Lecture 18:

# Rendering/Simulation for Model Training (+ intro to shading languages)

---

Visual Computing Systems
Stanford CS348K, Spring 2022

# Today

- We've talked about how ML/AI techniques are used to improve visual computing applications: computational photography, rendering, video compression, etc…
- Today we'll talk about how rendering/simulation are increasing being used to train better ML models.
- At the end we'll talk about a few GPU architecture issues that will be relevant to Tuesday's discussion of GPU programming languages

# Think back to earlier in course

## What was the biggest practical bottleneck to training good models?

### Snorkel: Rapid Training Data Creation with Weak Supervision

Alexander Ratner Stephen H. Bach Henry Ehrenberg
Jason Fries Sen Wu Christopher Ré
Stanford University
Stanford, CA, USA
{ajratner, bach, henryre, jfries, senwu, chrismre}@cs.stanford.edu

**ABSTRACT**

Labeling training data is increasingly the largest bottleneck in deploying machine learning systems. We present Snorkel, a first-of-its-kind system that enables users to train state-of-the-art models without hand labeling any training data. Instead, users write labeling functions that express arbitrary heuristics, which can have unknown accuracies and correlations. Snorkel denoises their outputs without access to ground truth by incorporating the first end-to-end implementation of our recently proposed machine learning paradigm, data programming. We present a flexible interface layer for writing labeling functions based on our experience over the past year collaborating with companies, agencies, and research labs. In a user study, subject matter experts build models 2.8× faster and increase predictive performance an average 45.5% versus seven hours of hand labeling. We study the modeling tradeoffs in this new setting and propose an optimizer for automating tradeoff decisions that gives up to 1.8× speedup per pipeline execution. In two collaborations, with the U.S. Department of Veterans Affairs and the U.S. Food and Drug Administration, and on four open-source text and image data sets representative of other deployments, Snorkel provides 132% average

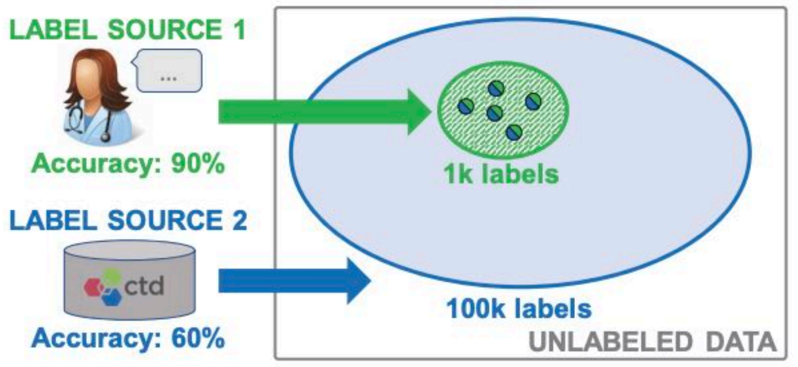



**Figure 1: In Example 1.1, training data is labeled by sources of differing accuracy and coverage. Two key challenges arise in using this weak supervision effectively. First, we need a way to estimate the unknown source accuracies to resolve disagreements. Second, we need to pass on this critical lineage information to the end model being trained.**

advent of *deep learning* techniques, which can learn task-specific representations of input data, obviating what used to be the most time-consuming development task: feature engineering. These learned representations are particularly effective for tasks like natural language processing and image

### Overton: A Data System for Monitoring and Improving Machine-Learned Products

Christopher Ré Apple
Feng Niu Apple
Pallavi Gudipati Apple
Charles Srisuwananukorn Apple

September 13, 2019



**Abstract**

We describe a system called Overton, whose main design goal is to support engineers in building, monitoring, and improving production machine learning systems. Key challenges engineers face are monitoring fine-grained quality, diagnosing errors in sophisticated applications, and handling contradictory or incomplete supervision data. Overton automates the life cycle of model construction, deployment, and monitoring by providing a set of novel high-level, declarative abstractions. Overton's vision is to shift developers to these higher-level tasks instead of lower-level machine learning tasks. In fact, using Overton, engineers can build deep-learning-based applications without writing any code in frameworks like TensorFlow. For over a year, Overton has been used in production to support multiple applications in both near-real-time applications and back-of-house processing. In that time, Overton-based applications have answered billions of queries in multiple languages and processed trillions of records reducing errors 1.7 − 2.9× versus production systems.

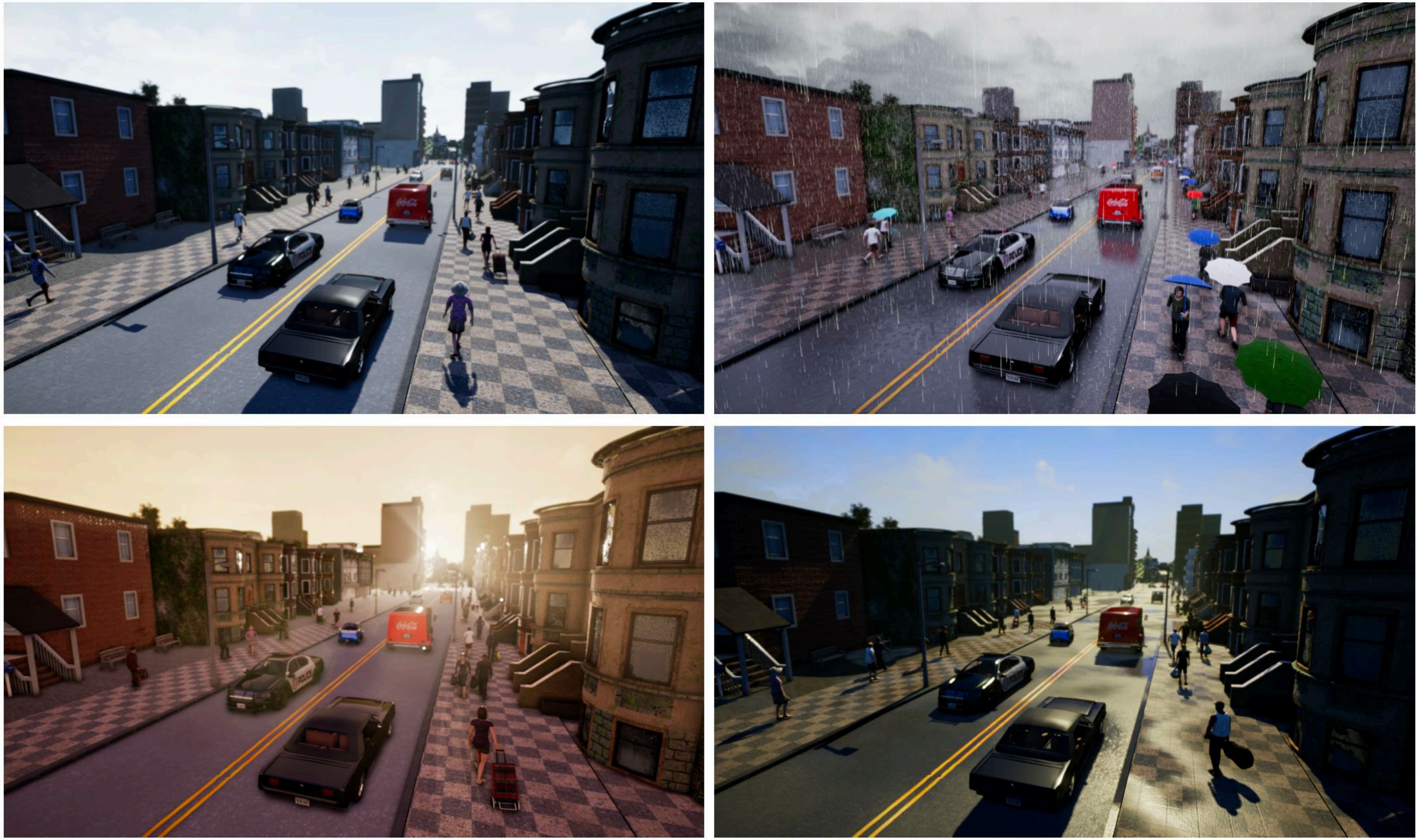
#### 1 Introduction

In the life cycle of many production machine-learning applications, maintaining and improving deployed models is the dominant factor in their total cost and effectiveness–much greater than the cost of *de novo* model construction. Yet, there is little tooling for model life-cycle support. For such applications, a key task for supporting engineers is to improve and maintain the quality in the face of changes to the input distribution and new production features. This work describes a new style of data management system called Overton that provides abstractions to support the model life cycle by helping build models, manage supervision, and monitor application quality.[1]

Overton is used in both near-real-time and backend production applications. However, for concreteness, our running example is a product that answers factoid queries, such as *"how tall is the president of the united states?"* In our experience, the engineers who maintain such machine learning products face several challenges on which they spend the bulk of their time.

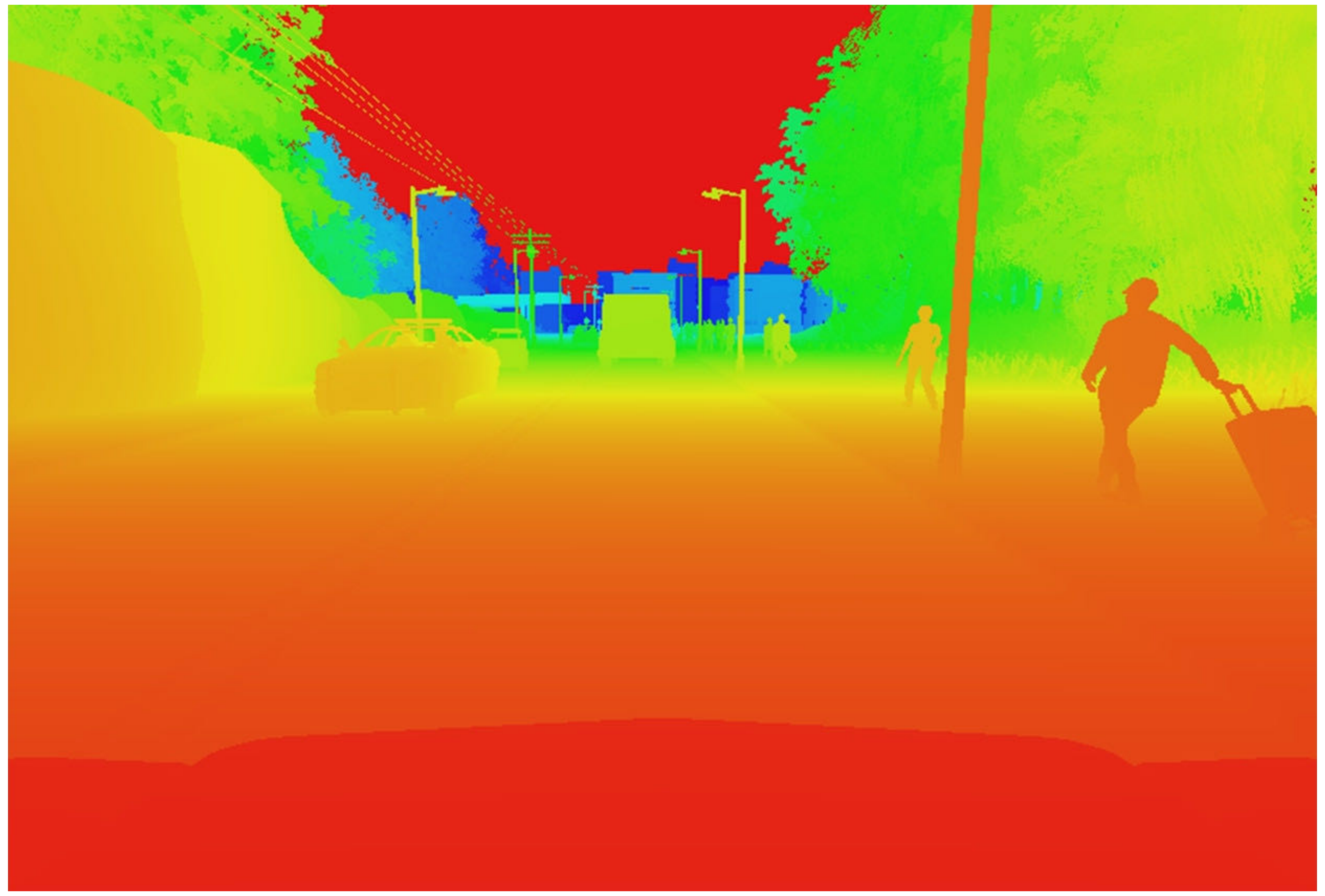
# Using advanced rendering/simulation to generate supervision to train better models

# Carla: urban driving simulator based on Unreal Engine

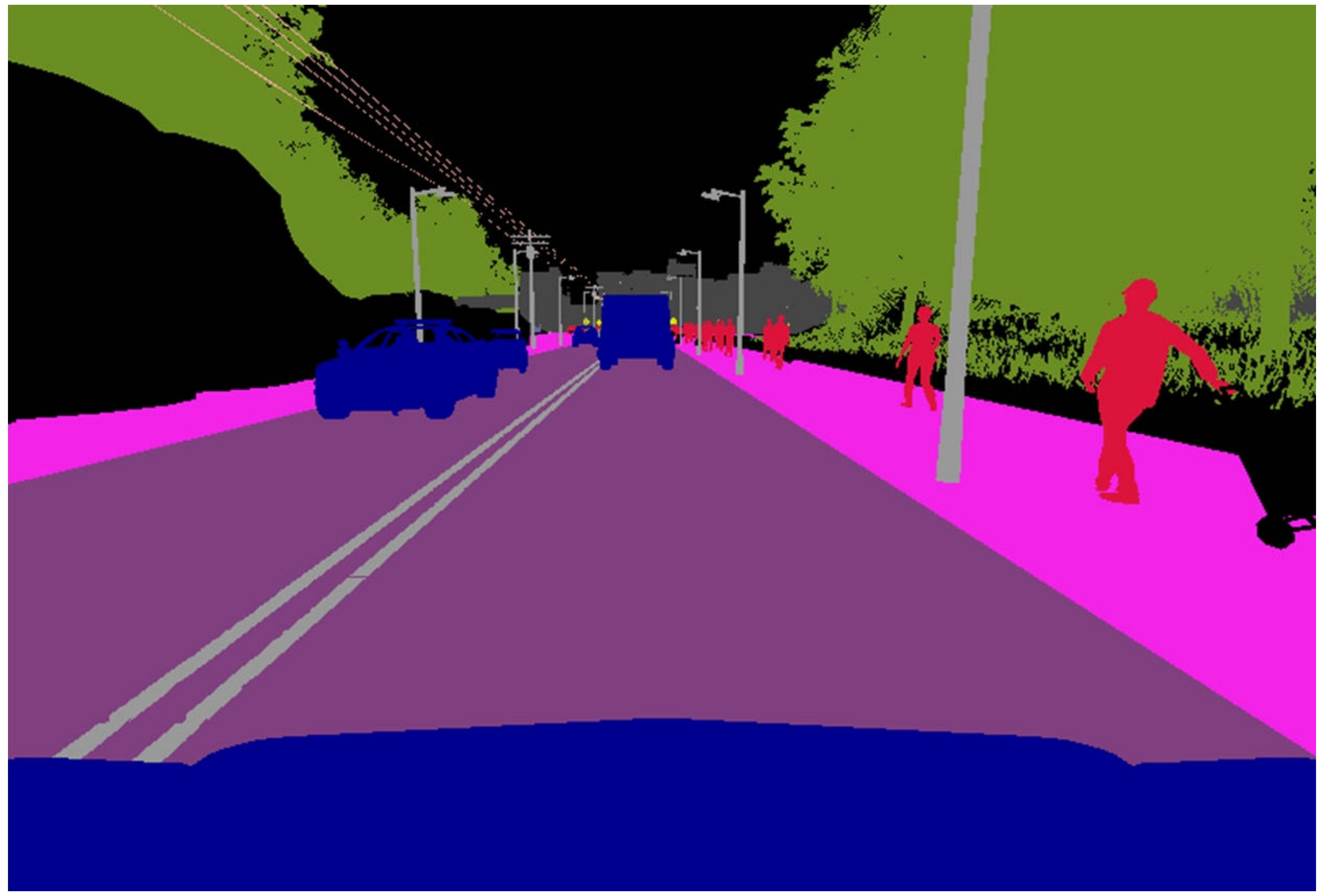


# Example Carla outputs

RGB



Depth



Object type

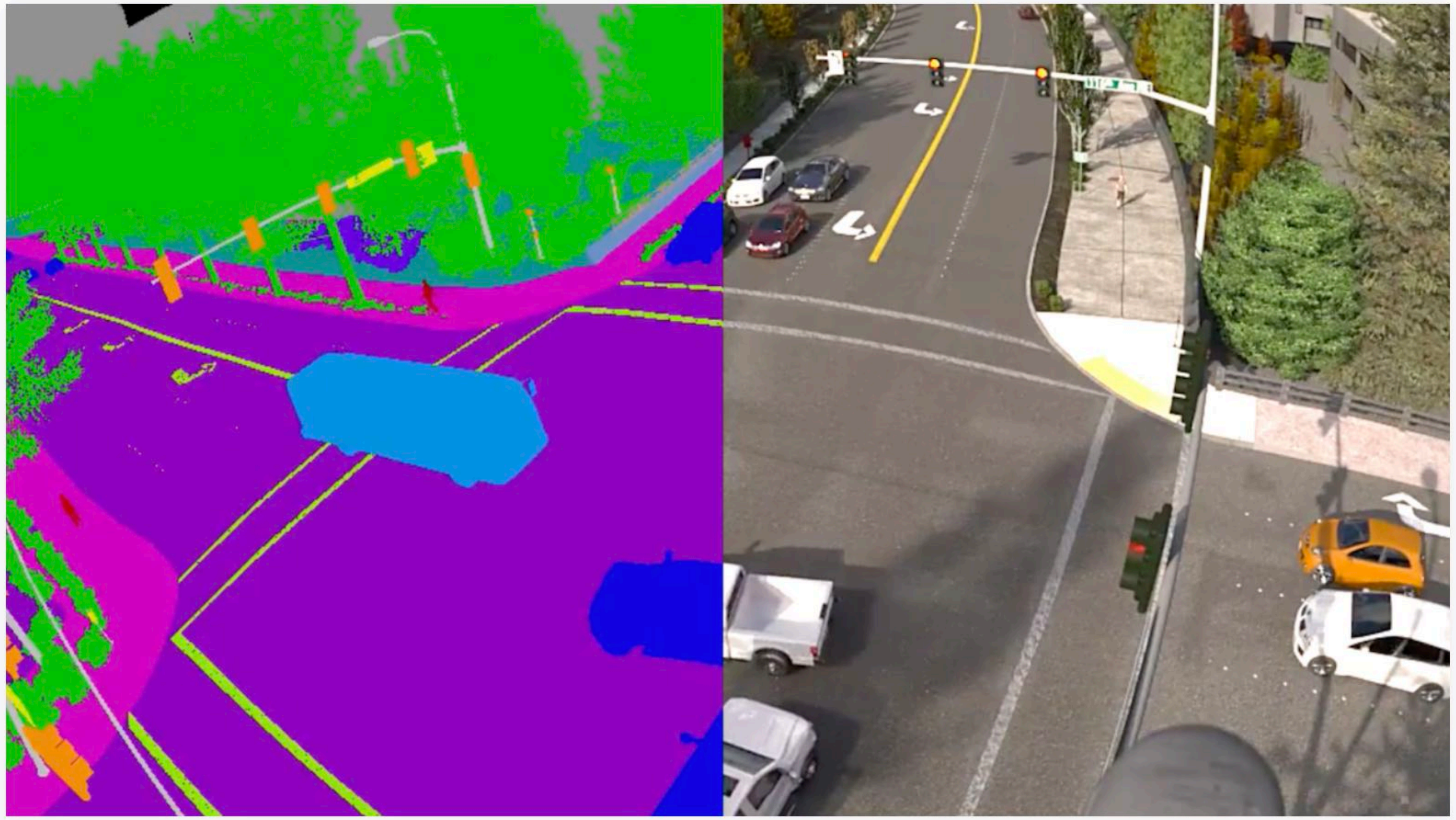
Since renderer has complete description of scene, it can output detailed, fine-grained labels as well as RGB image.

(would be laborious to annotate)

Unity **Blog**

Subscribe to RSS

Search the blog...



# Synthetic data: Simulating myriad possibilities to train robust machine learning models

Srinivas Annambhotla, Cesar Romero and Alex Thaman, May 1, 2020

Machine Learning | Manufacturing

## Share

## Categories

- All
- AEC
- Asset Store
- Community
- Events
- Machine Learning
- Made With Unity
- Manufacturing
- Monetize
- Services
- Technology

## Popular posts

**Introducing the new Input System**
October 14, 2019

**2D Pixel Perfect: How to set up your Unity project for retro 8-bit games**
March 13, 2019

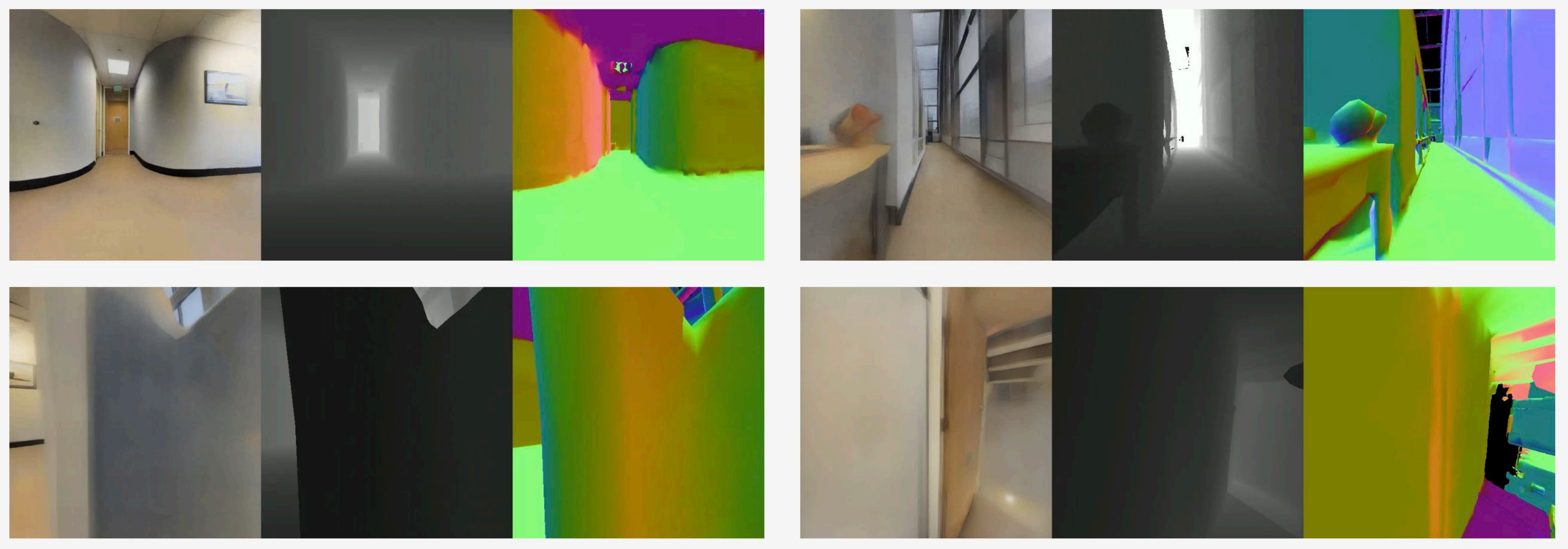
**The High Definition Render Pipeline: Getting Started Guide for Artists**
September 24, 2018

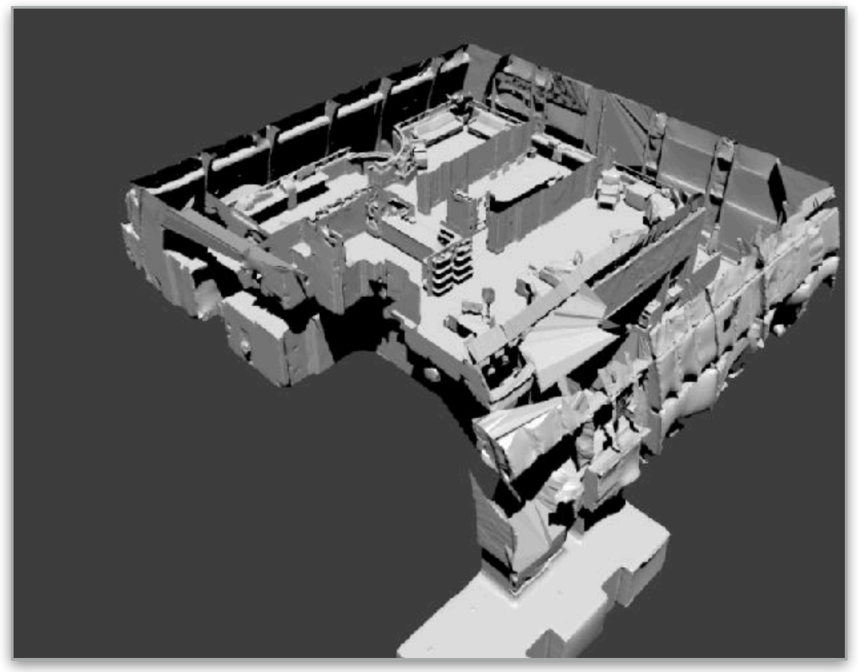

# NVIDIA Drive Sim

# NVIDIA Drive Sim

# Gibson: acquire/render real world data

- Dataset acquired via 3D scanning (3D mesh + texture)
- Geometry, normals, semantics, + (so-called) "photorealistic" 3D





