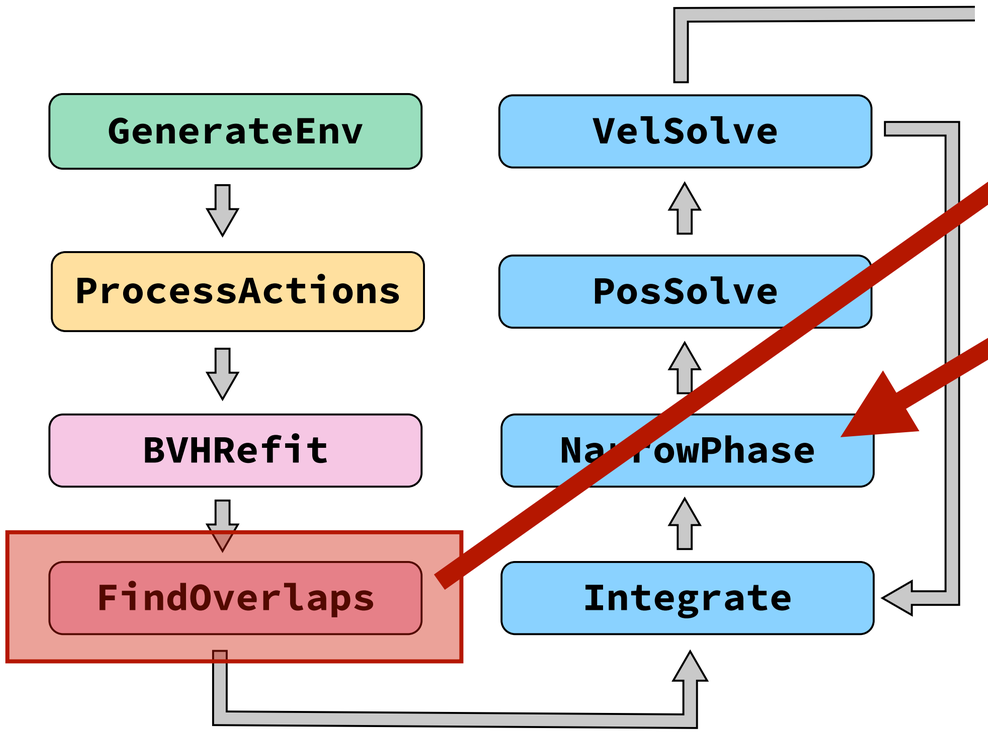
Agents

Id	EnvID	Pos	Bbox	Action	Reward	
12	0	[0,0,.5]	{min...	LEFT	0.1	...
32	1	[2,1,0]	{min...	FWD	-0.1	...
51	2	[1.5,0,1]	{min...	FWD	2.5	...
22	2	[2.5,0,1.5]	{min...	RIGHT	1.1	...

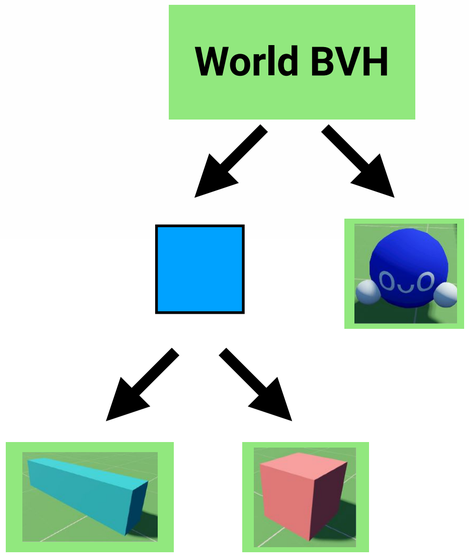
Madrona standard library provides high-performance per-world 3D acceleration structure (BVH)



BVH Standard Library Calls Allow ECS Systems to Make Spatial Queries

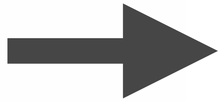



```
def find_overlaps(world, my_id, position, bbox):
  for bvh_node in world.bvh.find_overlapping(bbox)
    world.createEntity(CollisionPair(my_id, bvh_node.id))
```



Madrona needs a straightforward imperative language for authoring ECS systems

```
def process_action(world,
                   agent_position,
                   action):
 if action.type == MOVE:
       force = computeMovementForce(action.dir)
   if action.type == LOCK:
       hit_obj =
         raycastForward(world, agent_position)
       if hit_obj:
         lockObject(hit_obj)

 ...
```



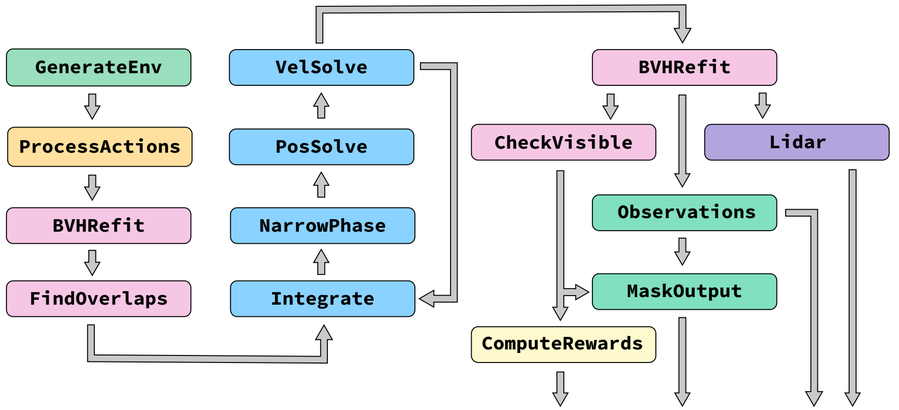
```
void process_action(World &world,
                    Position &position,
                    AgentAction &action) {
  if (action.type == MOVE) {
        Vector3 force =
            computeMovementForce(action.dir);
    }
    if (action.type == LOCK) {
        Entity hit_obj =
          raycastForward(world, agent_position);
        if (hit_obj) {
          lockObject(hit_obj);
        }
    }

  ...
}
```