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

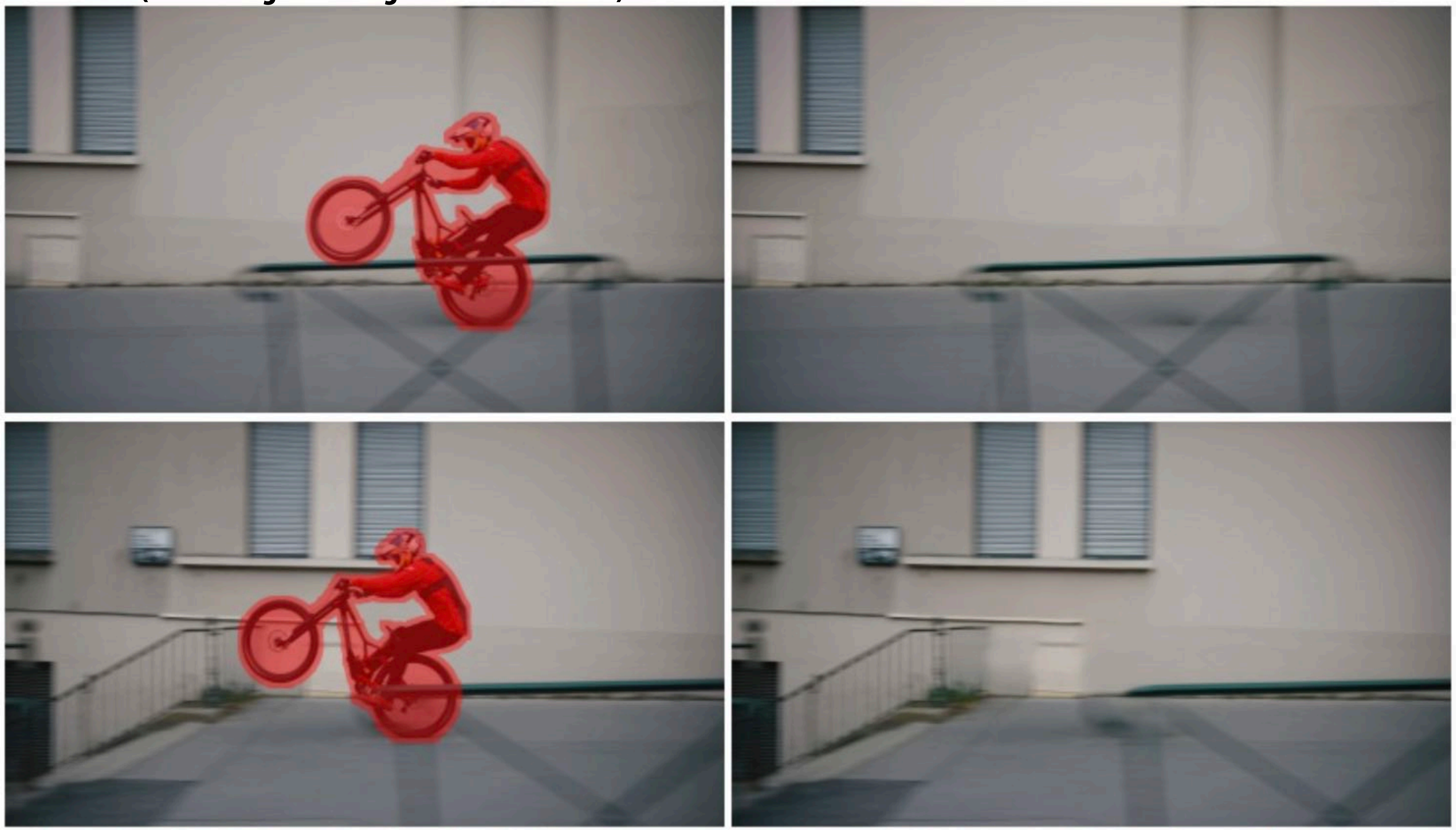
# Modifying real-world images to create novel situations

# Video inpainting

- Identify and remove foreground object
- Hallucinate background with deep neural network

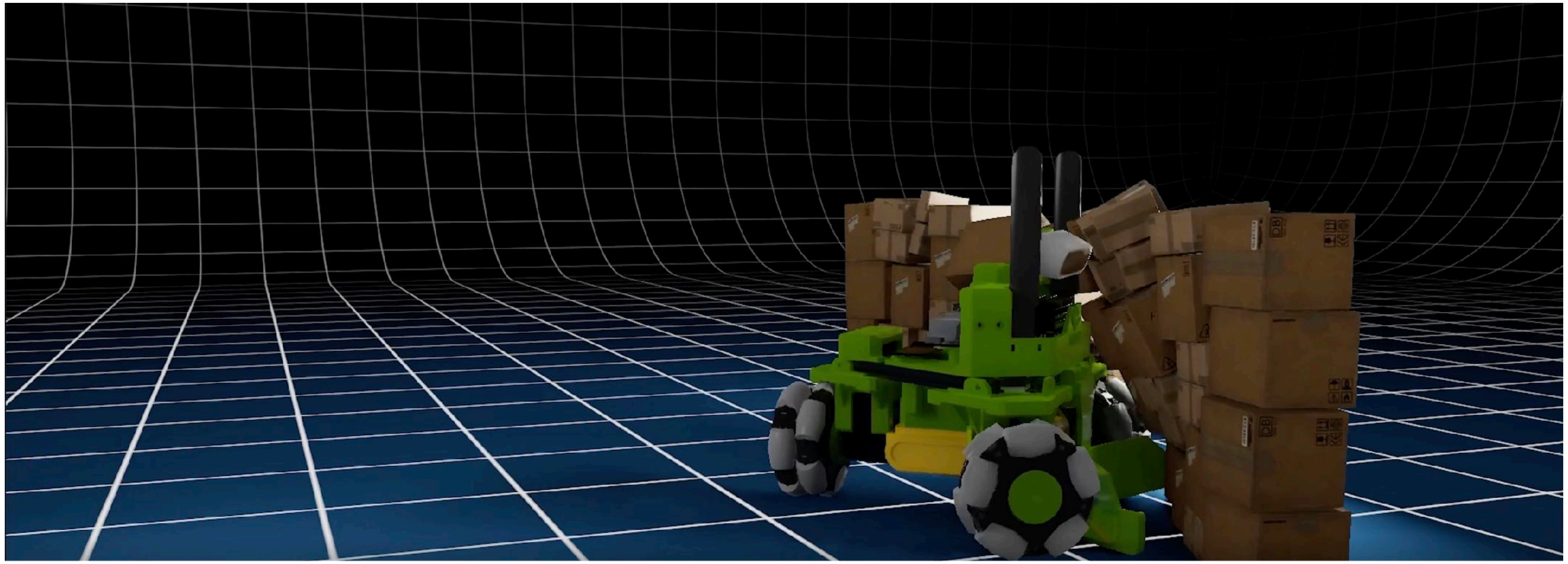
Original video frames
(with foreground segmentation shown)

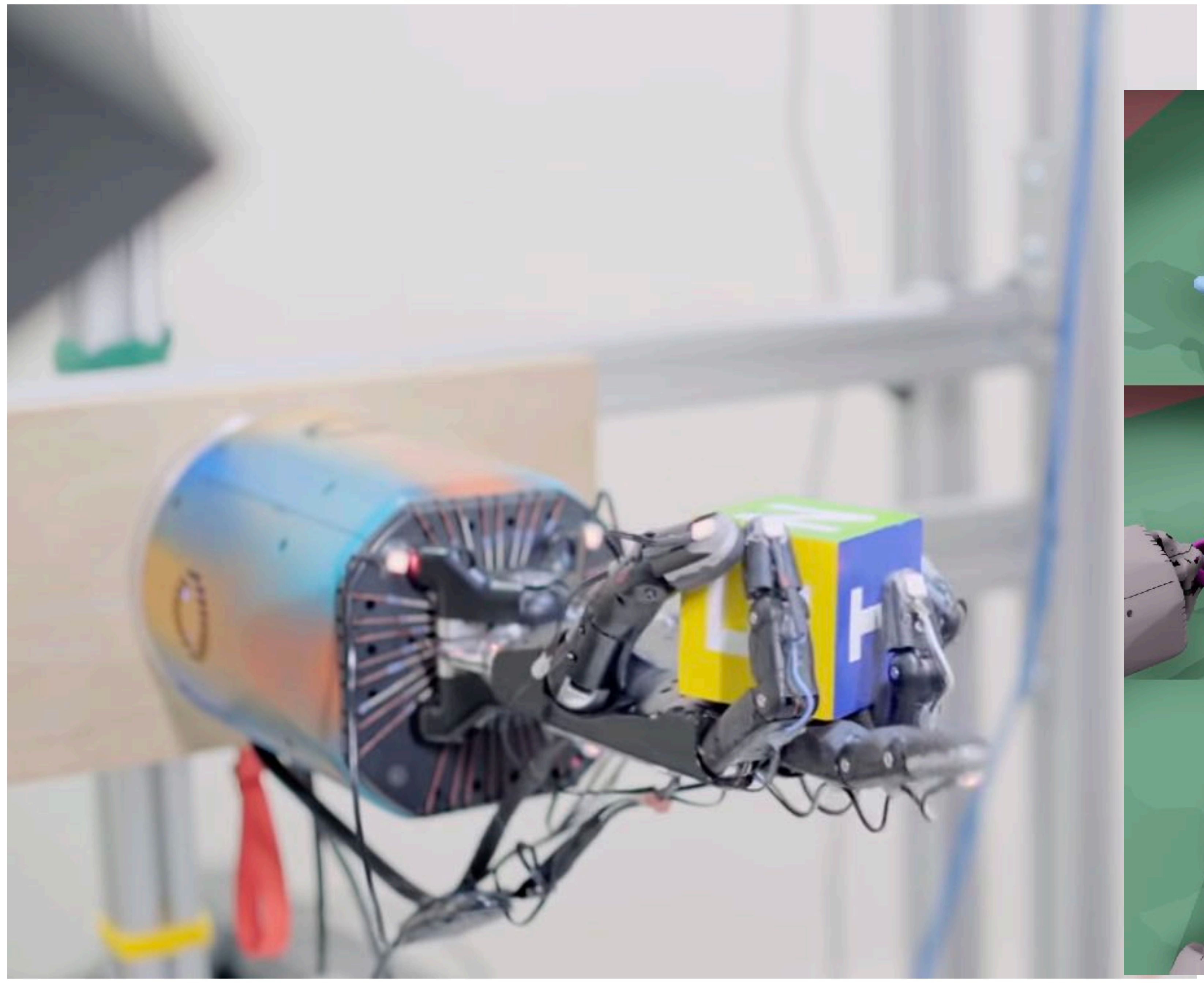
After inpainting foreground regions

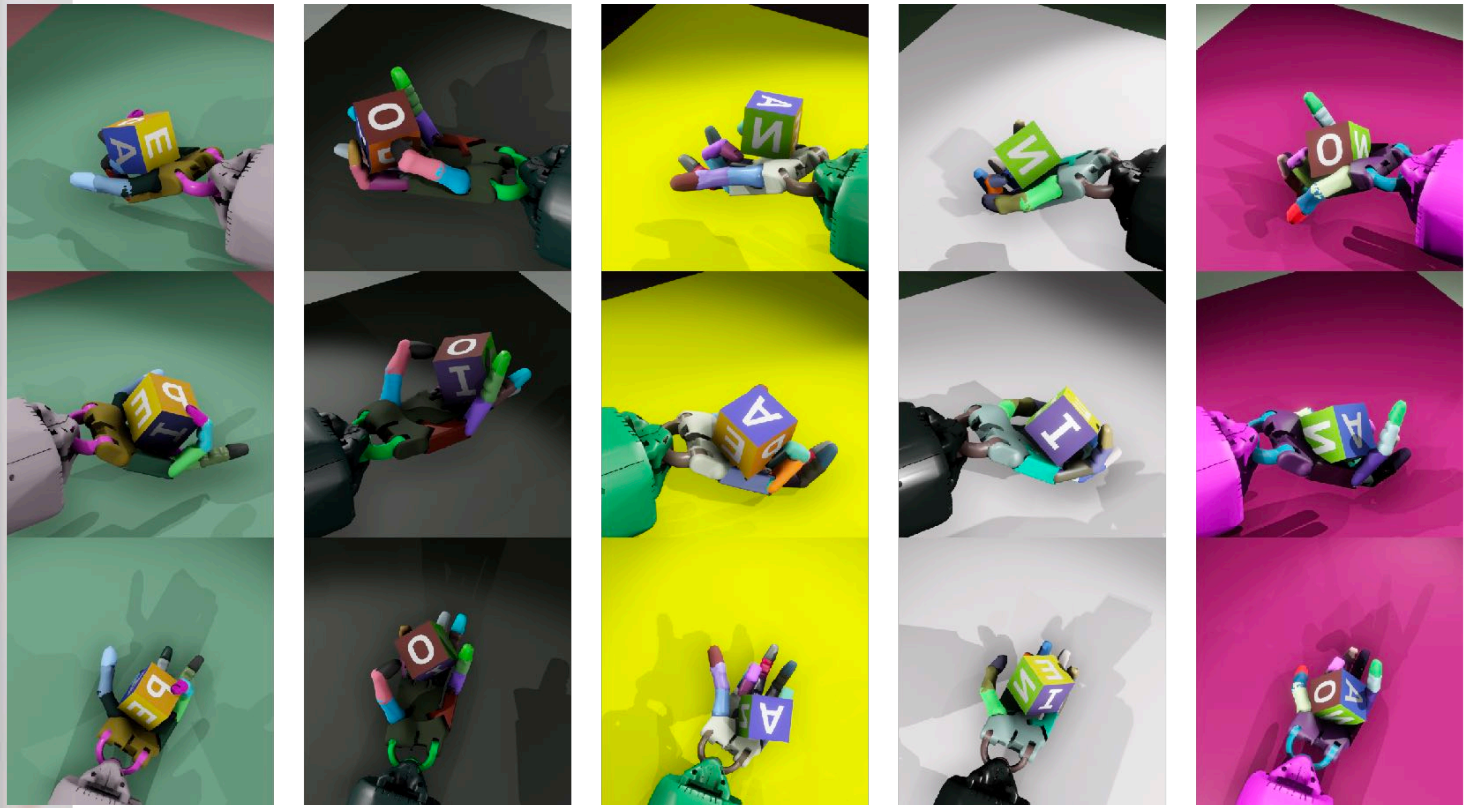


[Ouyang 2021]

# Physics simulation





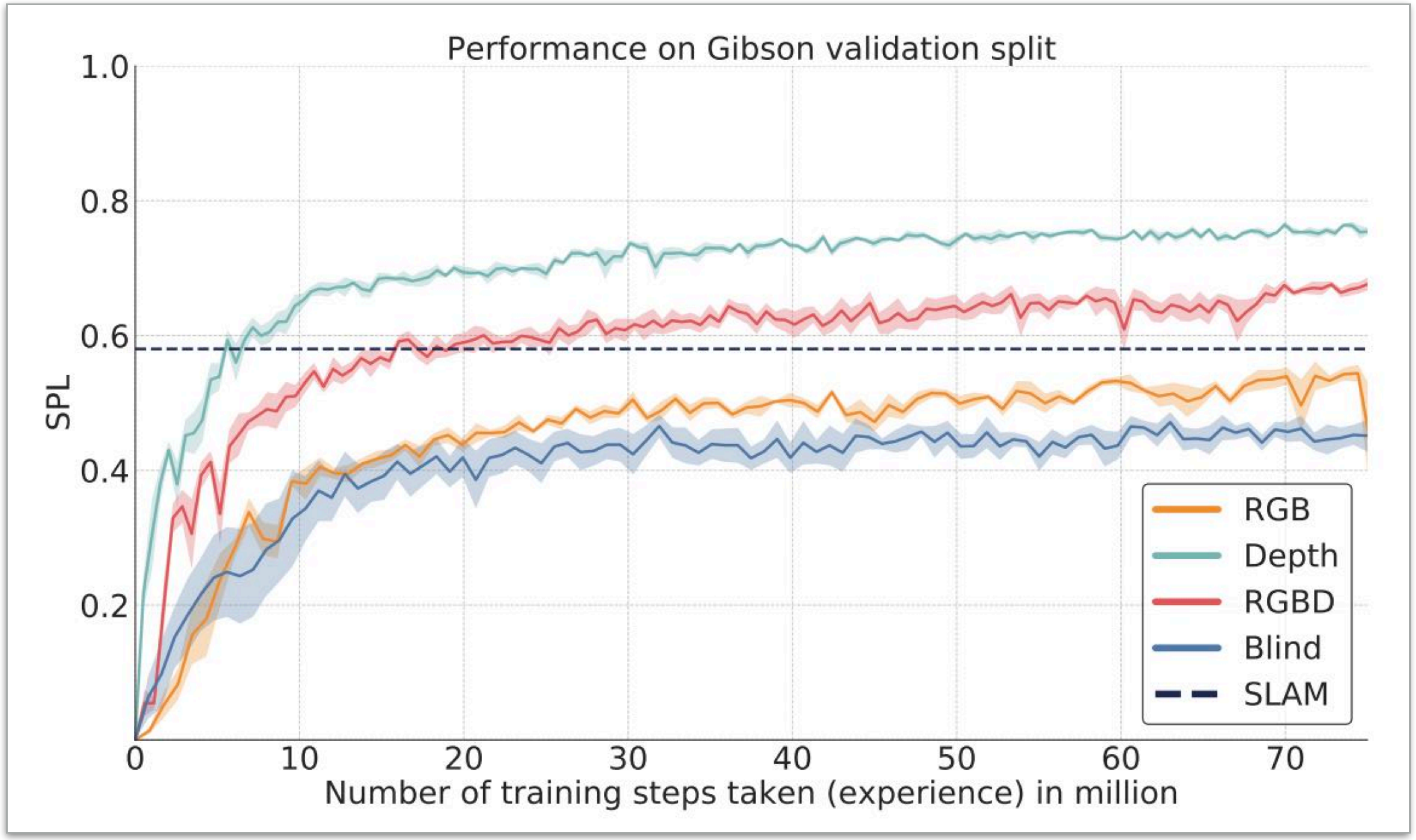


# OpenAI's "OpenAI 5" Dota 2 bot


| | 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 |

# 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**

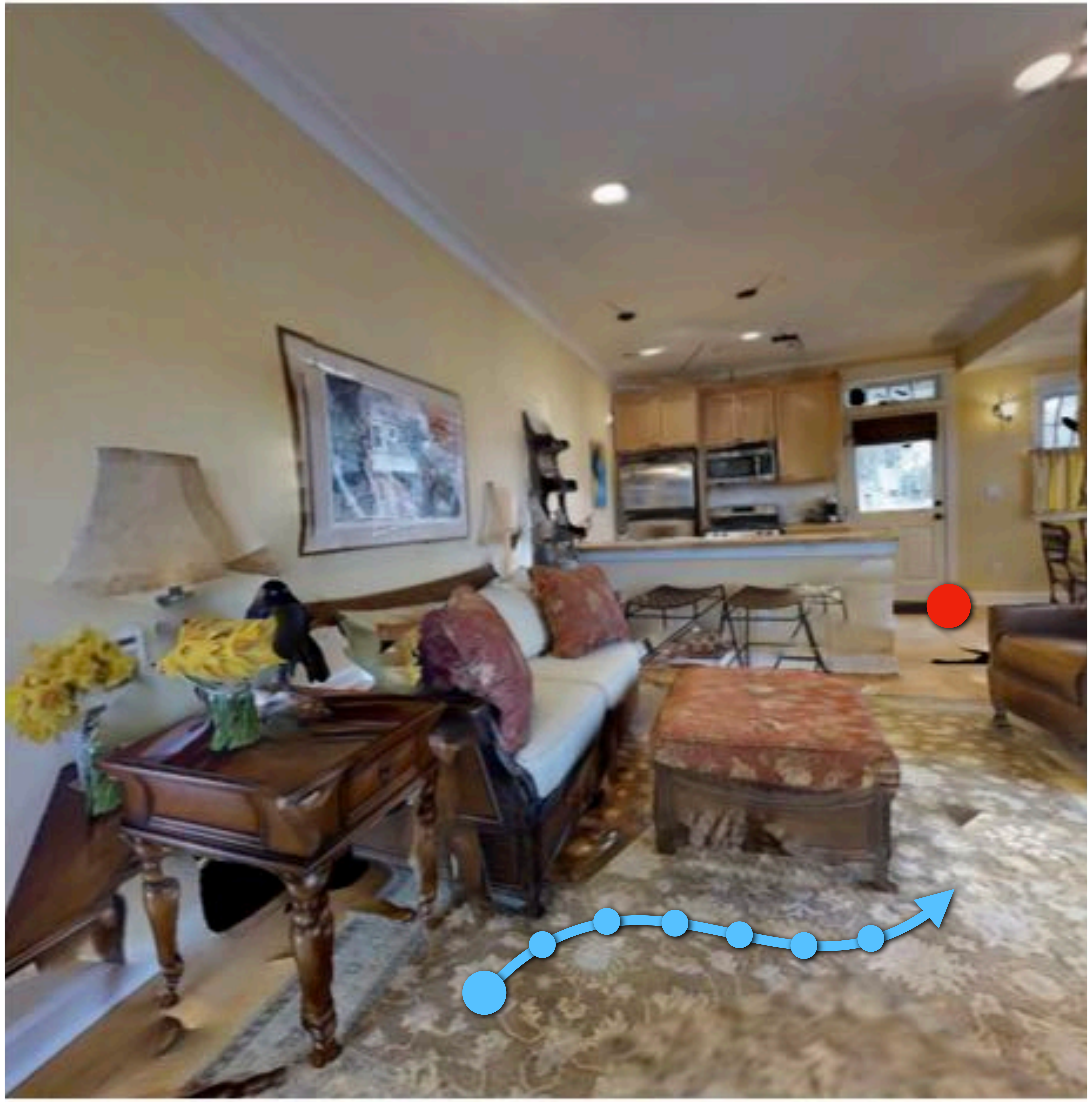


# Deeper dive:
# Accelerating reinforcement learning

# RL in 30 seconds

**Model Inference**

environment observation e.g. RGB image → $\pi_\theta$ → agent action



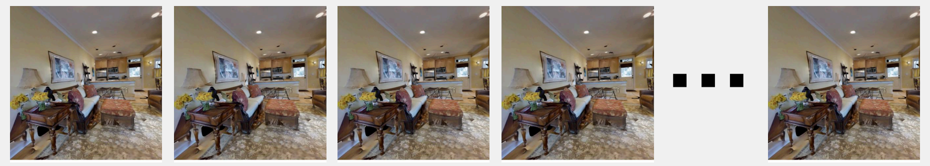
# RL in 30 seconds

## Model Inference

environment observation e.g. RGB image → $\pi_\theta$ → agent action

## Model Training

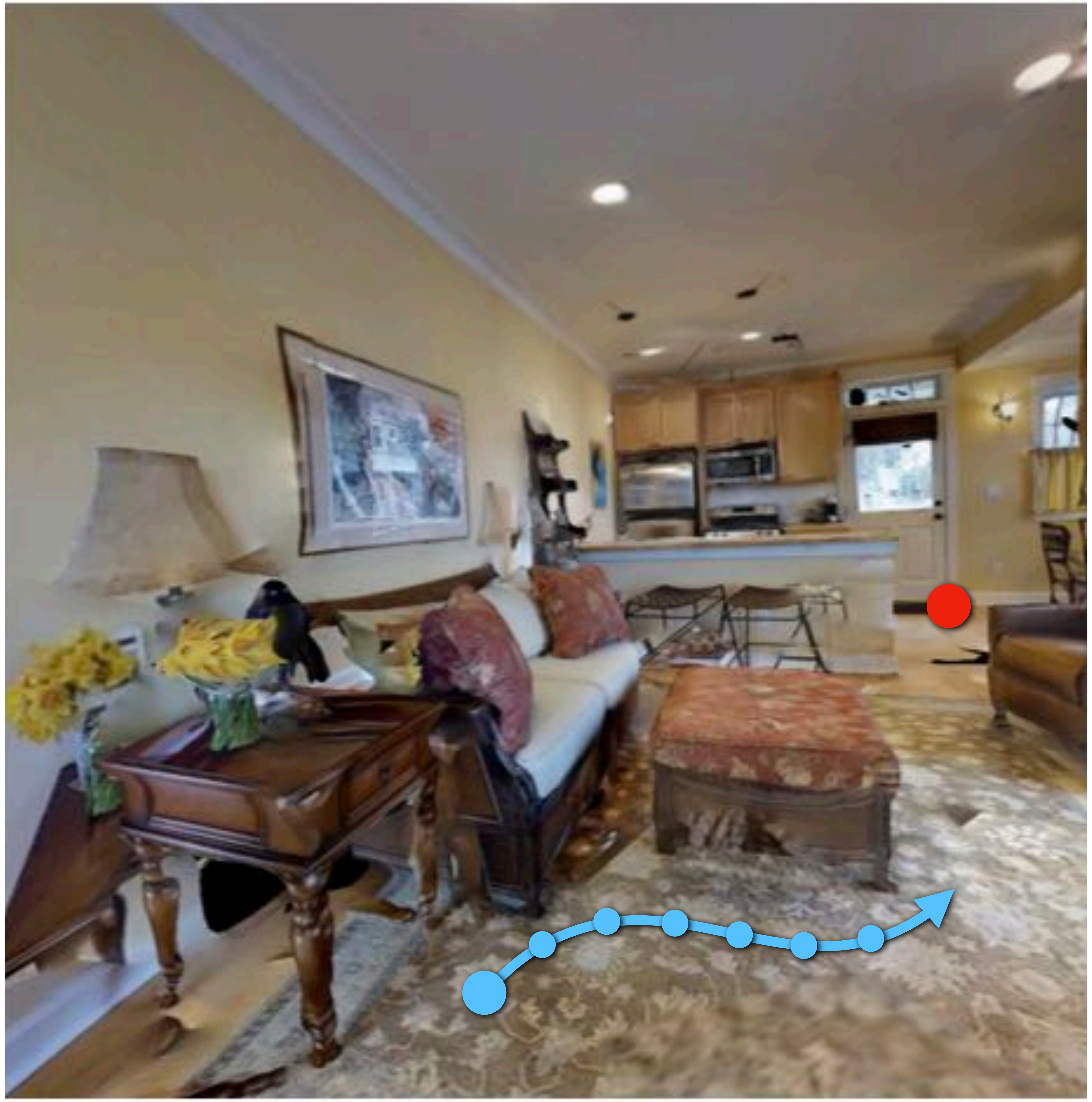
sequence of observations



sequence of agent actions

Reward: change in distance from goal

→ compute loss gradients $\pi_\theta$ → update model via SGD



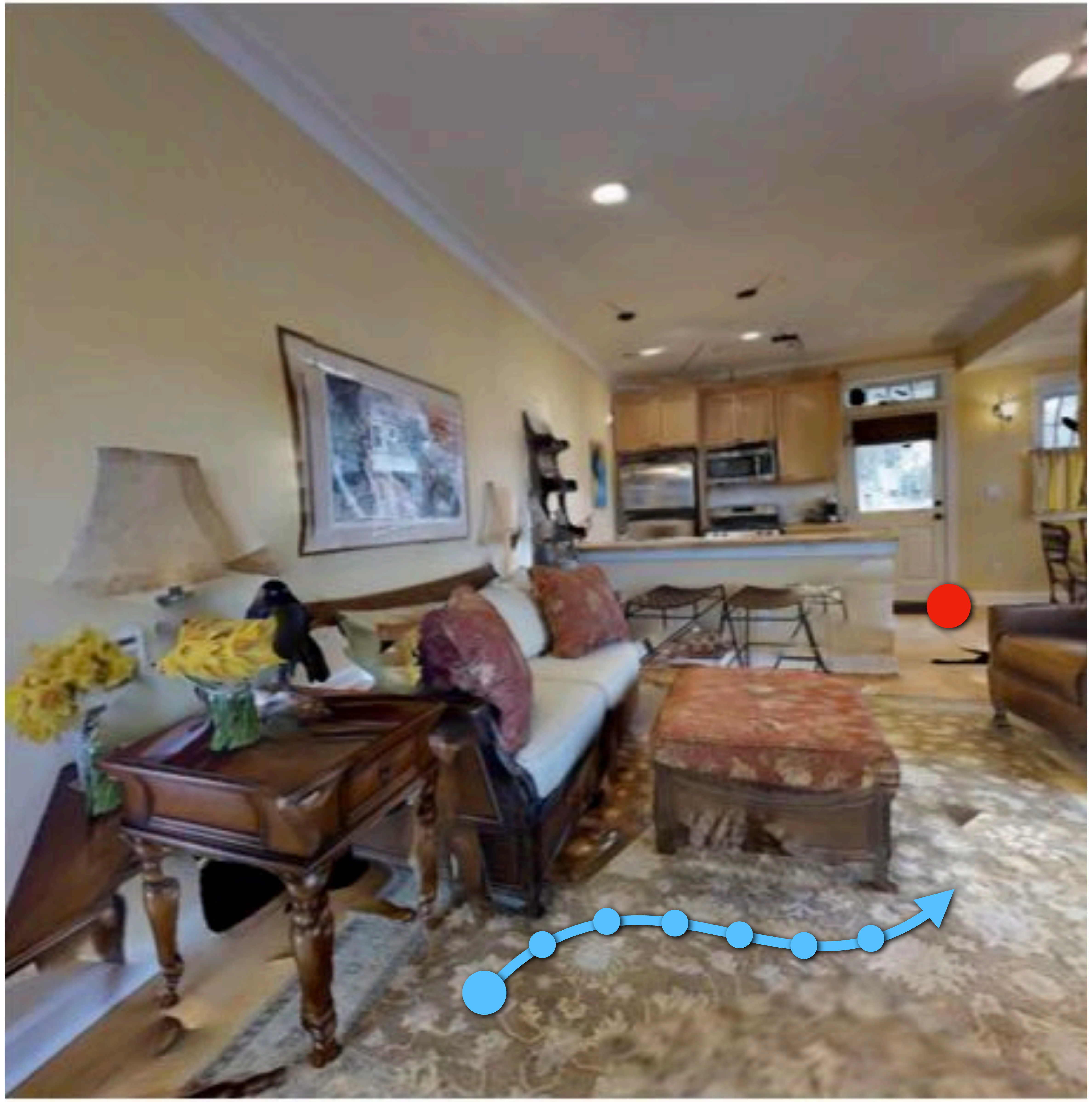
# RL in 30 seconds

## Model Inference

environment observation e.g. RGB image → $\pi_\theta$ → agent action

## Model Training

Rollout → compute loss gradients $\pi_\theta$ → update model via SGD



# RL in 30 seconds

**Many rollouts:**

- **Agents independently navigating same environments**

**Batch Model Training**

Rollout 0

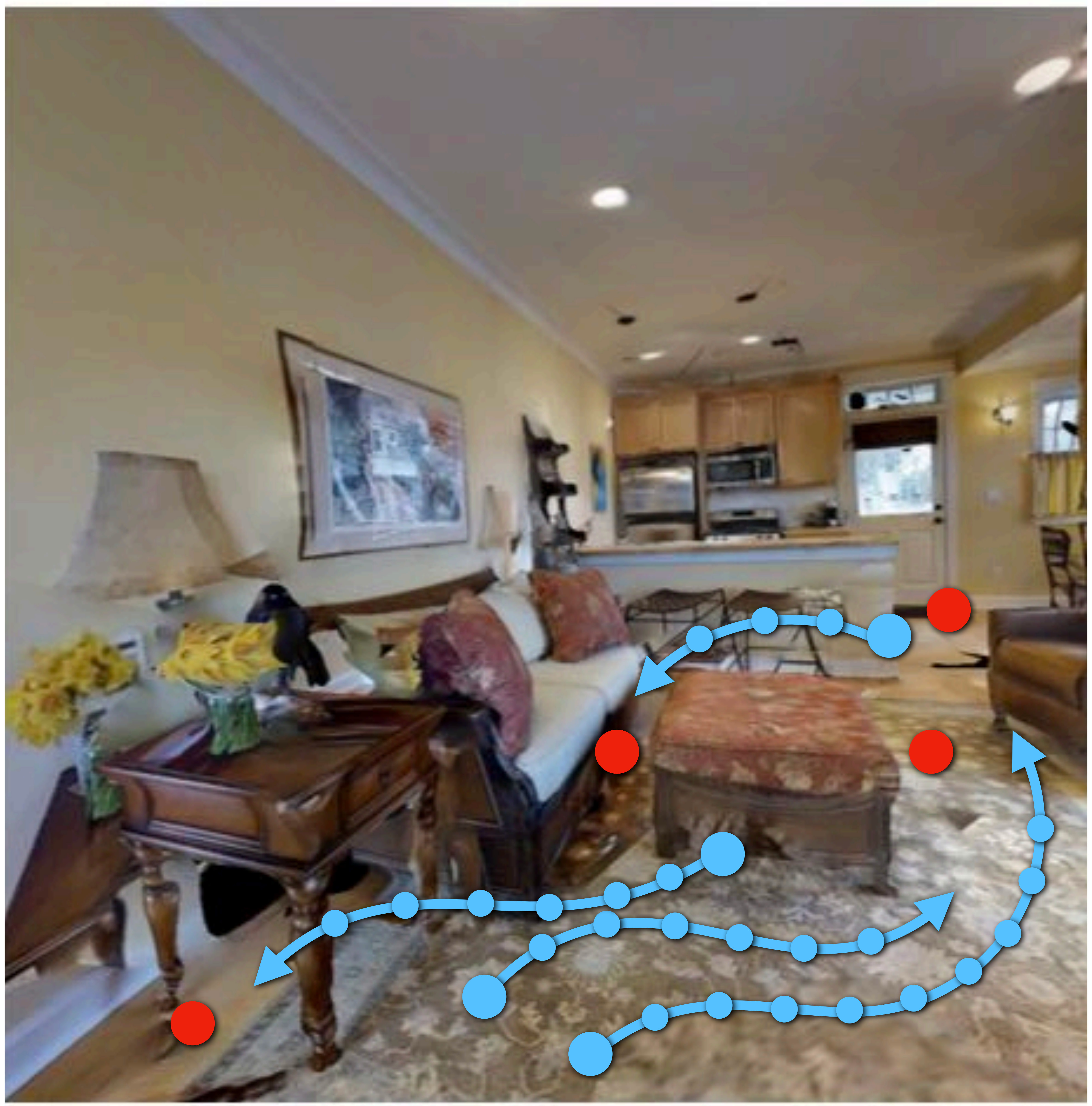
Rollout 1

Rollout 2

...

Rollout N-1

→ compute loss gradients $\pi_\theta$ → update model via SGD



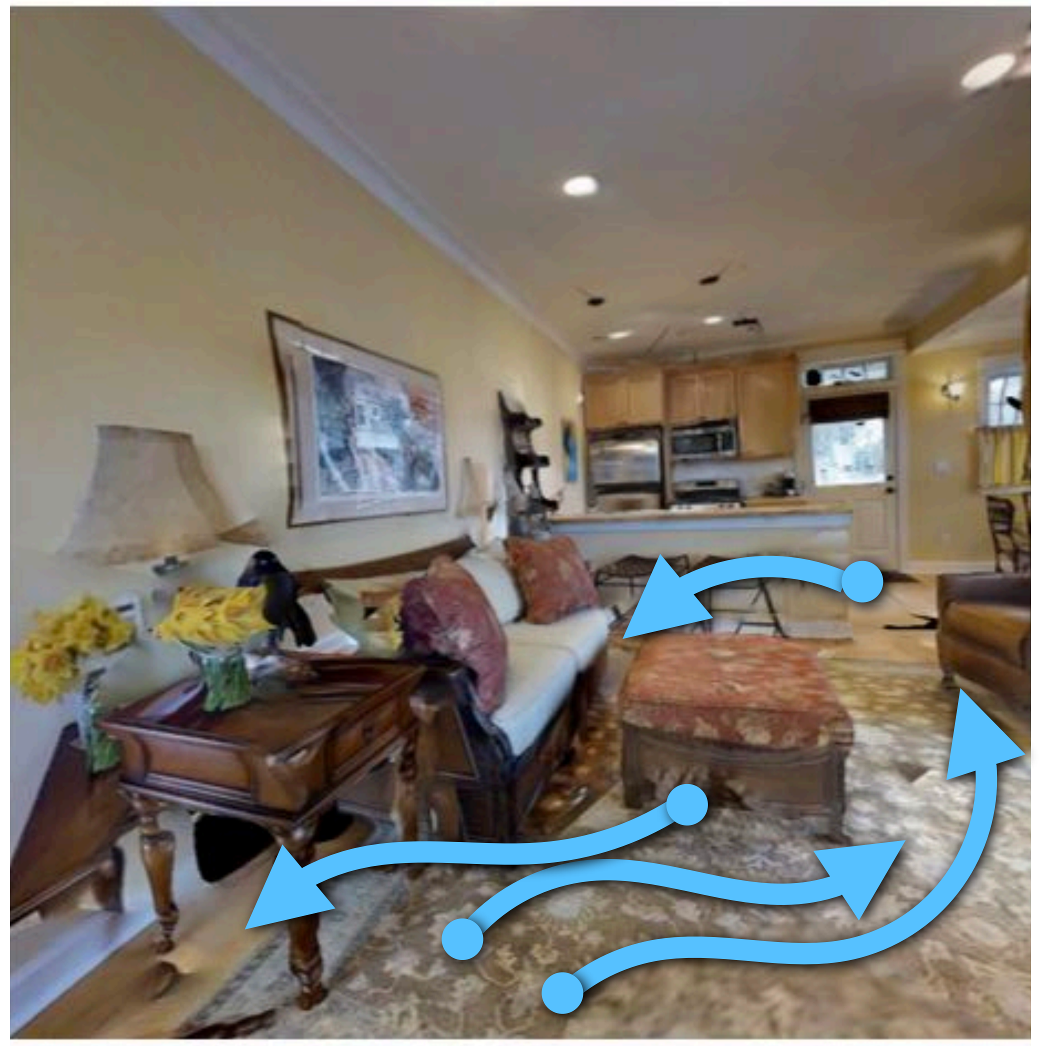
# RL in 30 seconds

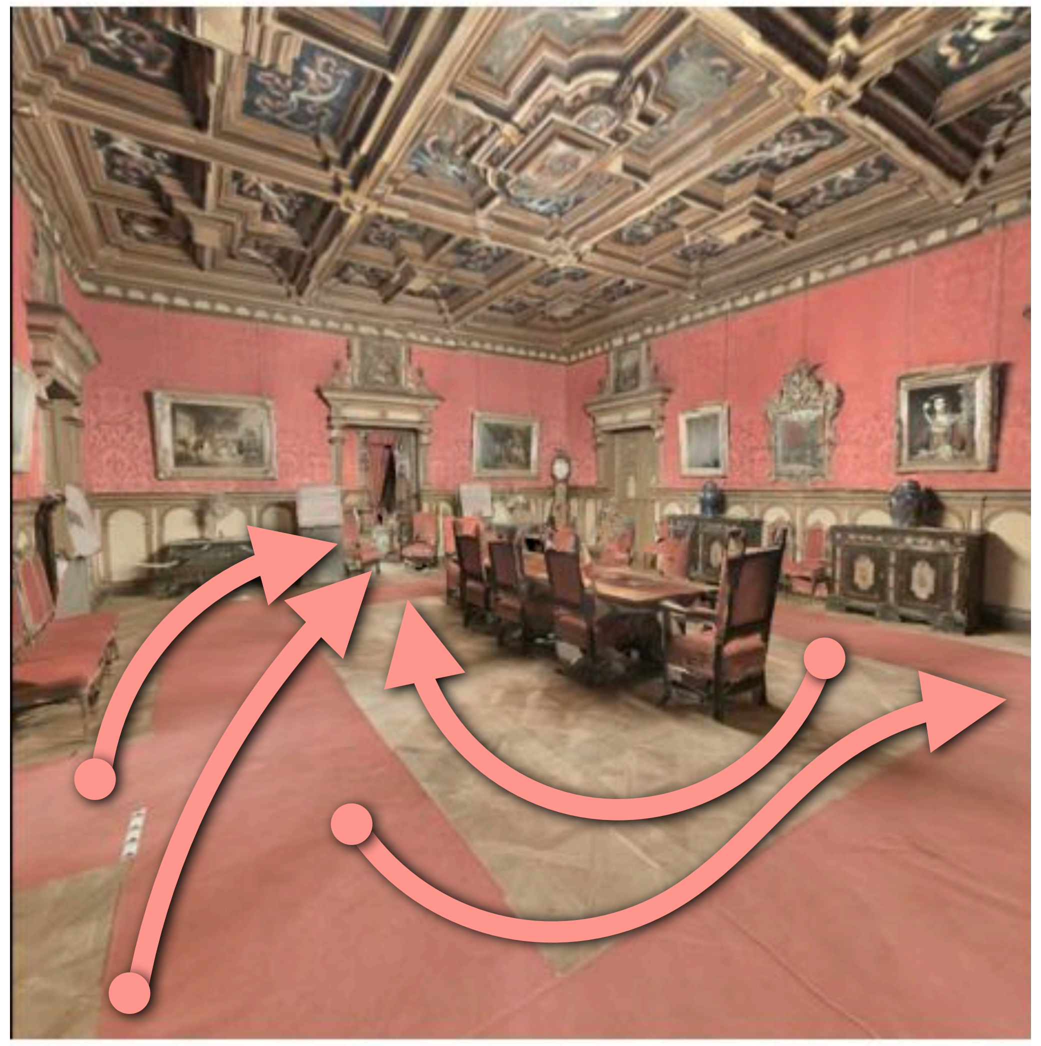
**Many rollouts:**
- Agents independently navigating same environments
- Or different environments

**Batch Model Training**

Rollout 0
Rollout 1
Rollout 2
Rollout 3
Rollout 4
Rollout 5
...
Rollout N-1

→ compute loss gradients $\pi_\theta$ → update model via SGD





# Learning robot skills requires many trials (billions) of learning experience

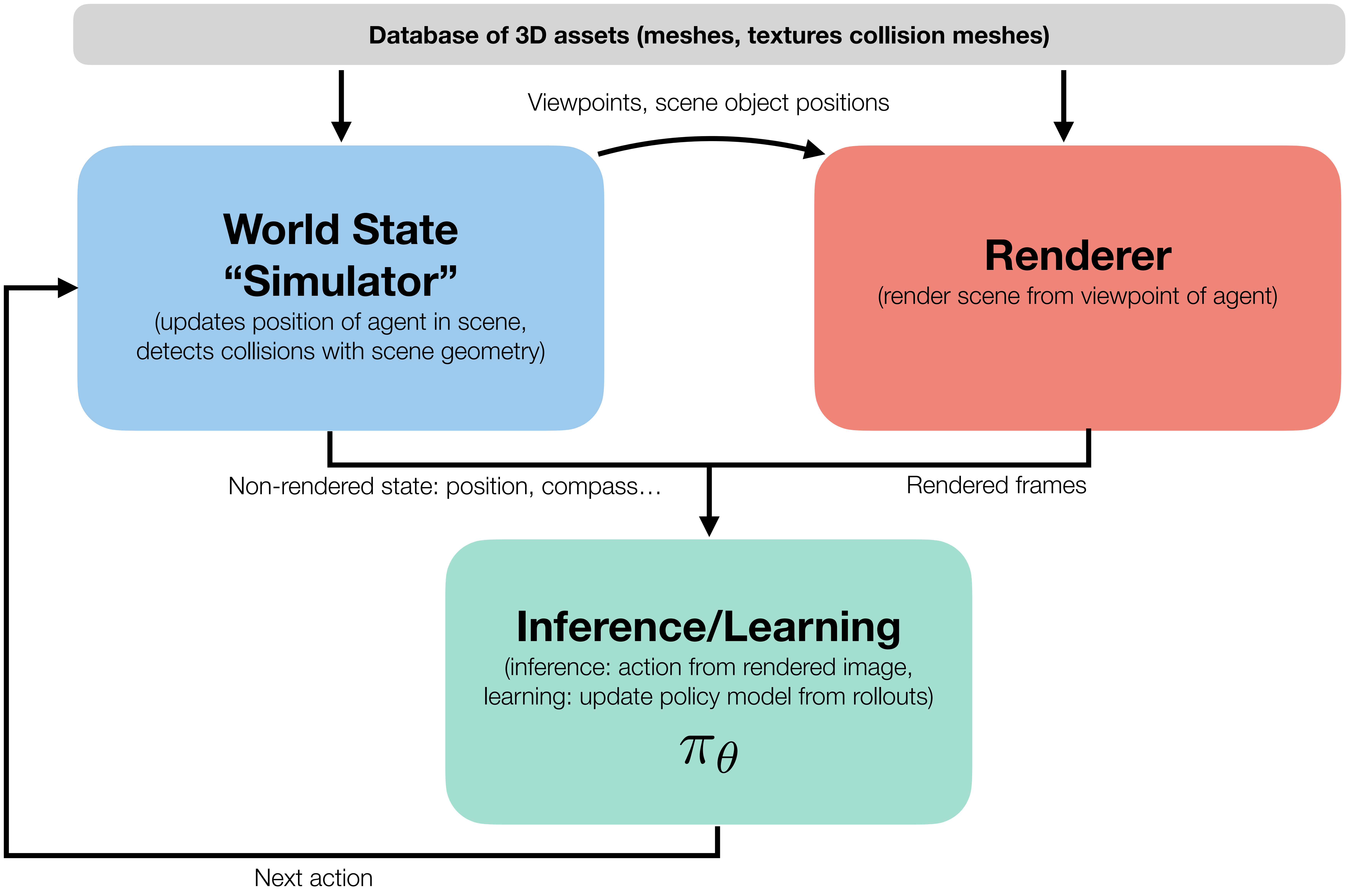
- Training in diverse set of virtual environments
- Many training trials in each environment

# Workload summary

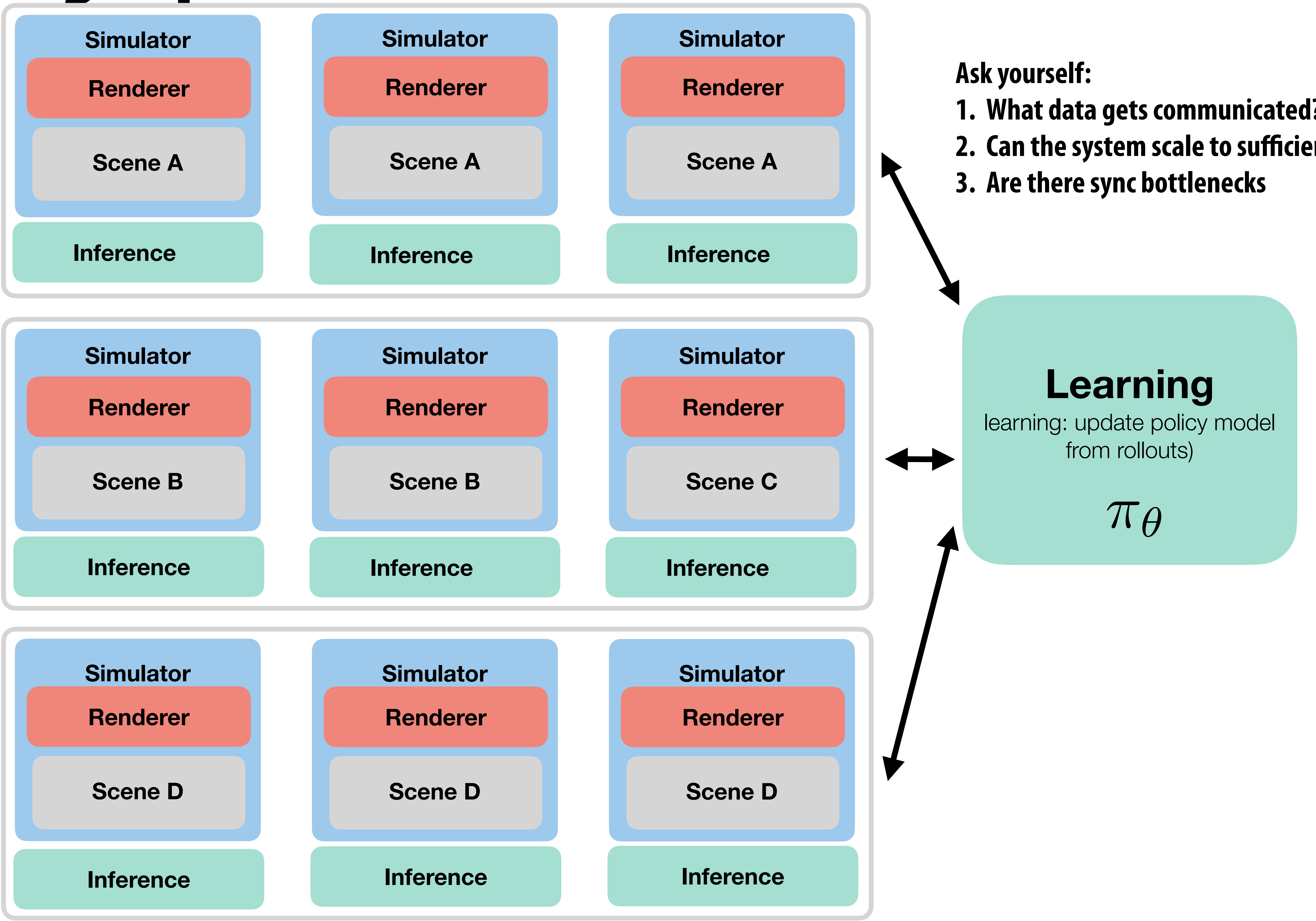
- Within a rollout
  - For each step of a rollout:
  - Render -> Execute policy inference -> simulate next world state

- Across *many* independent rollouts
  - Simulated agents may (or may not) share scene state
  - Diversity in scenes in a batch of rollouts is desirable to avoid overfitting, sample efficiency of learning