Madrona Framework Summary

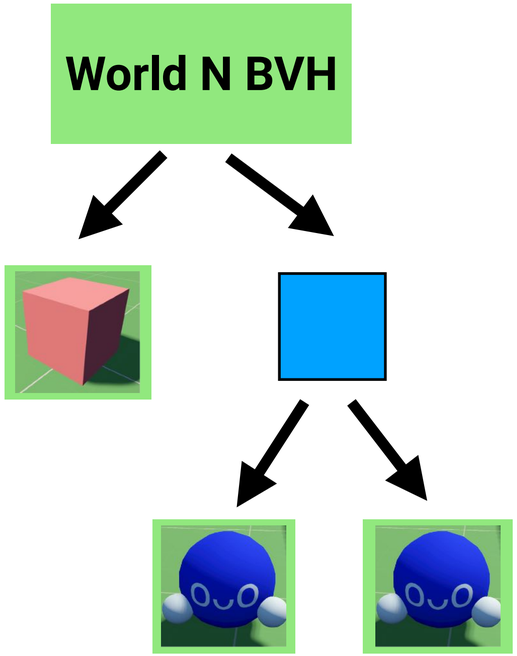
ECS Storage

Agents						
Id	EnvID	Pos	Bbox	Action	Reward	
12	0	[0,0,.5]	{min...	LEFT	0.1	...
32	1	[2,1,0]	{min...	FWD	-0.1	...
51	2	[1.5,0,1]	{min...	FWD	2.5	...
22	2	[2.5,0,1.5]	{min...	RIGHT	1.1	...

Parallel ECS Scheduler for GPU



ECS-Integrated Standard Library (BVH, Physics, etc)



Summary: Madrona meets core requirements for building wide range of batch simulators

		Madrona
Irregularly Sized Collections	Multi-World ECS Tables	✓
Dynamic GPU-Controlled Allocation		✓
Irregular Nested Parallelism	ECS Systems & Parallel GPU Task Graph Scheduling	✓
Dynamic GPU-Controlled Parallelism		✓
Implicit Parallelism		✓
Complex Spatial Joins	Built on General-Purpose Programming Language (CUDA C++)	✓
SPMD-Style Control Flow		✓

An Alternative approach: Generative AI as a means to generate world simulation output

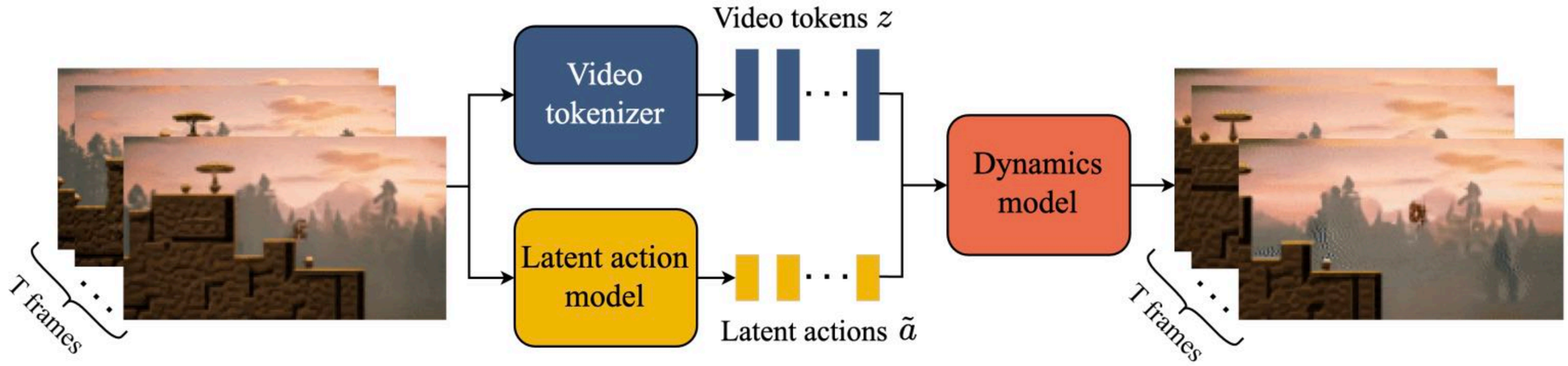
(previewing parts of next week)

Enhancing CG images to look like real-world images using image-to-image transfer

Modifying real-world images to create novel situations

Genie

- Key idea: learn a world simulator from videos of video game play
  - From video, learn latency user actions, and dynamics model that steps work given (current state, action)



  - Then at "test time" given a novel world state (perhaps one generated from a prompt), and given user input, time step the novel world forward in time