# System components



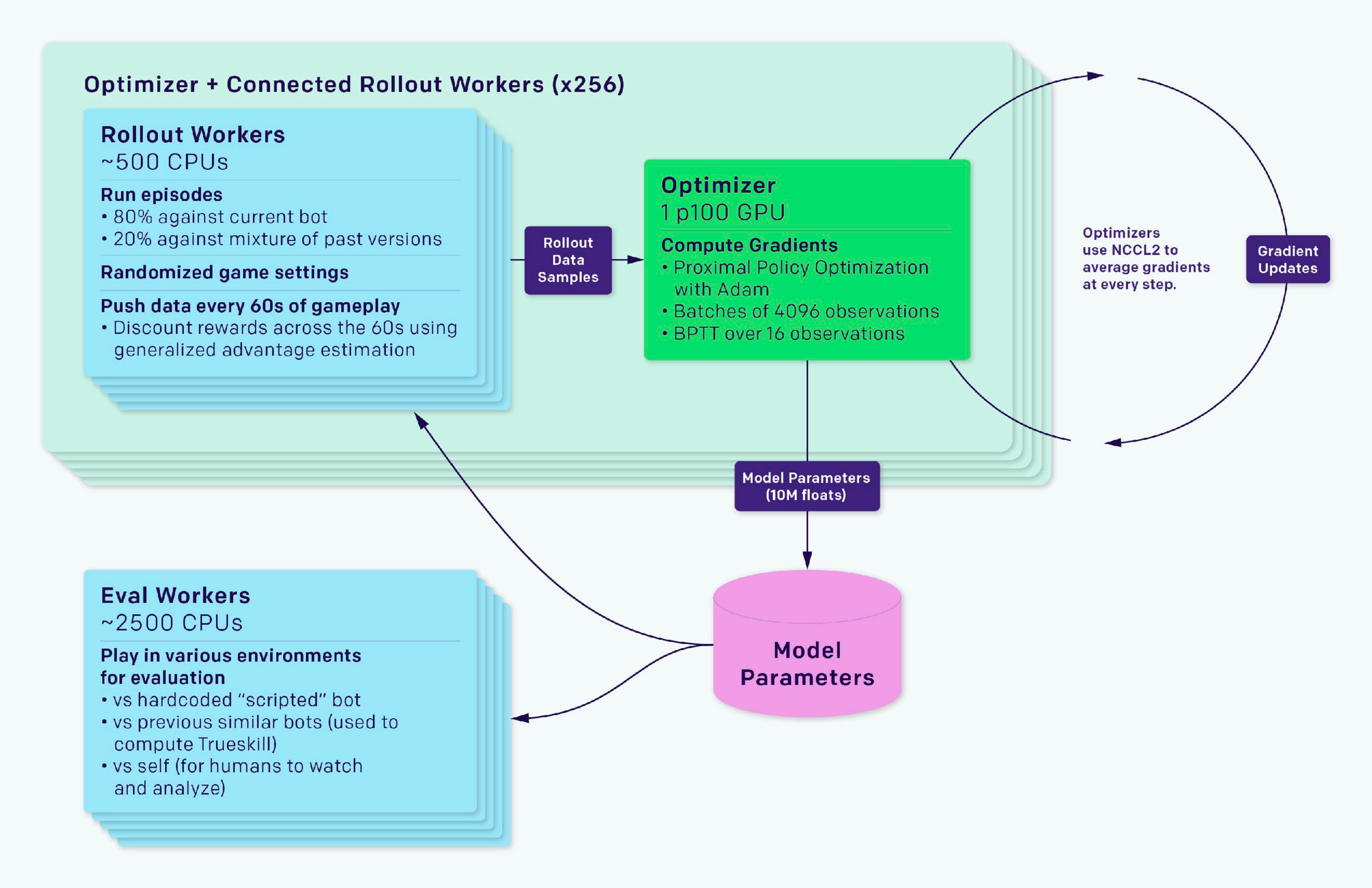
# Basic design: parallelize over workers




**Ask yourself:**

1. **What data gets communicated?**
2. **Can the system scale to sufficient parallelism?**
3. **Are there sync bottlenecks**

# Example: Rapid (OpenAI)

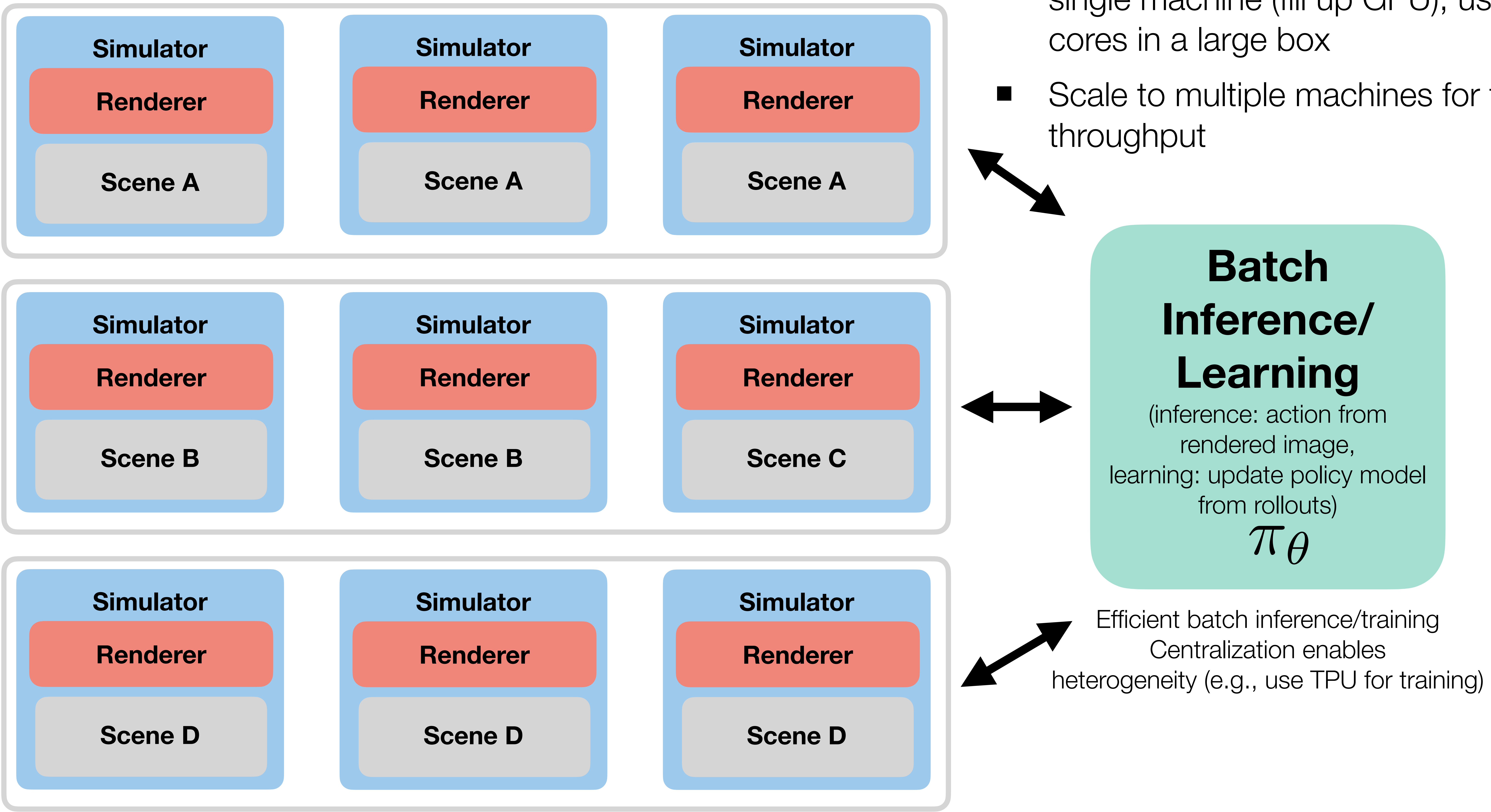


# Design issues

- Expensive communication of weights from learner node to workers
- Worker nodes inefficiently run inference
  - May run on CPU if simulation code on workers doesn't require GPU (use cheap worker nodes that don't feature GPUs)
  - Run inference on small batches since each worker is running one rollout sim

# Centralize inference AND training

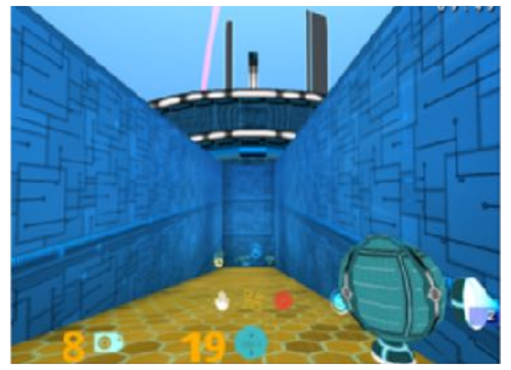
- Instance multiple copies of the engine on a single machine (fill up GPU), use all CPU cores in a large box
- Scale to multiple machines for further throughput

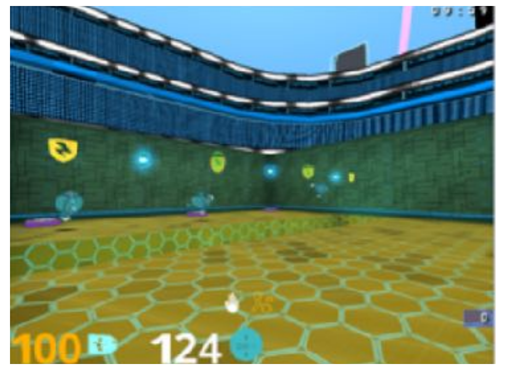


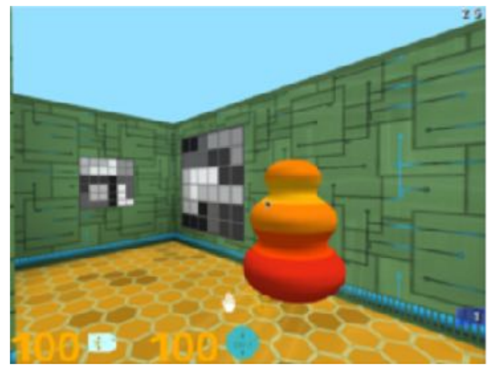
# Advantages

- No communication of model weights between workers and learner
- Must communicate simulation state — surprisingly this can be compact (object locations, smaller rendered image)
- Can use efficient batch inference in a centralized location (batch over rollouts from many workers)
- Can use machine optimized for DNN operations in centralized location — e.g., run on a TPU

# SEED RL







| Architecture | Accelerators | Environments | Actor CPUs | Batch Size | FPS | Ratio |
|---|---|---|---|---|---|---|
| **DeepMind Lab** | | | | | | |
| IMPALA | Nvidia P100 | 176 | 176 | 32 | 30K | — |
| SEED | Nvidia P100 | 176 | 44 | 32 | 19K | **0.63x** |
| SEED | TPU v3, 2 cores | 312 | 104 | 32 | 74K | **2.5x** |
| SEED | TPU v3, 8 cores | 1560 | 520 | 48[1] | 330K | **11.0x** |
| SEED | TPU v3, 64 cores | 12,480 | 4,160 | 384[1] | 2.4M | **80.0x** |
| **Google Research Football** | | | | | | |
| IMPALA, Default | 2 x Nvidia P100 | 400 | 400 | 128 | 11K | — |
| SEED, Default | TPU v3, 2 cores | 624 | 416 | 128 | 18K | **1.6x** |
| SEED, Default | TPU v3, 8 cores | 2,496 | 1,664 | 160[3] | 71K | **6.5x** |
| SEED, Medium | TPU v3, 8 cores | 1,550 | 1,032 | 160[3] | 44K | — |
| SEED, Large | TPU v3, 8 cores | 1,260 | 840 | 160[3] | 29K | — |
| SEED, Large | TPU v3, 32 cores | 5,040 | 3,360 | 640[3] | 114K | **3.9x** |
| **Arcade Learning Environment** | | | | | | |
| R2D2 | Nvidia V100 | 256 | N/A | 64 | 85K[2] | — |
| SEED | Nvidia V100 | 256 | 55 | 64 | 67K | **0.79x** |
| SEED | TPU v3, 8 cores | 610 | 213 | 64 | 260K | **3.1x** |
| SEED | TPU v3, 8 cores | 1200 | 419 | 256 | 440K[4] | **5.2x** |

# Design issues

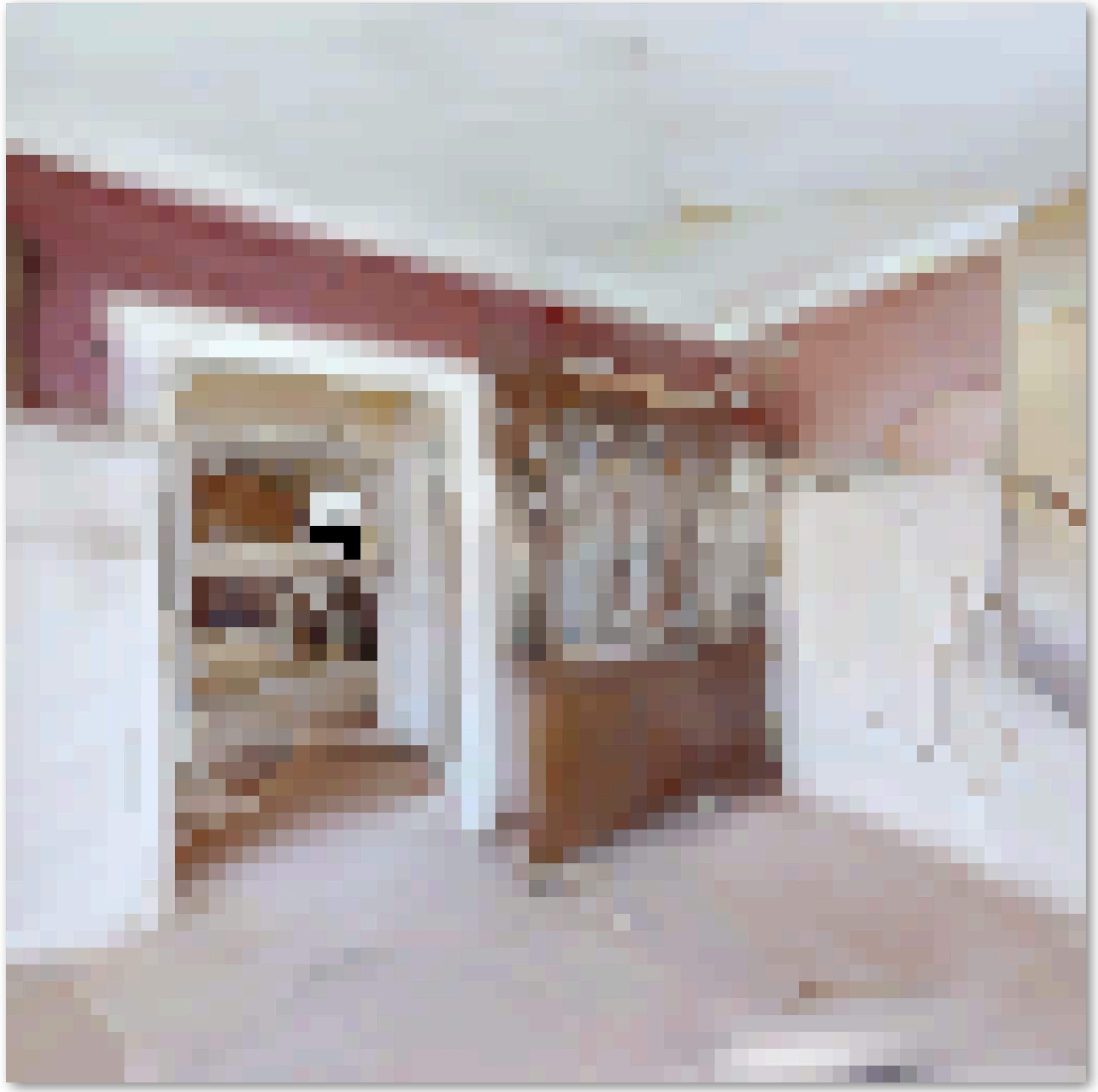
- Inefficient simulation/rendering: rendering a small image does not make good use of a modern GPU (rendering throughput is low)
- Duplication of computation and memory footprint (for scene data) across renderer/simulator instances

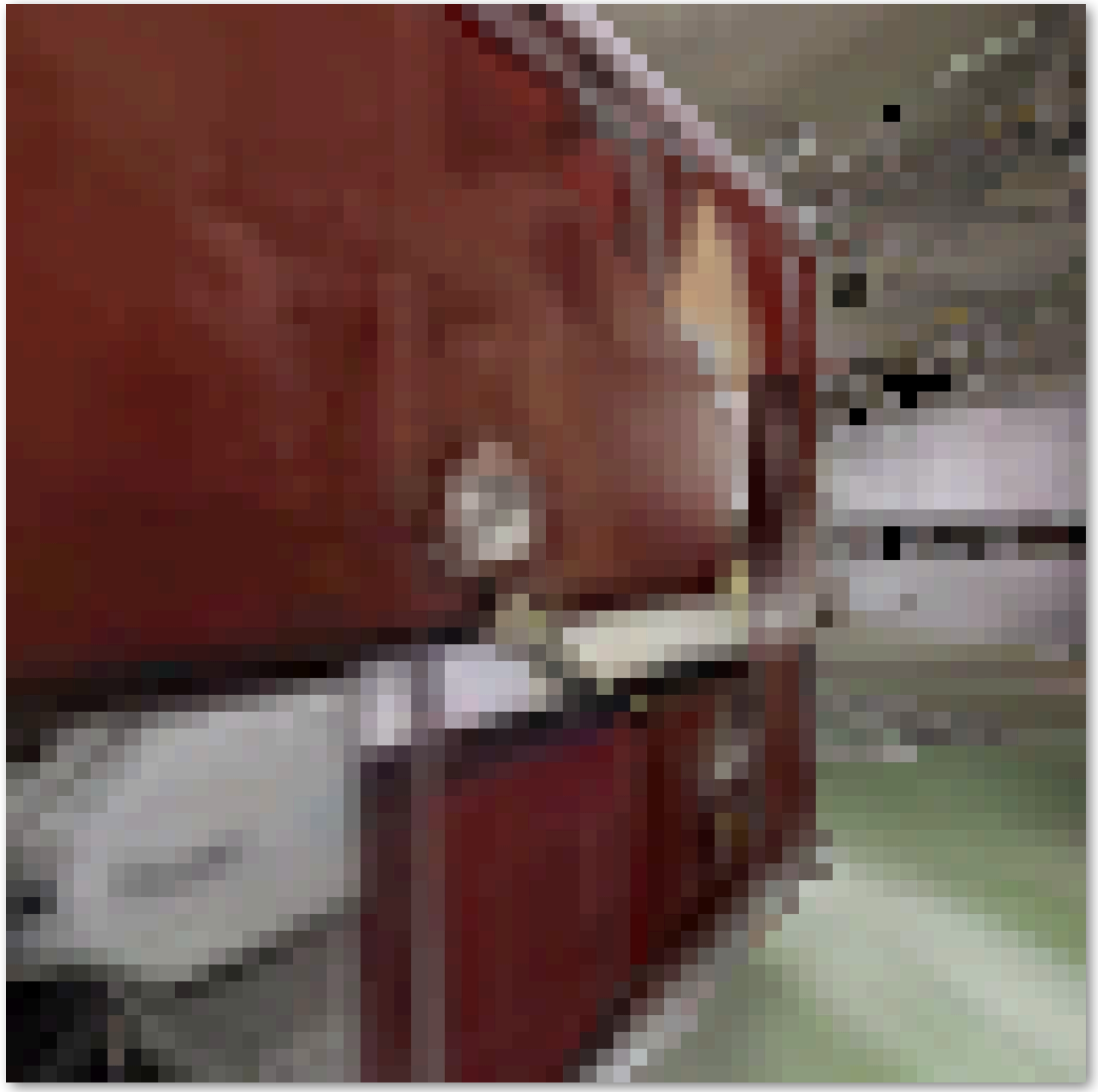
# What modern graphics engines are designed to render

- 4K image outputs
- 30-60 fps
- Advanced lighting and material simulation

Forza-Horizon-5

# Low-resolution images with pre-captured lighting (from Gibson): clearly not state-of-the-art rendering! ;-)





**Often the best way to reduce communication / increase efficiency is often to make the best possible use out of one node**

**Can we make simulation faster?**

# AI Habitat

- Focus on high-performance rendering/simulation to enable order of magnitude longer RL training runs

The table below reports performance statistics for a test scene from the Matterport3D dataset (id `17DRP5sb8fy`) on a `Xeon E5-2690 v4 CPU` and `Nvidia Titan Xp`. Single-thread performance reaches several thousand frames per second, while multi-process operation with several independent simulation backends can reach more than 10,000 frames per second on a single GPU!

| | 1 proc | | | 3 procs | | | 5 procs | | |
|---|---|---|---|---|---|---|---|---|---|
| **Sensors / Resolution** | **128** | **256** | **512** | **128** | **256** | **512** | **128** | **256** | **512** |
| RGB | 4093 | 1987 | 848 | 10638 | 3428 | 2068 | 10592 | 3574 | 2629 |
| RGB + depth | 2050 | 1042 | 423 | 5024 | 1715 | 1042 | 5223 | 1774 | 1348 |
| RGB + depth + semantics* | 709 | 596 | 394 | 1312 | 1219 | 979 | 1521 | 1429 | 1291 |

Previous simulation platforms that have operated on similar datasets typically produce on the order of a couple hundred frames per second. For example Gibson reports up to about 150 fps with 8 processes, and MINOS reports up to about 167 fps with 4 threads.

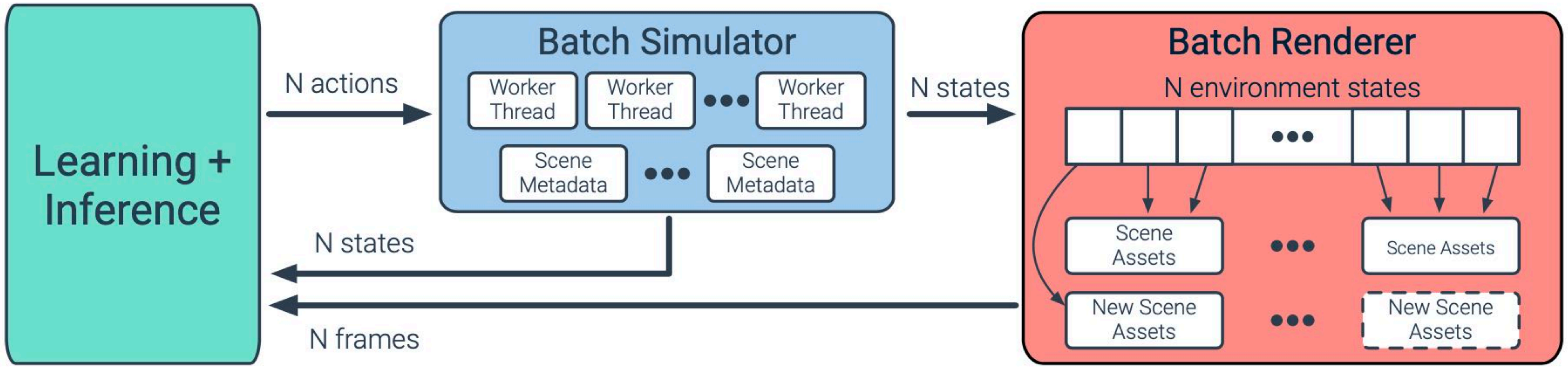
**Prior work was still using simulators (game engines) designed to render large high-resolution images for human eyes.**

**How would you design an engine “from the ground up” for the RL workload?**

# Main idea: design a renderer that executes rendering for 100s-1000's of unique rollouts in a single request

Inference/training, simulation, and rendering all operate on batches of N requests (rollouts)

Efficient bulk communication between three components



# Example renderer output (PointNav task)

# Opportunities provided by a batch rendering interface

- Wide parallelism: rendering each scene in a batch is independent
  - "Fill up" large parallel GPU with rendering work
  - Enables graphics optimizations like pipelining frustum culling (removing off-screen geometry before drawing it) for one environment with rendering of another

- Footprint optimizations: rendering requests in a batch can share same geometry assets
  - Significantly reduces memory footprint, enables large batch size
  - N ~ 256-1024 (per GPU) in our experiments: fills up large GPU
  - Limit number of unique scenes in a batch to $K \ll N$ scenes.
    - GPU RAM and scene size determines K

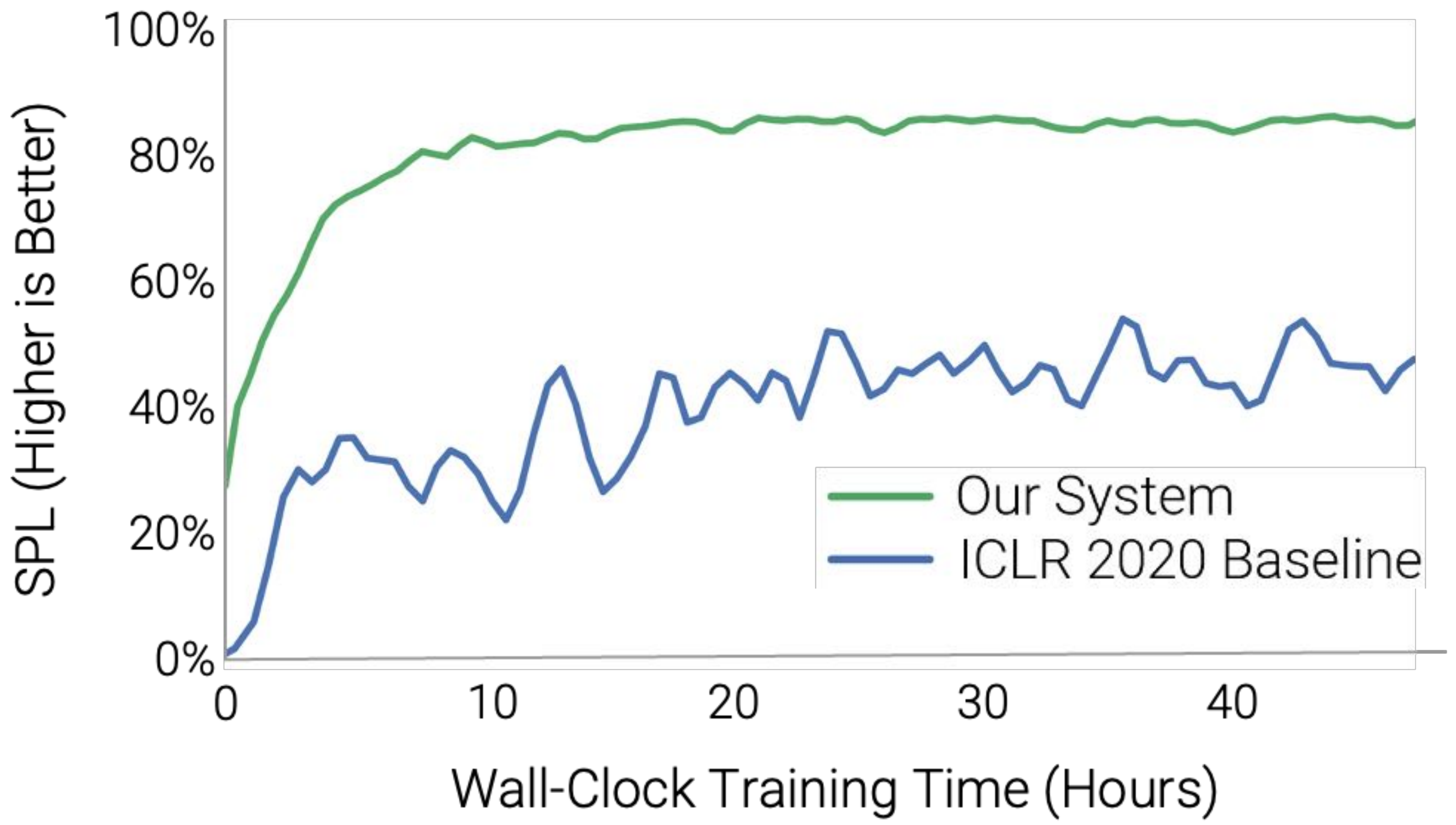
- Amortize communication: rendering requests in a batch can be packaged and drawn together
  - Render frames in batch to tiles in a single large frame buffer to avoid state update

# Also, simultaneously optimize policy DNN

- DNN design/engineering (DNN encoder followed by policy LSTM)
- Reduce resolution of rendered input to from 128x128 to 64x64
- Move to ResNet9-based visual encoder from ResNet50
- Replace key layers with performant alternatives (e.g. replace normalization with Fixup Initialization)
- Adjust learning rates and use Lamb optimization

# Example: 10,000+ FPS render → infer → train on a single GPU *

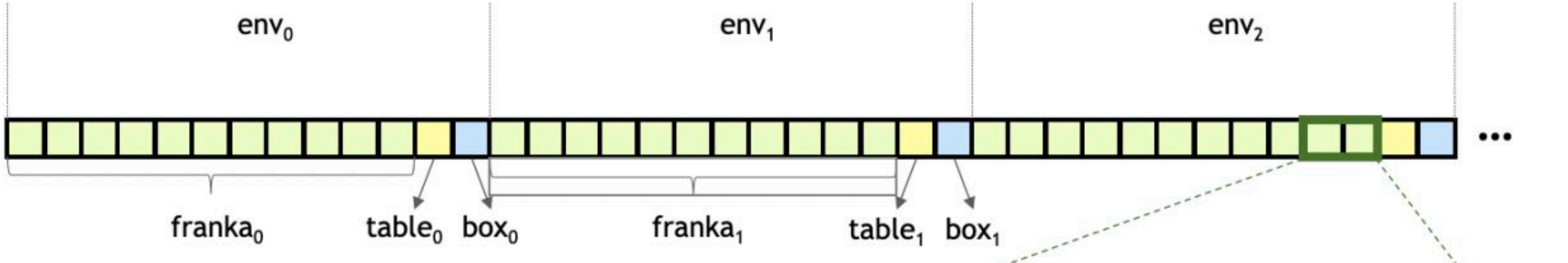
| Sensor | System | CNN | Agent Res. | RTX 3090 | RTX 2080Ti | Tesla V100 | 8×2080Ti | 8×V100 |
|---|---|---|---|---|---|---|---|---|
| Depth | BPS | SE-ResNet9 | 64 | 19900 | 12900 | 12600 | 72000 | 46900 |
| | BPS-R50 | ResNet50 | 128 | 2300 | 1400 | 2500 | 10800 | 18400 |
| | WIJMANS++ | SE-ResNet9 | 64 | 2800 | 2800 | 2100 | 9300 | 13100 |
| | WIJMANS20 | ResNet50 | 128 | 180 | 230 | 200 | 1600 | 1360 |
| RGB | BPS | SE-ResNet9 | 64 | 13300 | 8400 | 9000 | 43000 | 37800 |
| | BPS-R50 | ResNet50 | 128 | 2000 | 1050 | 2200 | 6800 | 14300 |
| | WIJMANS++ | SE-ResNet9 | 64 | 990 | 860 | 1500 | 4600 | 8400 |
| | WIJMANS20 | ResNet50 | 128 | 140 | OOM | 190 | OOM | 1320 |

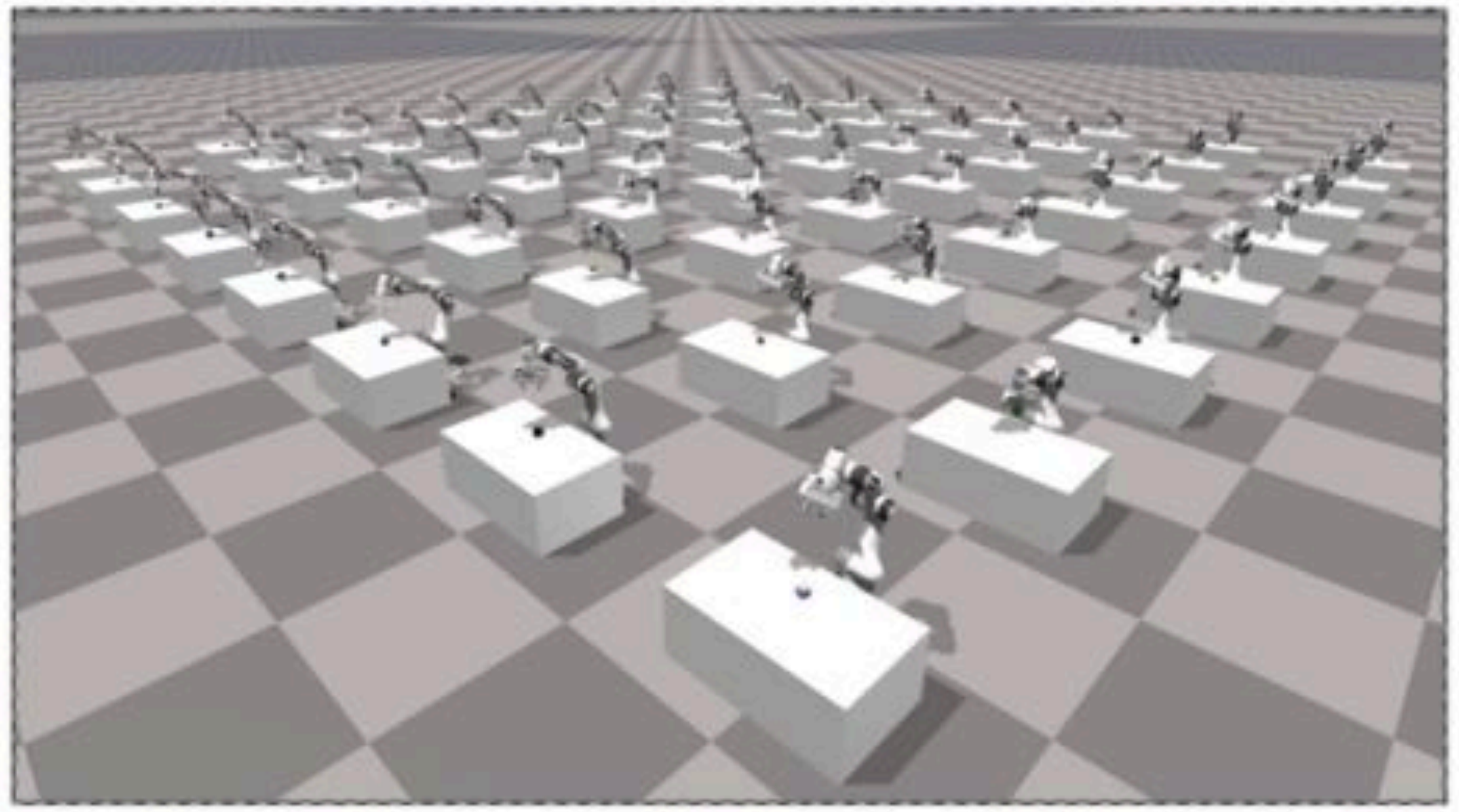


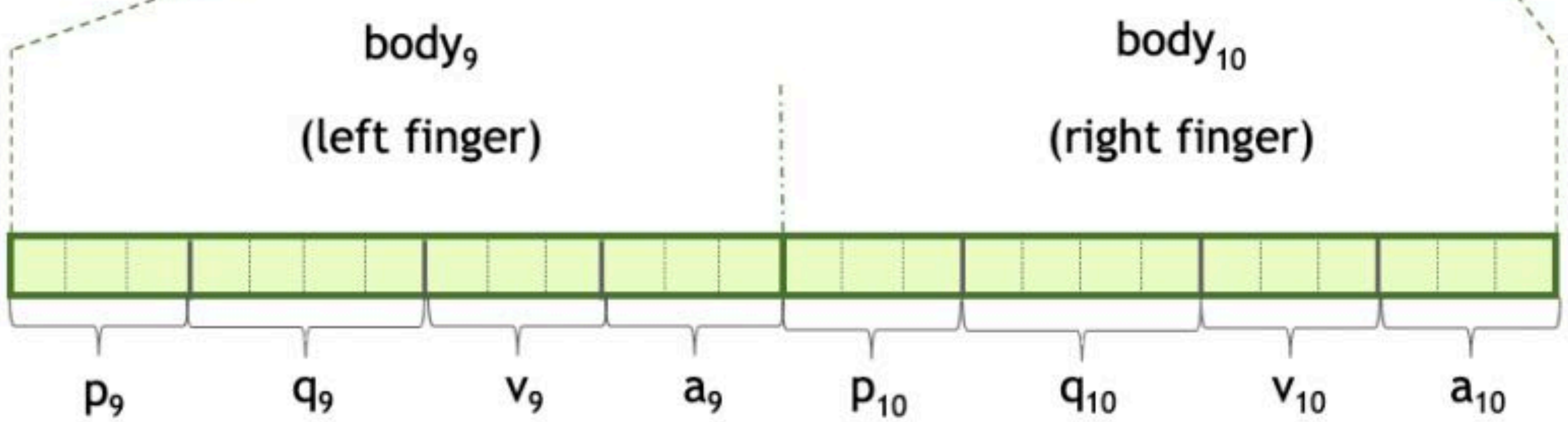

*** But low resolution: 64x64 rendered output resolution**

# NVIDIA Issac Gym

- Same idea of batched many-environment execution, applied to physics
- 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



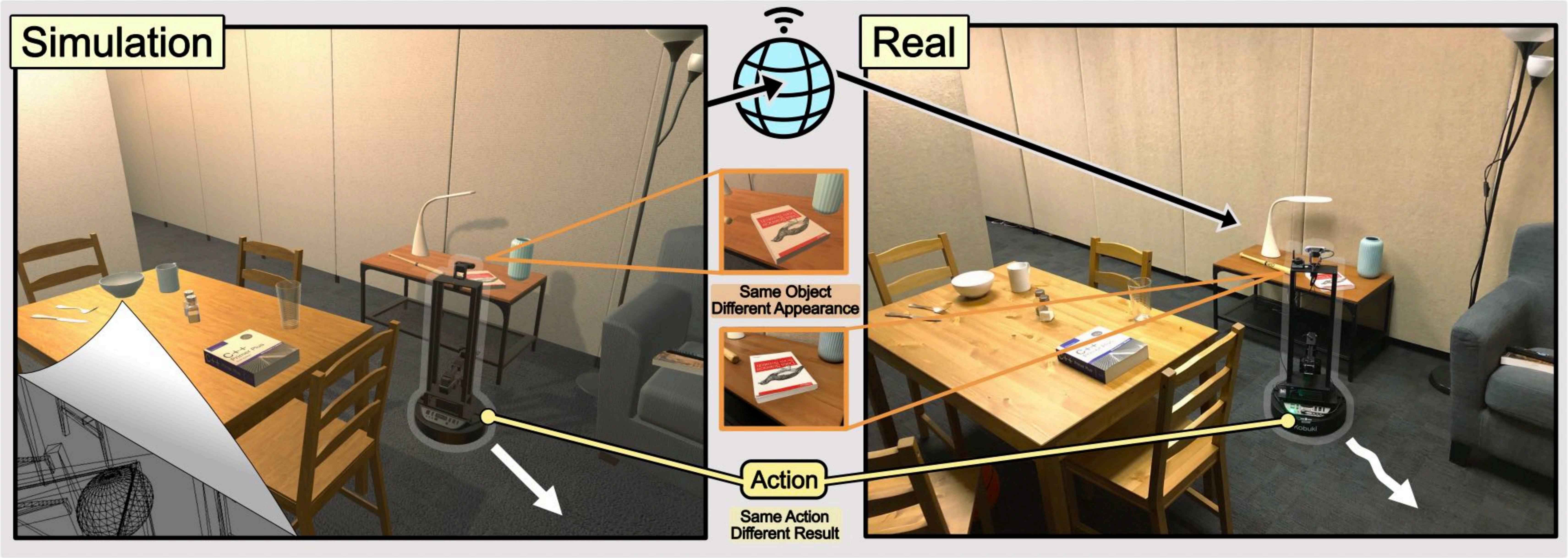


# Interesting rendering/simulation systems research questions

- If you had to design a rendering/simulation system "from the ground up" to support ML model training, what would you do differently from a modern high-performance game engine?
- What new opportunities for performance optimization are there? (amortize simulation and rendering across multiple virtual sensors, agents, etc.)
  - What should the architecture/API to the renderer be?
- How much fidelity is needed to train models that successfully transfer into the real-world?
  - Do we even need photorealistic quality (or advanced physics) to train policies that work in the real world?

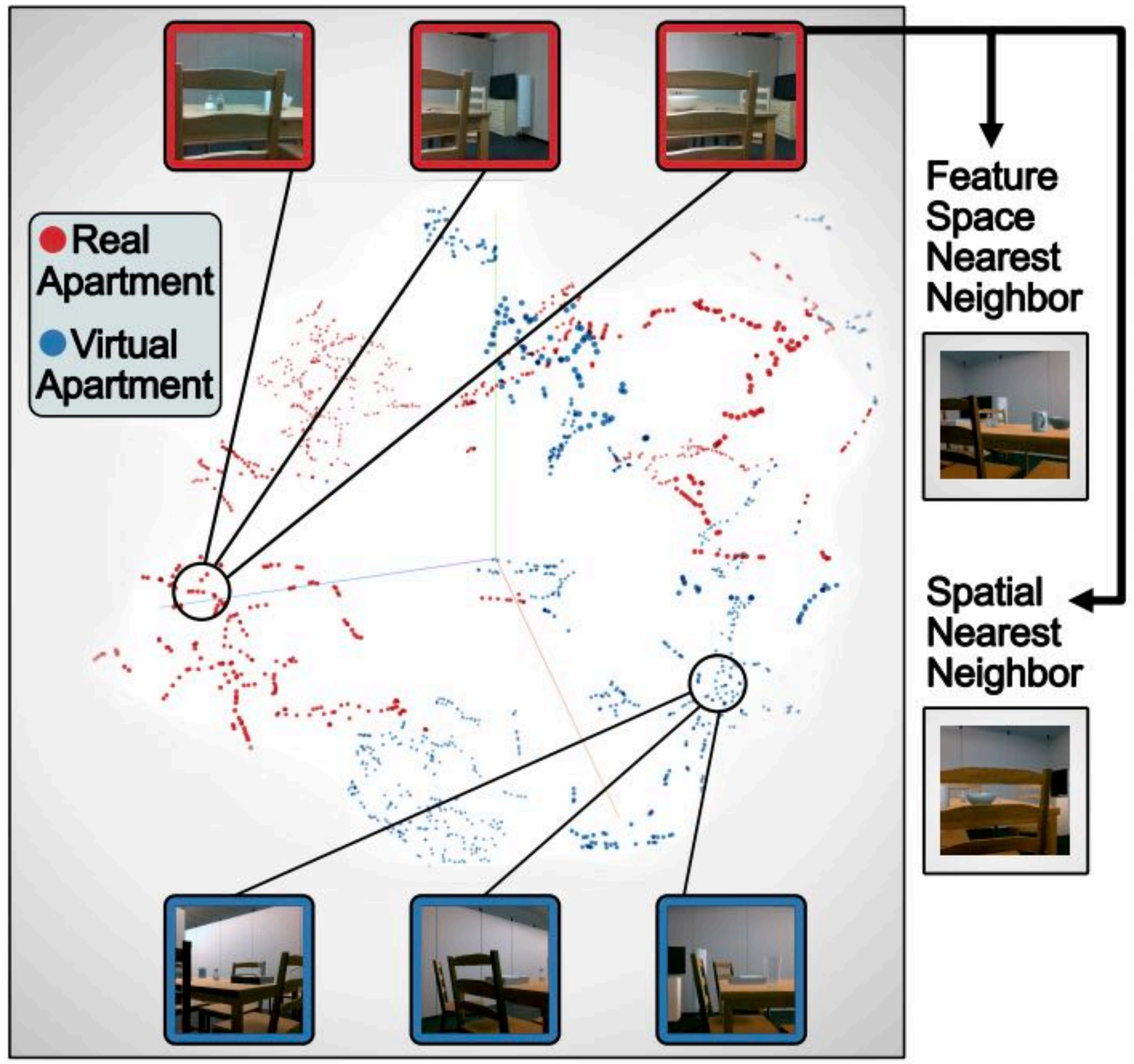
# Example Sim2Real experiments: RoboTHOR

[Dietke 20]




Virtual environment

Real world photo of corresponding environment (in lab)



# RobotTHOR: Sim2Real initial study

[Dietke 20]

| | Easy | | | | Medium | | | | Hard | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Success | SPL | Episode length | Path length | Success | SPL | Episode length | Path length | Success | SPL | Episode length | Path length |
| Random | 7.58 | 5.32 | 4.36 | 0.34 | 0.00 | 0.00 | 4.27 | 0.30 | 0.00 | 0.00 | 3.06 | 0.19 |
| Instant Done | 4.55 | 3.79 | 1.00 | 0.00 | 0.00 | 0.00 | 1.00 | 0.00 | 0.00 | 0.00 | 1.00 | 0.00 |
| Blind | 4.55 | 3.79 | 1.00 | 0.00 | 0.00 | 0.00 | 1.00 | 0.00 | 0.00 | 0.00 | 1.00 | 0.00 |
| Image | 55.30 | 38.12 | 45.87 | 9.26 | 28.79 | 19.12 | 78.49 | 14.82 | 1.47 | 0.97 | 81.09 | 14.22 |
| Image+Detection | 36.36 | 19.89 | 63.41 | 11.39 | 11.36 | 5.25 | 90.37 | 16.65 | 0.74 | 0.61 | 83.01 | 14.00 |

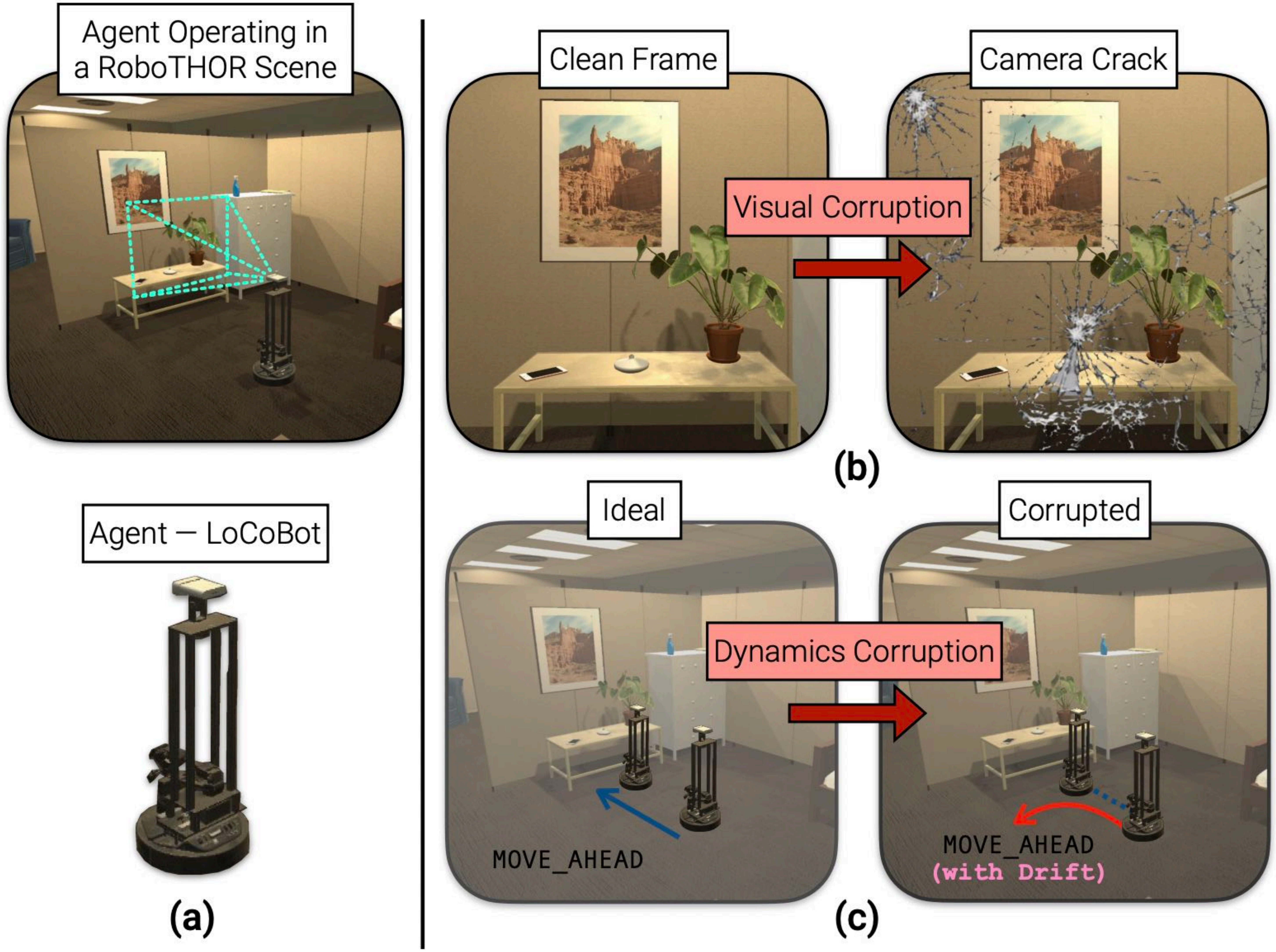
Table 1: **Benchmark results for Sim-to-Sim**

| | Easy | | | | Medium | | | | Hard | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Success | SPL | Episode length | Path length | Success | SPL | Episode length | Path length | Success | SPL | Episode length | Path length |
| Image | 33.33 | 3.53 | 53.16 | 7.18 | 16.66 | 3.70 | 43.83 | 5.33 | 0.00 | 0.00 | 67.83 | 7.00 |

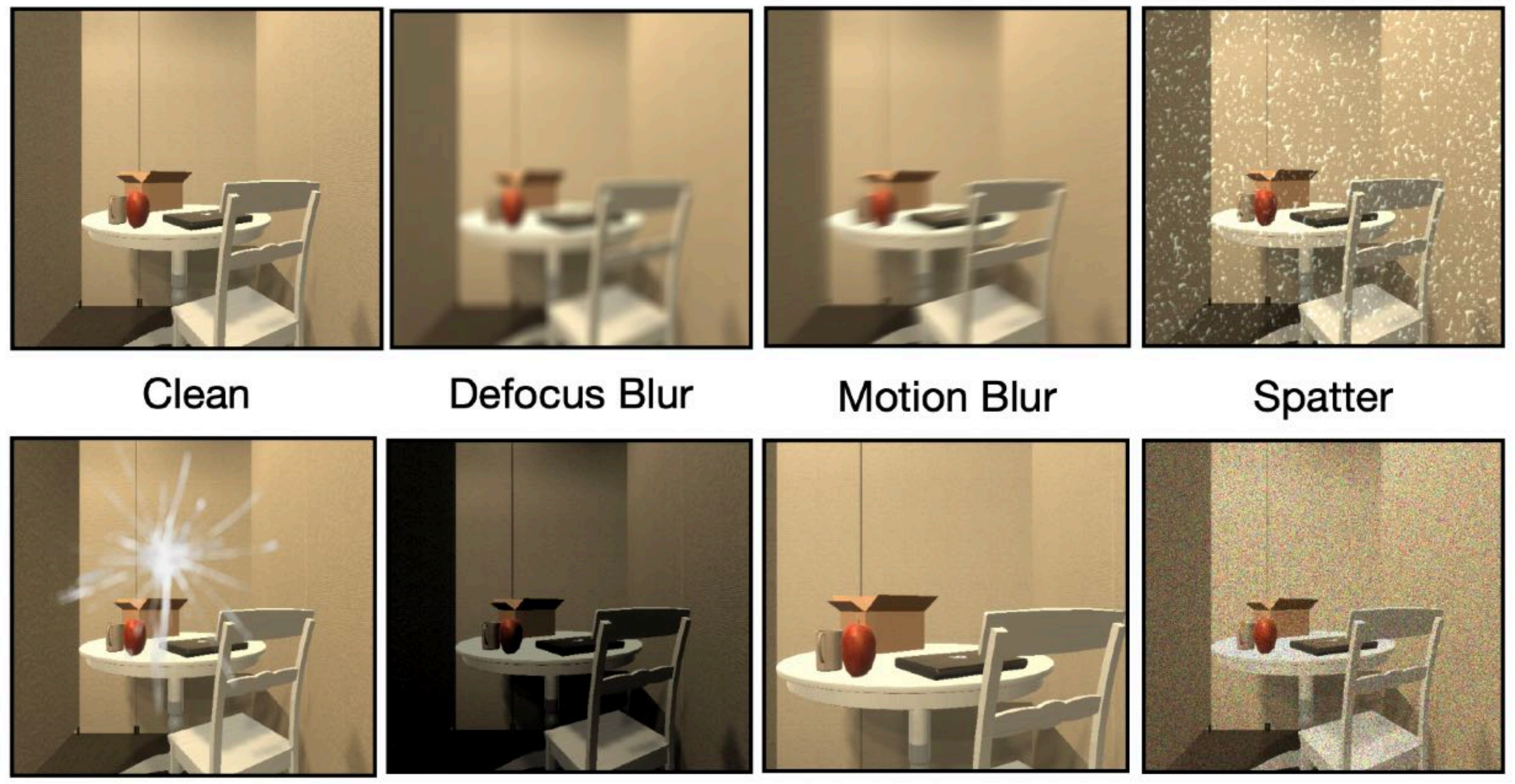
# Understanding the effects of sim2real gap

[Chattopadhyay 21]

**What parts of real-world sensing do we really need to model in simulation?**




**Example visual corruptions**



# Prep/background for next class

# Key parts of a shader

[Slide credits: Yong He]

# The rendering equation *

[Kajiya 86]

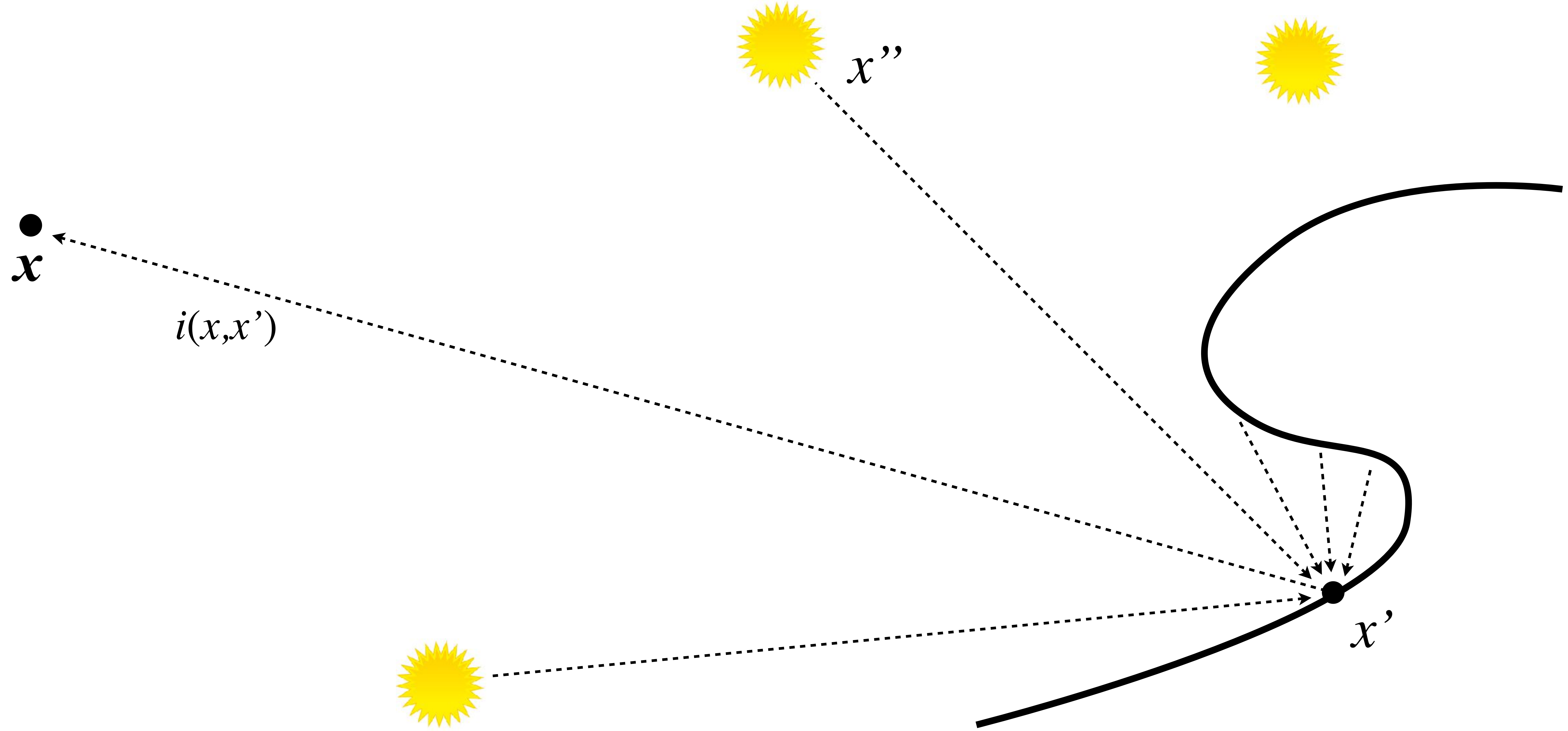
$$i(x,x') = v(x,x')\left[l(x,x') + \int r(x,x',x'')i(x',x'')dx''\right]$$

$i(x, x')$ = Radiance (light energy along a ray) from point x' in direction of point x

$v(x, x')$ = Binary visibility function (1 if ray from x' reaches x, 0 otherwise)

$l(x, x')$ = Radiance emitted from x' in direction of x (if x' is an emitter)

$r(x, x',x'')$ = BRDF: fraction of energy arriving at x' from x'' that is reflected in direction of x

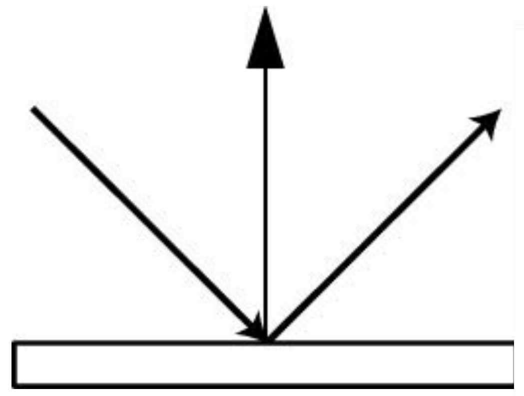



* Note: using notation from Hanrahan 90 (to match suggested reading)

# Categories of reflection functions: r(x,x′,x″)

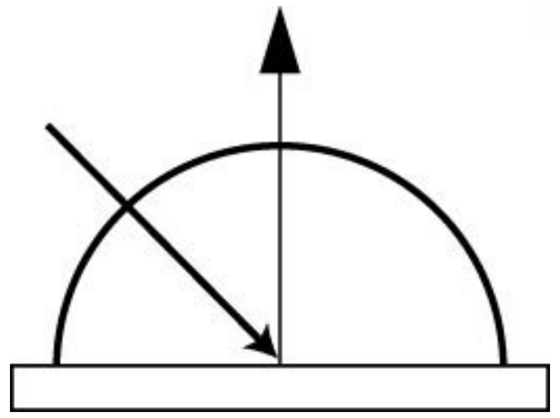
- **Ideal specular**

  Perfect mirror



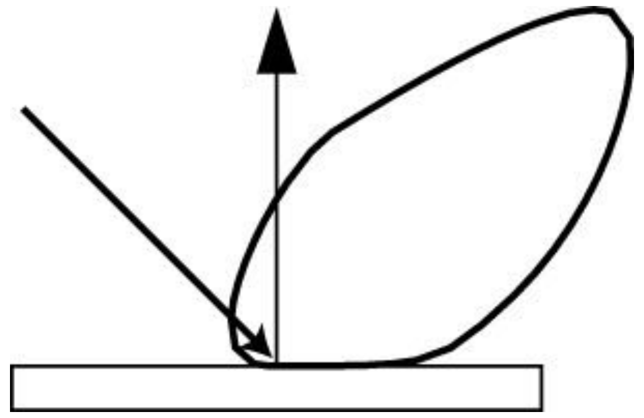
- **Ideal diffuse**

  Uniform reflection in all directions



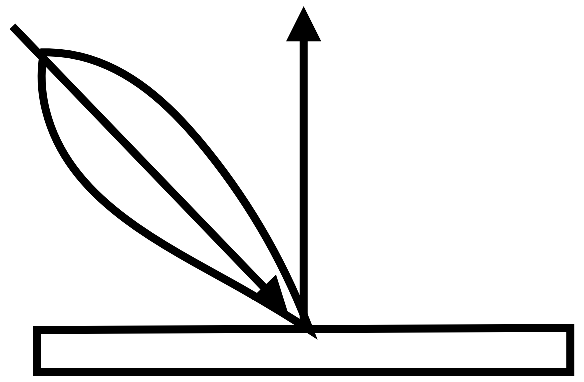

- **Glossy specular**

  Majority of light distributed in reflection direction



- **Retro-reflective**

  Reflects light back toward source

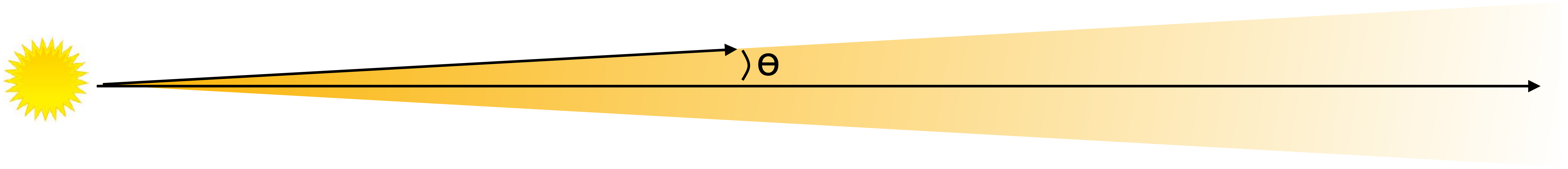



Diagrams illustrate how incoming light energy from given direction is reflected in various directions.

# Types of lights

- **Attenuated omnidirectional point light**
  (emits equally in all directions, intensity falls off with distance: $1/R^2$ falloff)

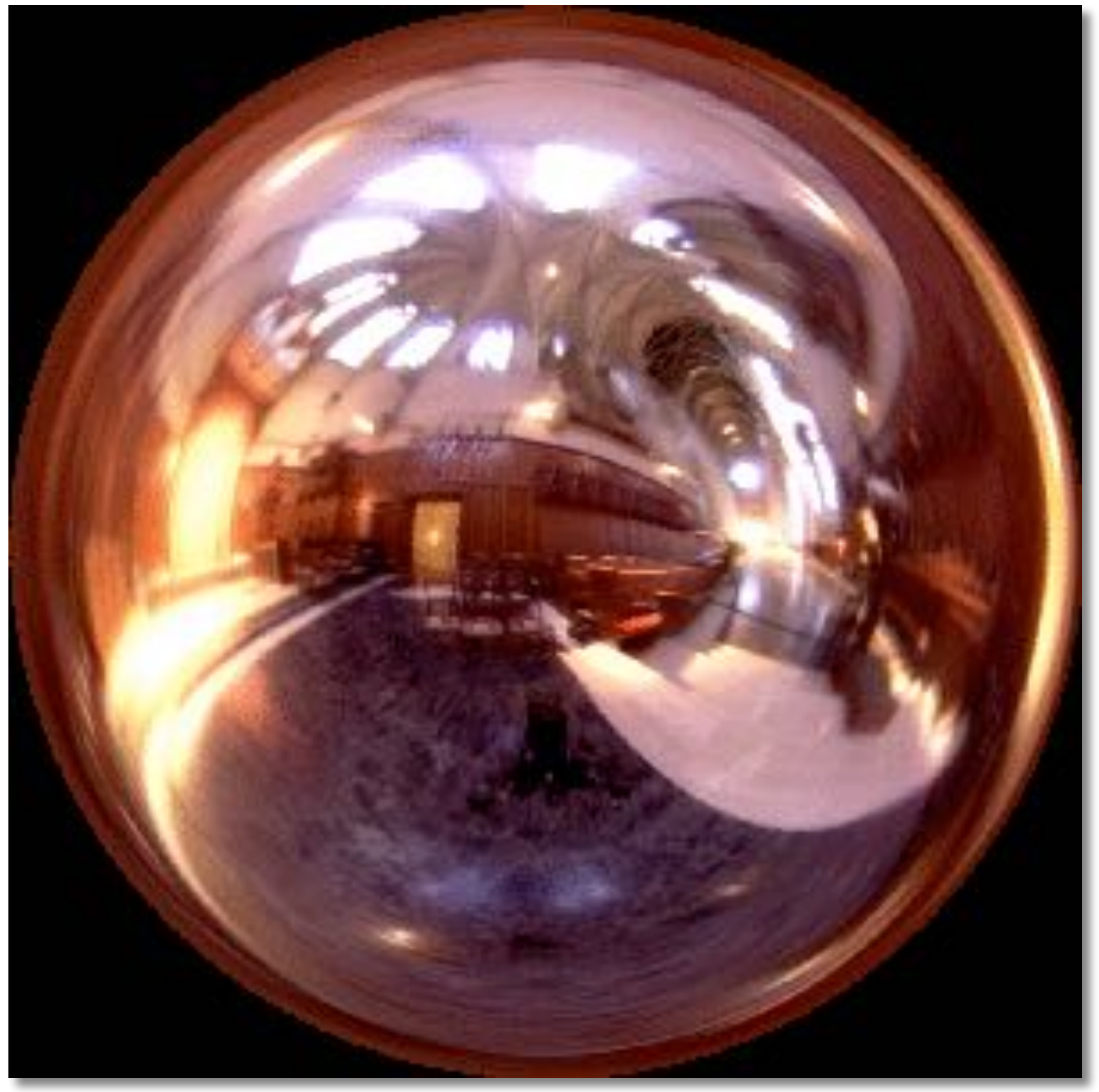
- **Spot light**
  (does not emit equally in all directions)



# More sophisticated lights

- **Environment light**

  **(not a point light source: defines incoming light from all directions)**



**Environment Map**
**(Grace cathedral)**

**Rendering using environment map**
**(pool balls have varying material properties)**
**[Ramamoorthi et al. 2001]**

# Environment map

Image credit: USC High-Resolution Light Probe Image Gallery

# The rendering equation *

[Kajiya 86]

Defined by material · Defined by lights

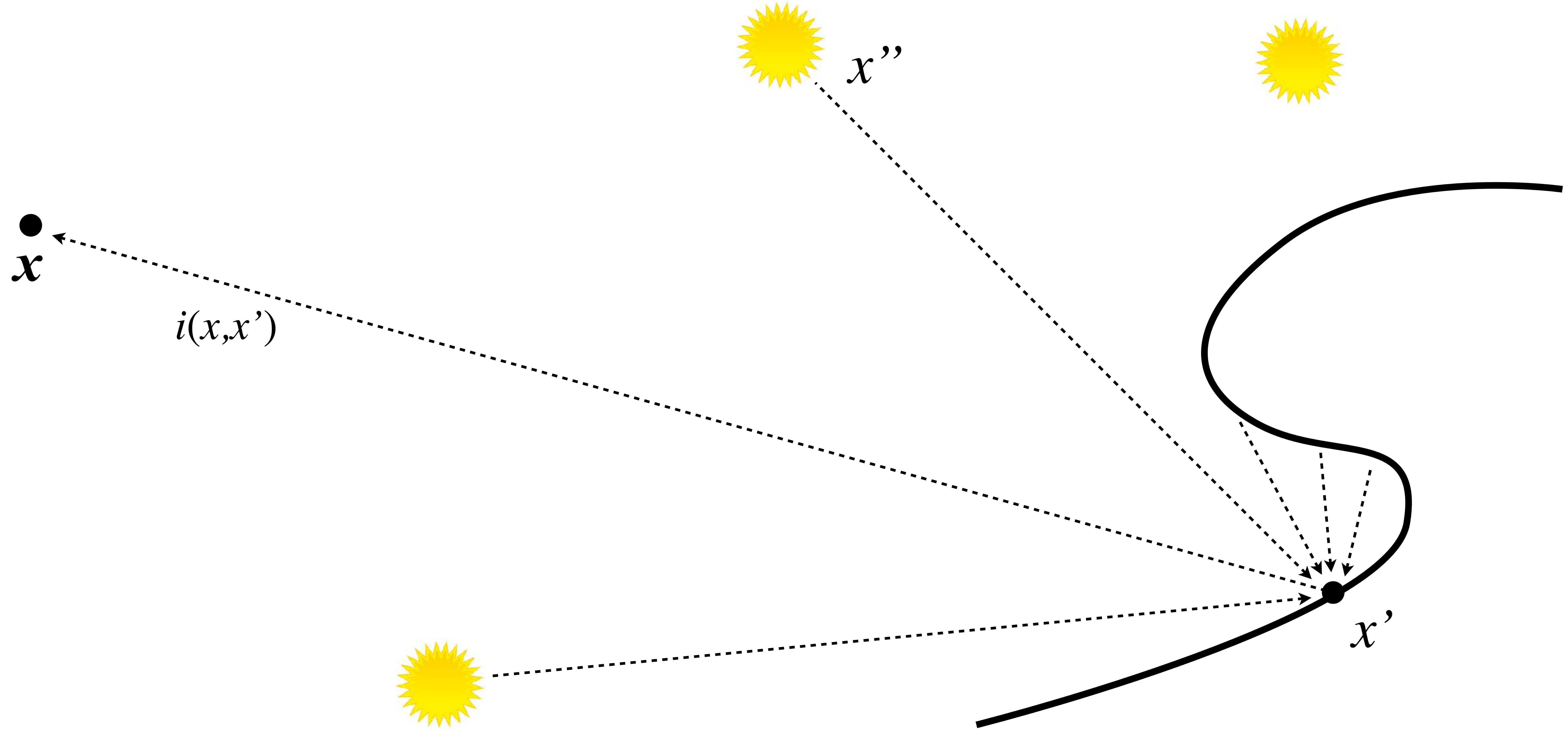
$$i(x,x') = v(x,x')\left[l(x,x') + \int r(x,x',x'')\, i(x',x'')\, dx''\right]$$

$i(x, x')$ = Radiance (light energy along a ray) from point x' in direction of point x

$v(x, x')$ = Binary visibility function (1 if ray from x' reaches x, 0 otherwise)

$l(x, x')$ = Radiance emitted from x' in direction of x (if x' is an emitter)

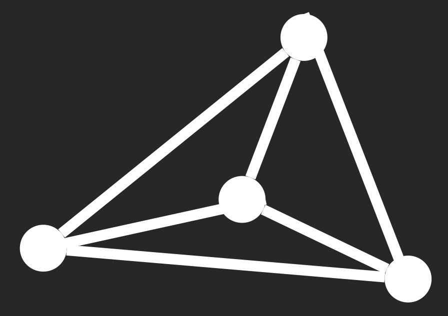
$r(x, x', x'')$ = BRDF: fraction of energy arriving at x' from x'' that is reflected in direction of x



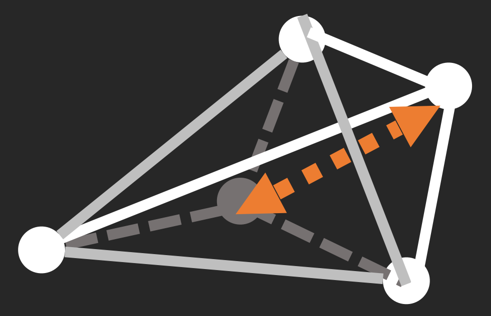

* Note: using notation from Hanrahan 90 (to match suggested reading)


Vertex Animation
Atmosphere Scattering
Geometry / Shader LOD
Sub-surface Scattering
Double-sided lighting
Pre-baked Lighting
Complex Wear Pattern
Layered Terrain Texturing
Skeletal Animated Character
Vegetation Instancing
Dynamic Soft Shadow
Epic Games, Inc.

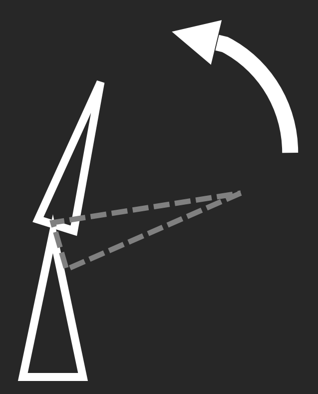
# Geometry / Animation



StaticMesh

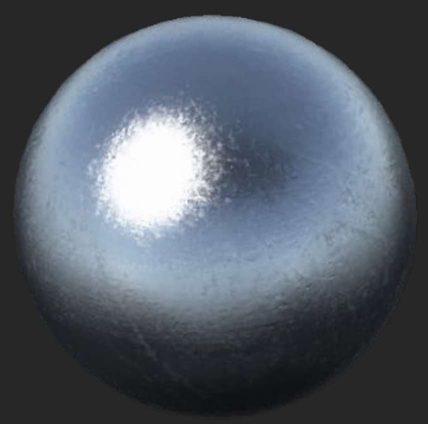


Displacement



SkeletalAnim

# Material



Metal

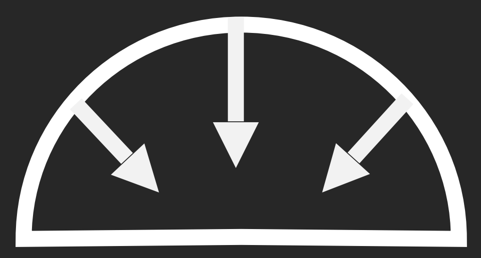
Cloth

Glass

# Light

SpotLight

PointLight



Skylight

# Geometry

Static Mesh

Skeletal Animated Mesh

# Materials

# Lighting

## Geometry / Animation

StaticMesh

Displacement

SkeletalAnim

## Material

Metal

Cloth

Glass

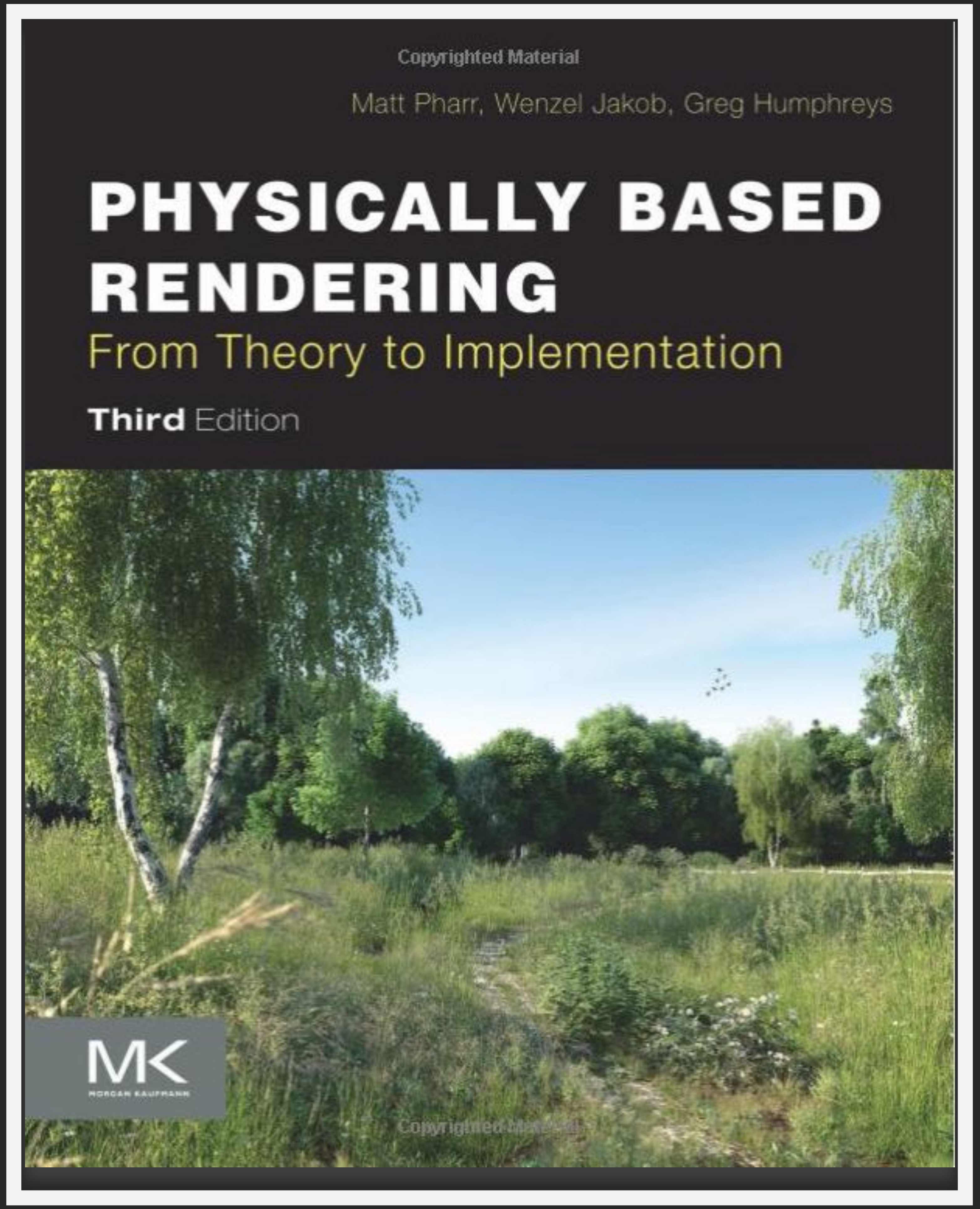
## Light

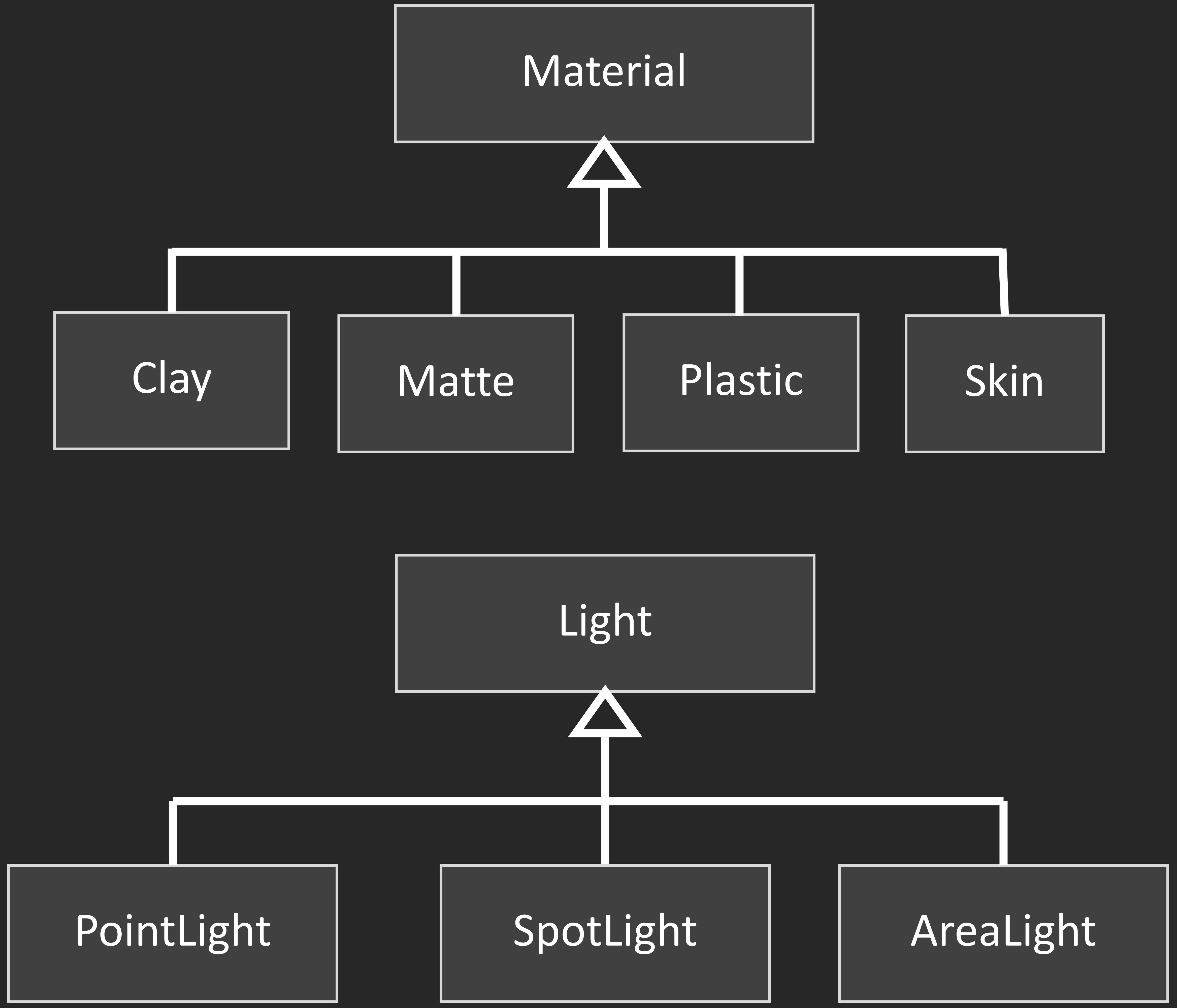
SpotLight

PointLight

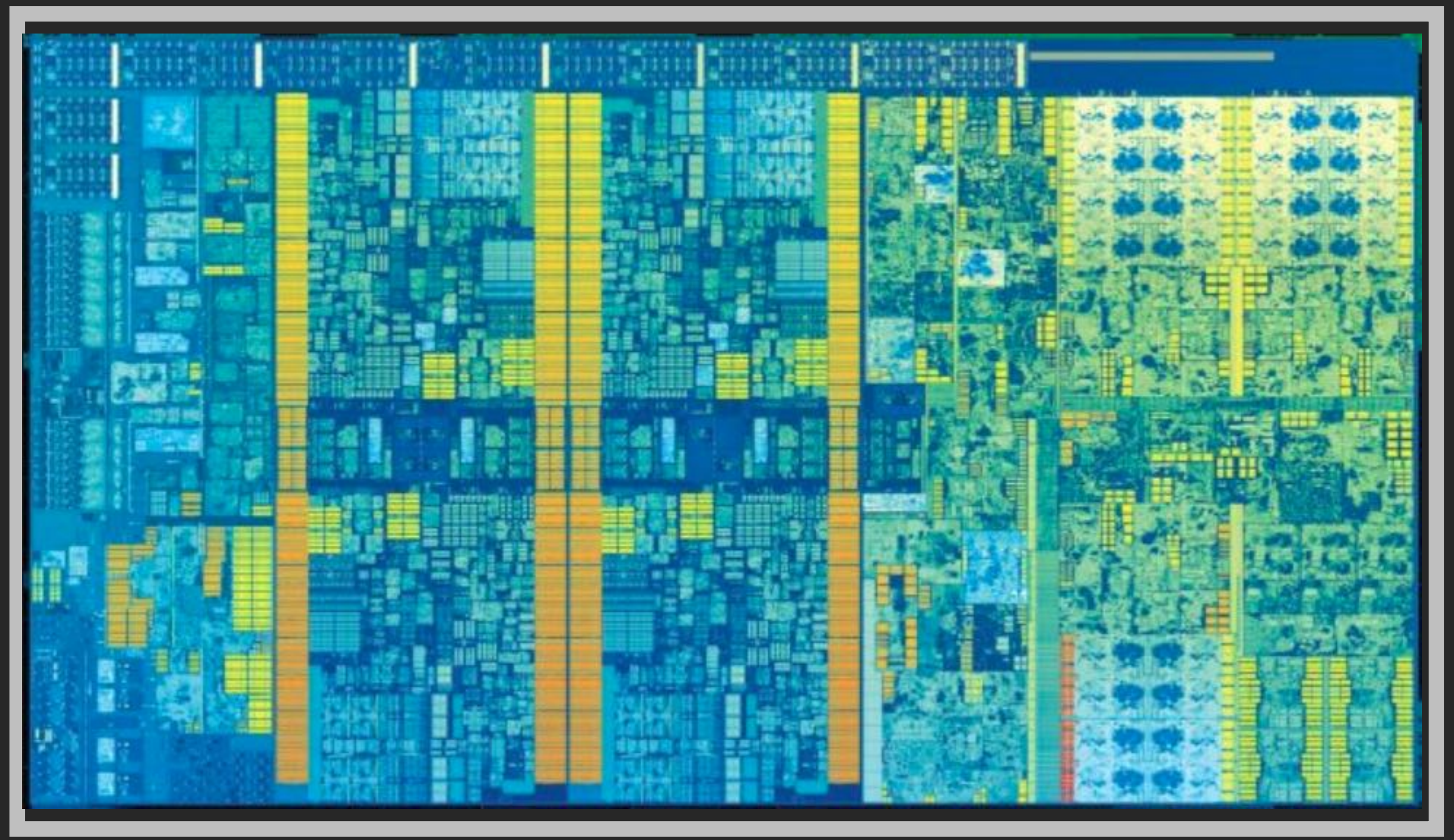
Skylight

# Extensibility is easy when performance is not a priority

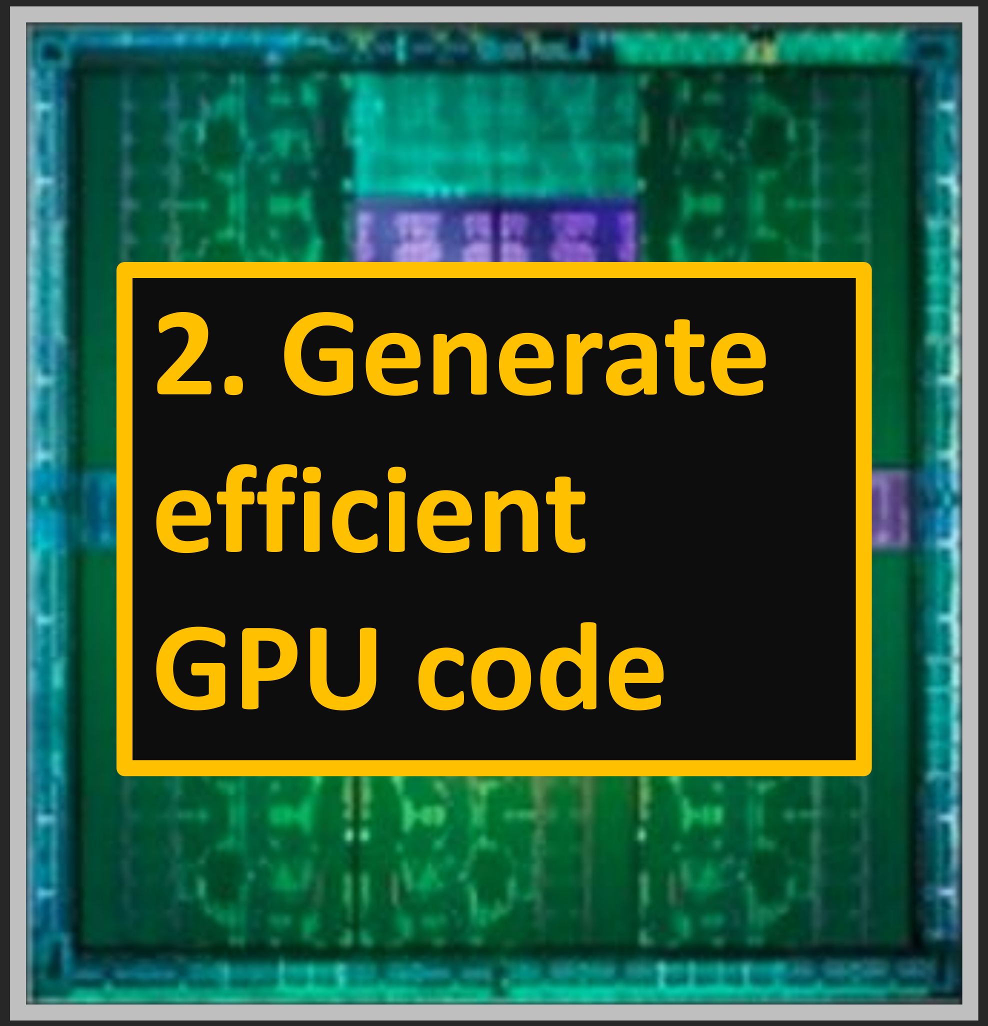


# Real-time renderers need to be efficient



**1. Efficient communication**




## Multi-core CPU

4-8 out-of-order execution cores

Managing Resources

Issuing Draw Commands to GPU

## GPU

Thousands of throughput-oriented cores

Executing Draw Commands

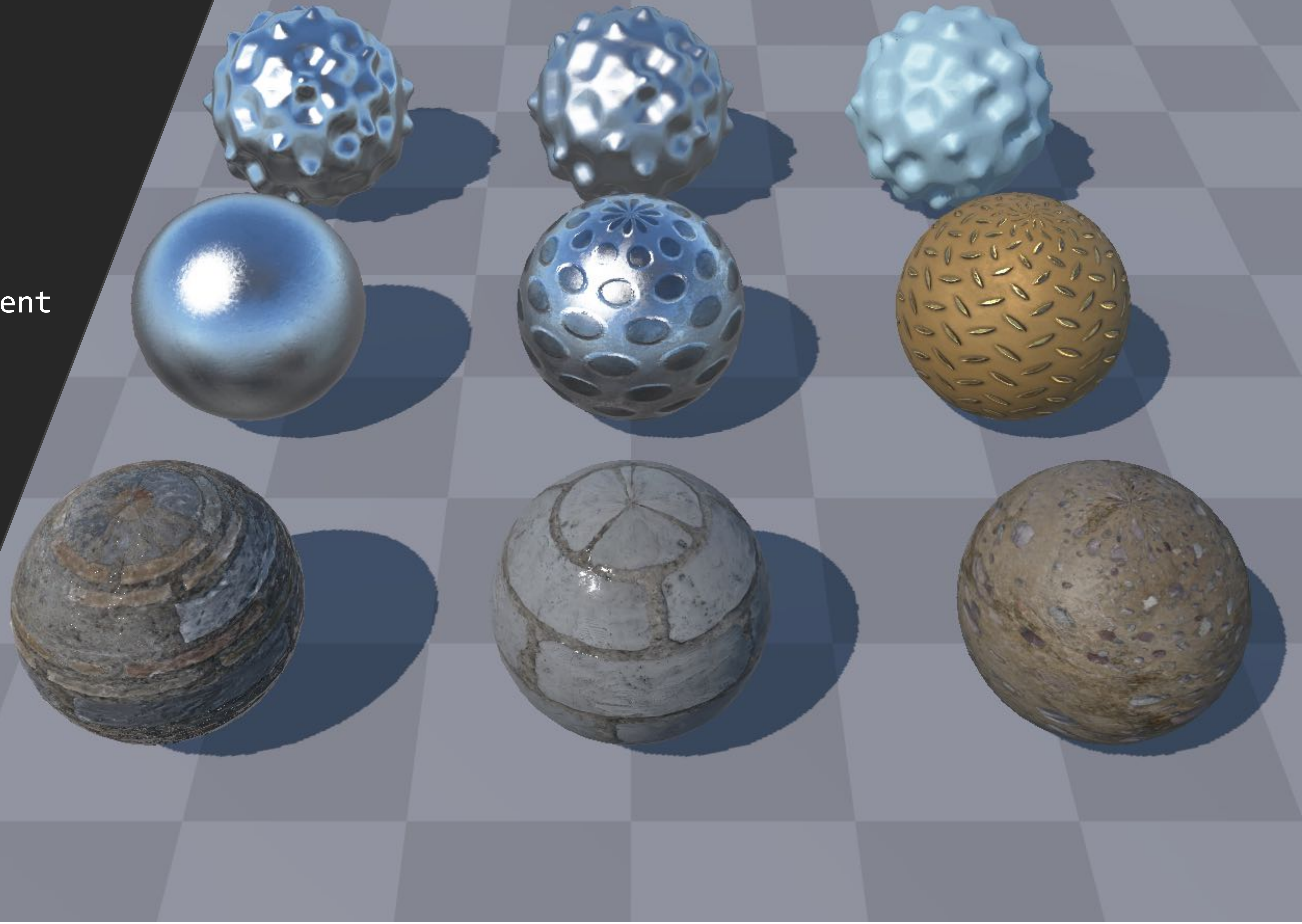
Evaluating Shading Features

# Shading System

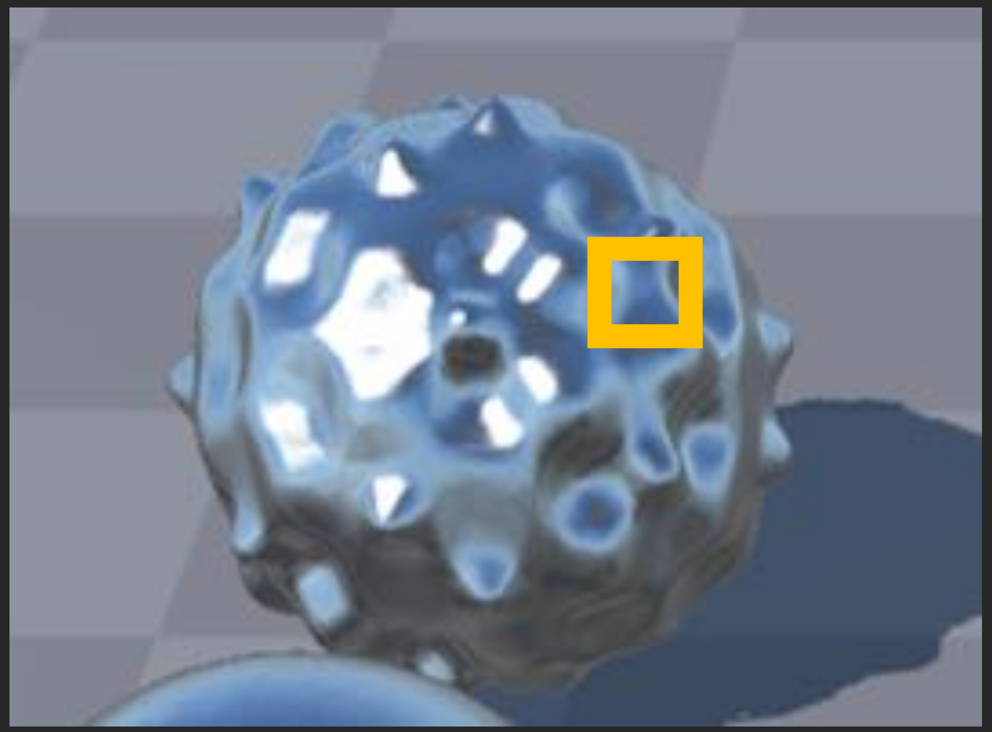
## Input

Many objects to render
Each has a set of features to use

```
obj0: Skylight, Metal, Displacement
obj1: Skylight, Metal
obj2: Skylight, Brick
obj3: Skylight, Dirt
…
```

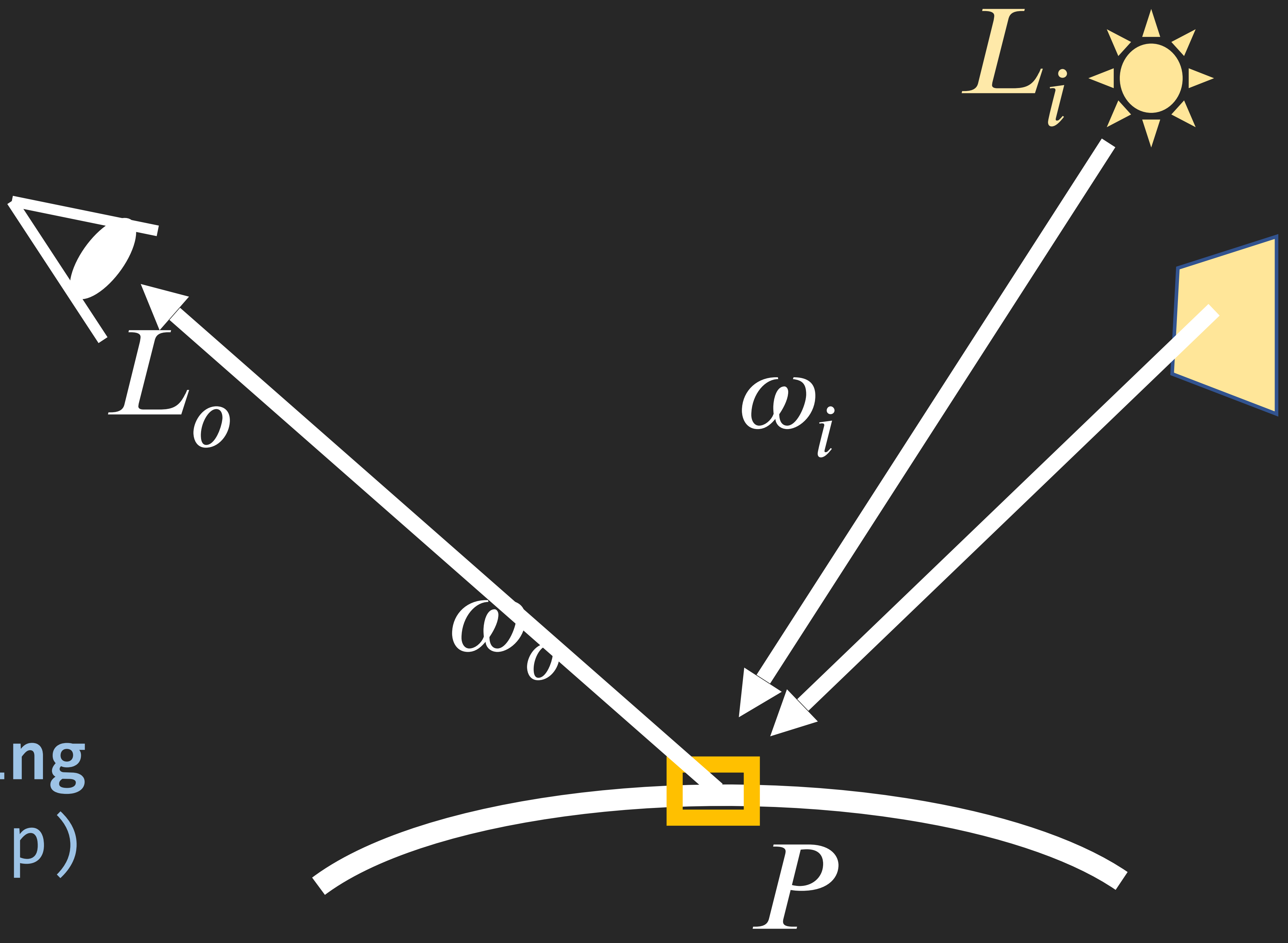


# The basic physics model that a shading system computes



bidirectional reflectance function (BxDF)

$$L_o = \sum_i L_i f(\omega_i, \omega_o)$$




**1. Material Shading**

```
f = evalMaterial(p)
```

**2. Light Shading**

```
Li, Wi = light[i].illum(p)
```

**3. Lighting Integration**

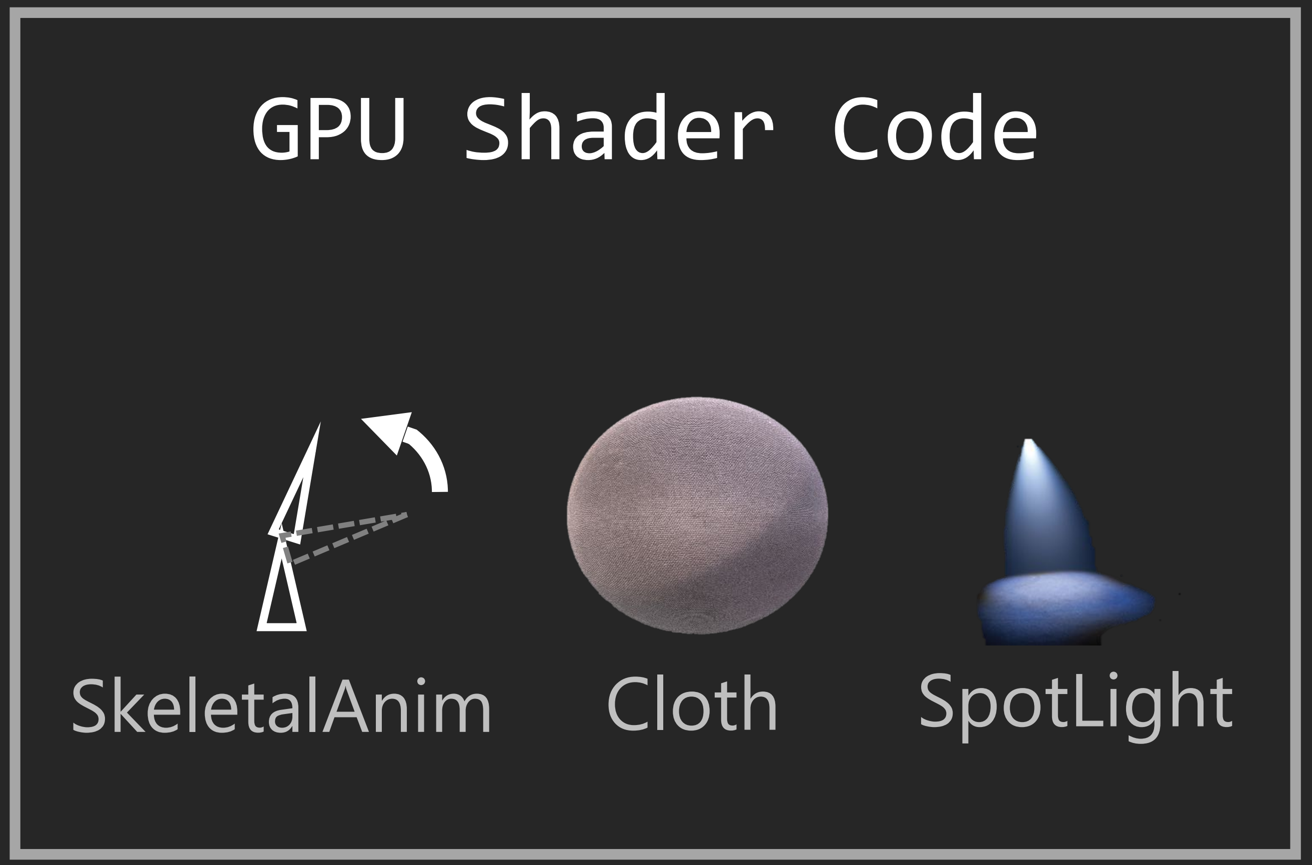
```
Lo = integrate(Li, f, Wi, Wo);
```


Skylight
lightProbe
strength
2.0
shadowMap
Displacement
displacementMap
normalMap
Metal Material
roughness
tint
[0.4 0.4 0.4]

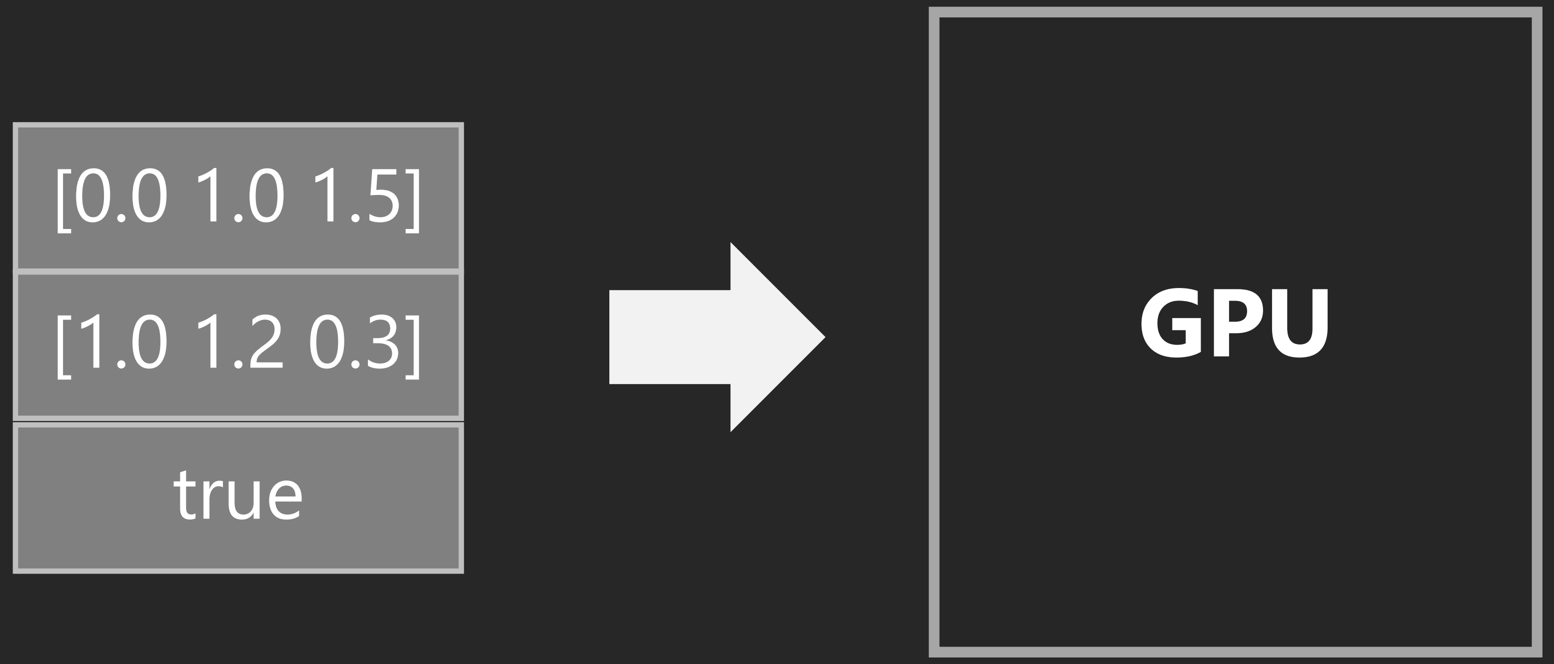

Skylight
lightProbe
strength
2.0
shadowMap
Brick Material
diffuse
tiling
[0.4]
uvOffset
[0.0, 0.0]

# A shading system does two things to draw an object

**1** Determine what code to run on current GPUs




**2** Communicate the parameters to the GPU



# Dynamically dispatch GPU code for shading features

Geometry

Material

Lighting

**Shader Code**

```
void myShader  (int geometryType, GP geomParams,
                int materialType, MP materialParams,
                int lightType, LP lightParams)
{
  if (geometryType == STATIC_MESH)
     computeStaticMeshGeometry(geomParams);
  else if (geometryType == DISPLACEMENT)
     computeDisplacementGeometry(geomParams);
  else if (geometryType == SKELETAL_ANIM)
     computeSkeletalAnimGeometry(geomParams);

  if (materialType == METAL)
     computeMetal(materialParams);
  else if (materialType == CLOTH)
     computeCloth(materialParams);
  else if (materialType == GLASS)
     computeGlass(materialParams);

  if (lightType == SPOT_LIGHT)
     computeSpotLight(lightParams);
  else if (lightType == POINT_LIGHT)
     computePointLight(lightParams);
  else if (lightType == SKY_LIGHT)
     computeSkyLight(lightParams);
}
```

# Dynamic dispatching is bad for performance

- Overhead of branching instructions on wide SIMD processors

**Shader Code**

```
void myShader  (int geometryType, GP geomParams,
                int materialType, MP materialParams,
                int lightType, LP lightParams)
{
   if (geometryType == STATIC_MESH)
      computeStaticMeshGeometry(geomParams);
   else if (geometryType == DISPLACEMENT)
      computeDisplacementGeometry(geomParams);
   else if (geometryType == SKELETAL_ANIM)
      computeSkeletalAnimGeometry(geomParams);

   if (materialType == METAL)
      computeMetal(materialParams);
   else if (materialType == CLOTH)
      computeCloth(materialParams);
   else if (materialType == GLASS)
      computeGlass(materialParams);

   if (lightType == SPOT_LIGHT)
      computeSpotLight(lightParams);
   else if (lightType == POINT_LIGHT)
      computePointLight(lightParams);
   else if (lightType == SKY_LIGHT)
      computeSkyLight(lightParams);
}
```

# Dynamic dispatching is bad for performance

- Overhead of branching instructions on wide SIMD processors
- Larger working set limits the ability of hardware multi-threading to hide memory latency

**Shader Code**

```
void myShader  (int geometryType, GP geomParams,
                int materialType, MP materialParams,
                int lightType, LP lightParams)
{
   if (geometryType == STATIC_MESH)
      computeStaticMeshGeometry(geomParams);
   else if (geometryType == DISPLACEMENT)
      computeDisplacementGeometry(geomParams);
   else if (geometryType == SKELETAL_ANIM)
      computeSkeletalAnimGeometry(geomParams);

   if (materialType == METAL)
      computeMetal(materialParams);
   else if (materialType == CLOTH)
      computeCloth(materialParams);
   else if (materialType == GLASS)
      computeGlass(materialParams);

   if (lightType == SPOT_LIGHT)
      computeSpotLight(lightParams);
   else if (lightType == POINT_LIGHT)
      computePointLight(lightParams);
   else if (lightType == SKY_LIGHT)
      computeSkyLight(lightParams);
}
```

# Common approach: specialize shader code for shading features in-use

```
compile myShader -D SKELETAL_ANIM, CLOTH, SPOT_LIGHT

draw(myShader, ...);
```

**Shader Code (using preprocessor directives)**

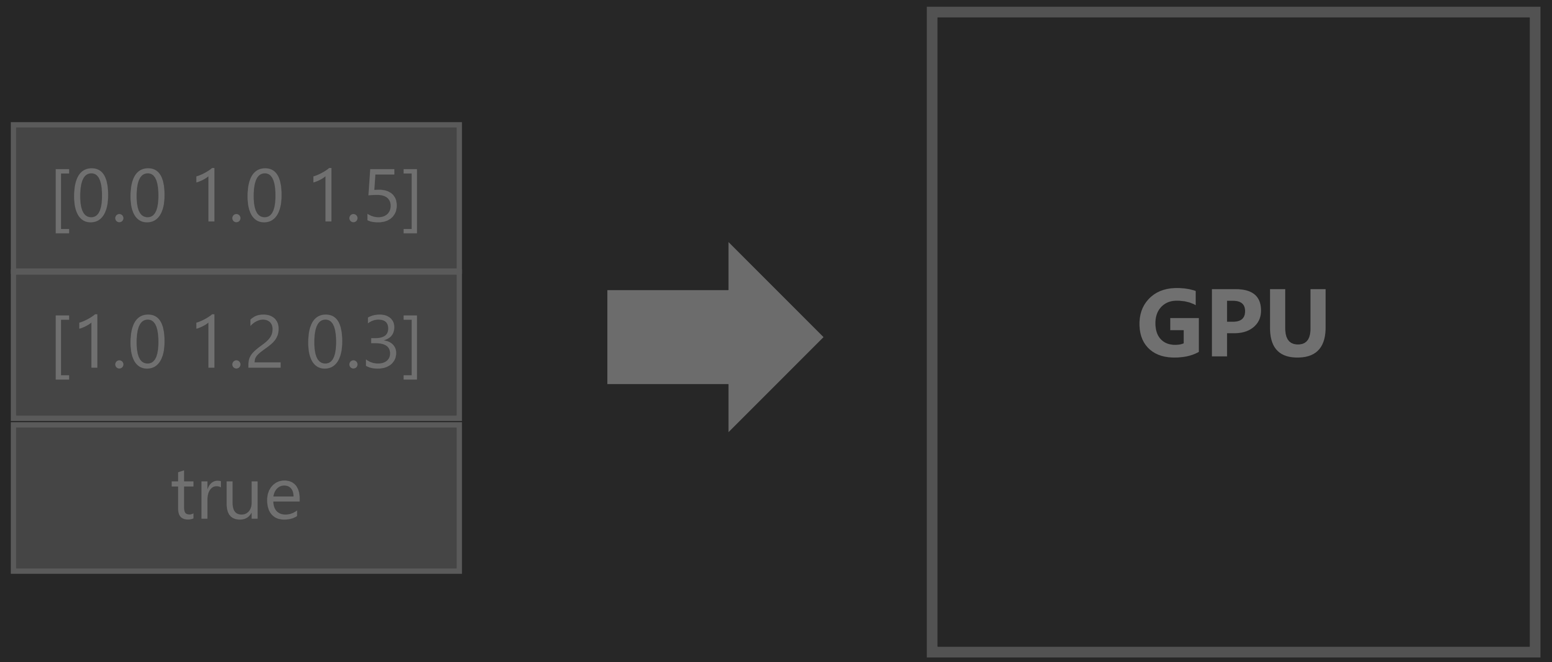
```
void myShader(...)
{
    #if defined(STATIC_MESH)
        computeStaticMeshGeometry(geomParams);
    #elif defined(DISPLACEMENT)
        computeDisplacementGeometry(geomParams);
    #elif defined(SKELETAL_ANIM)
        computeSkeletalAnimGeometry(geomParams);
    #endif
    #if defined(METAL)
        computeMetal(materialParams);
    #elif defined(CLOTH)
        computeCloth(materialParams);
    #elif defined(GLASS)
        computeGlass(materialParams);
    #endif
    #if defined(SPOT_LIGHT)
        computeSpotLight(lightParams);
    #elif defined(POINT_LIGHT)
        computePointLight(lightParams);
    #elif defined(SKY_LIGHT)
        computeSkyLight(lightParams);
    #endif
}
```

# A shading system does two things to draw an object

**1** Determine what code to run on current GPUs

GPU Shader Code


**2** Communicate the parameters to the GPU

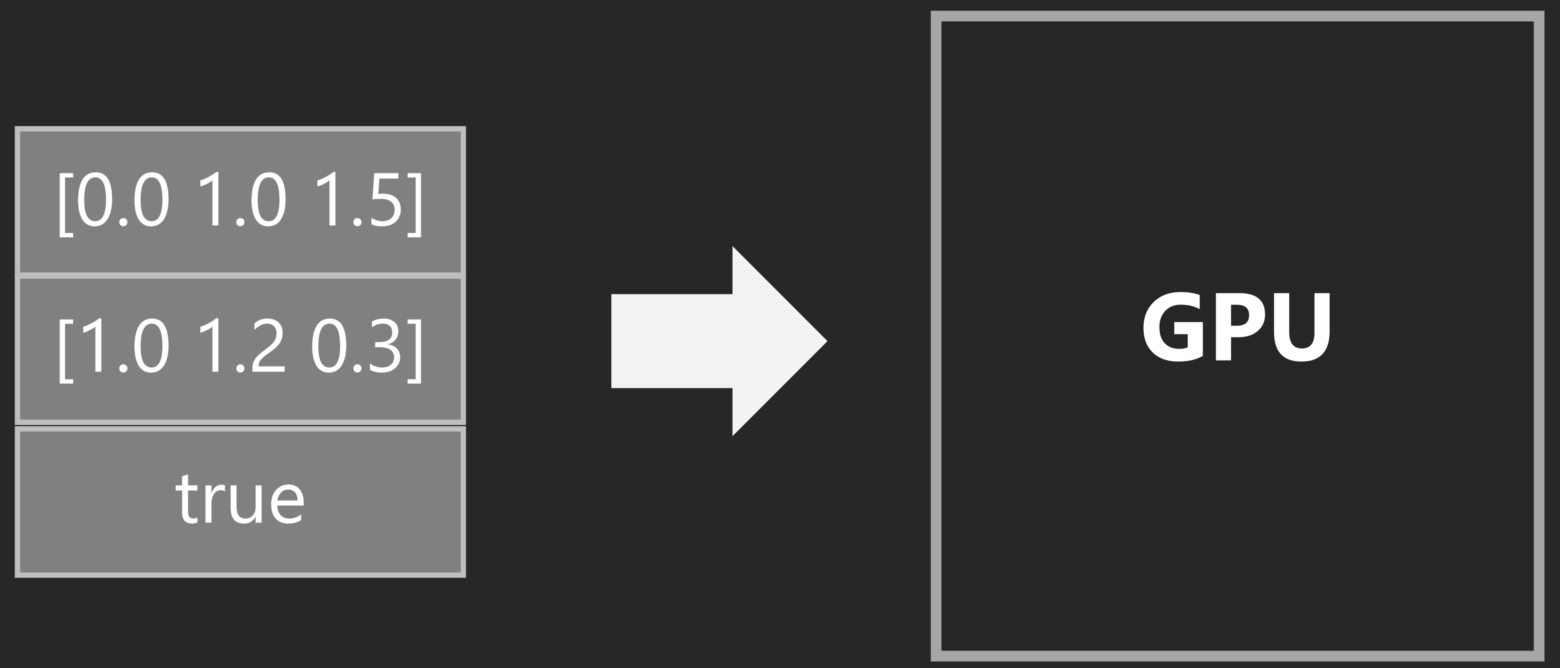


# A shading system does two things to draw an object

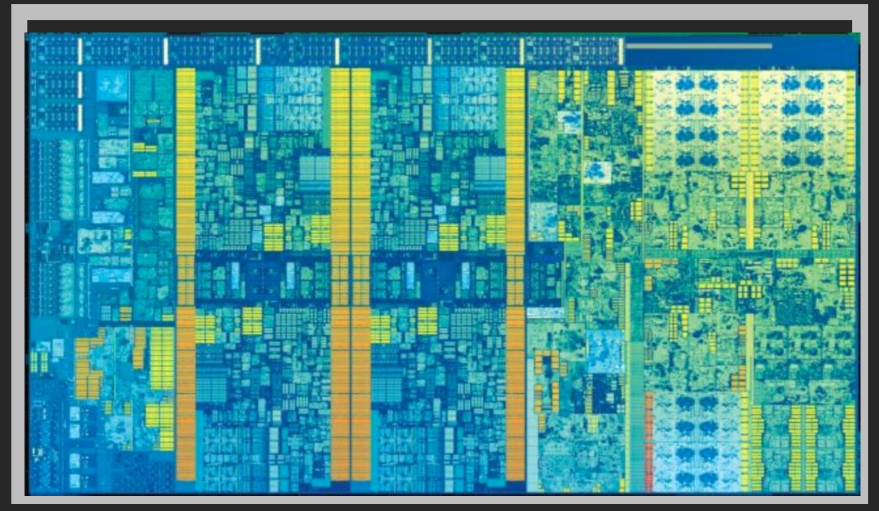
**1** Determine what code to run on current GPUs

GPU Shader Code

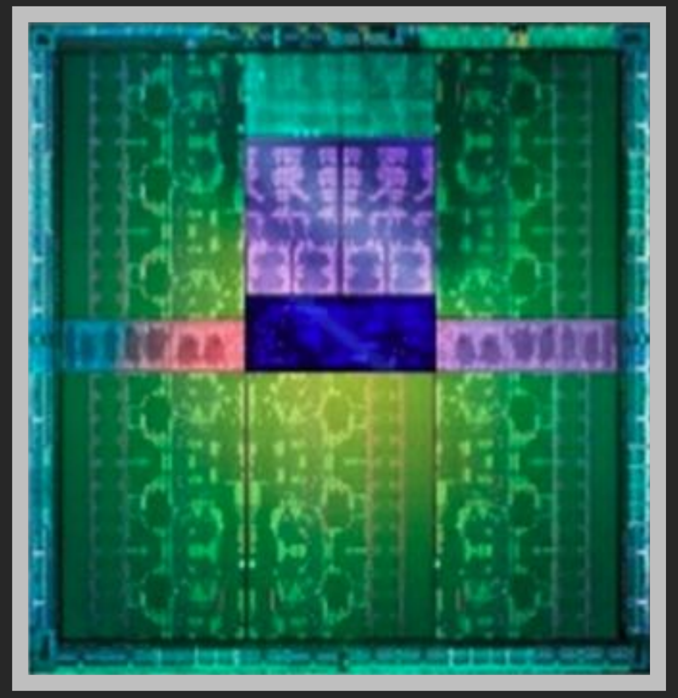

**2** Communicate the parameters to the GPU



# CPU-GPU communication model

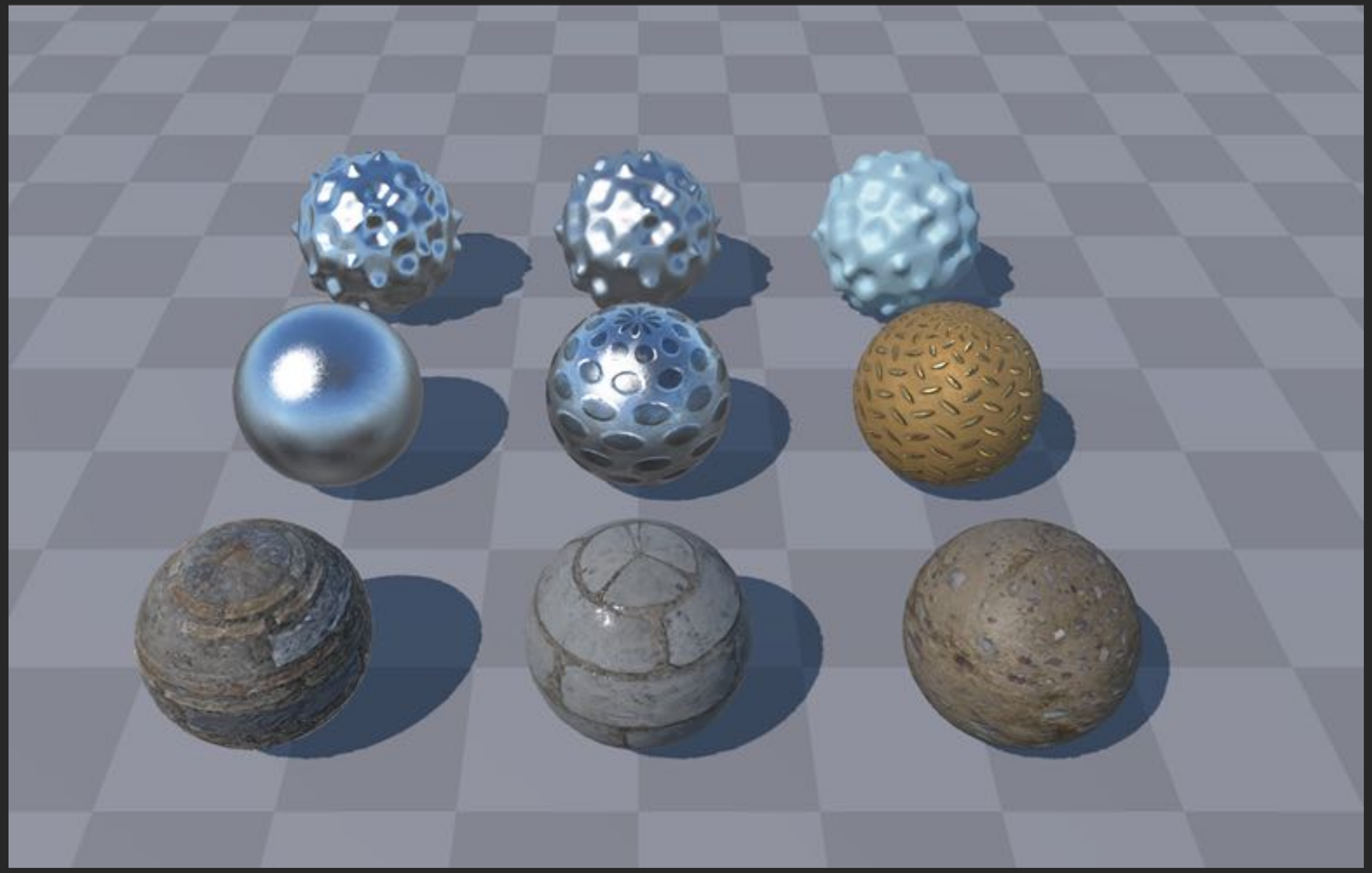


Draw(obj0) | SetParam(p5) | ... | SetParam(p1) | SetShader(s)



CPU

GPU



Skylight
lightProbe
strength
2.0
shadowMap
Displacement
displacementMap
normalMap
Metal Material
roughness
tint
[0.4 0.4 0.4]


Skylight
lightProbe
strength
2.0
shadowMap
Displacement
displacementMap
normalMap
Metal Material
roughness
tint
[0.5 0.5 0.5]


Block0
lightProbe
strength
2.0
shadowMap
displacementMap
normalMap
Block 2
roughness
tint
[0.5 0.5 0.5]
Block 1
roughness
tint
[0.4 0.4 0.4]


SetParamBlock(0, &block0)
SetParamBlock(1, &block1)
Draw(obj0)
Block0
lightProbe
strength 2.0
shadowMap
displacementMap
normalMap
Block 2
roughness
tint [0.5 0.5 0.5]
Block 1
roughness
tint [0.4 0.4 0.4]

SetParamBlock(0, &block0)
SetParamBlock(1, &block1)
Draw(obj0)
SetParamBlock(1, &block2)
Draw(obj1)
Block0
lightProbe
strength 2.0
shadowMap
displacementMap
normalMap
Block 2
roughness
tint [0.5 0.5 0.5]
Block 1
roughness
tint [0.4 0.4 0.4]

## Input

```
obj0: Skylight, Metal(p1),
      Displacement
obj1: Skylight, Metal(p2),
      Displacement
…
```

1. Include skylight feature in GPU code
2. Allocate and initialize parameter blocks
3. Ensure CPU and GPU agree on the parameter layout

## Specialize Shader Code

```
void entryPoint(
    @block0 geomParams,
    @block0 lightParams,
    @block1 materialParams)
{
   #if defined(STATIC_MESH)
    computeStaticMeshGeometry(geomParams);
   #elif defined(DISPLACEMENT)
      computeDisplacementGeometry(geomParams);
   #elif defined(SKELETAL_ANIM)
      computeSkeletalAnimGeometry(geomParams);

   #if defined(METAL)
      computeMetal(materialPar
   #elif defined(CLOTH)
      computeCloth(materialPar
   #elif defined(GLASS)
      computeGlass(materialPar

   #if defined(SPOT_LIGHT)
      computeSpotLight(lightParams);
   #elif defined(POINT_LIGHT)
      computePointLight(lightParams);
   #elif defined(SKY_LIGHT)
      computeSkyLight(lightParams);
}
```

```
struct SkylightParams
{
  TextureCube lightProbe;
  float strength;
  TextureCube shadowMap
};
```

lightParams

## Create Parameter Blocks

**Block0**

| | |
|---|---|
| lightProbe | |
| strength | 2.0 |
| shadowMap | |
| displacementMap | |
| normalMap | |

**Block 1**

| | |
|---|---|
| roughness | |
| tint | [0.4 0.4 0.4] |

**Block 2**

| | |
|---|---|
| roughness | |
| tint | [0.5 0.5 0.5] |

## Input

```
obj0: Skylight, Metal(p1),
      Displacement
obj1: Skylight, Metal(p2),
      Displacement
…
```

## Specialize Shader Code

```
void entryPoint(
   @block0 geomParams,
   @block0 lightParams,
   @block1 materialParams)
{
  #if defined(STATIC_MESH)
   computeStaticMeshGeometry(geomParams);
  #elif defined(DISPLACEMENT)
     computeDisplacementGeometry(geomParams);
  #elif defined(SKELETAL_ANIM)
     computeSkeletalAnimGeometry(geomParams);

  #if defined(METAL)
     computeMetal(materialPar
  #elif defined(CLOTH)
     computeCloth(materialPar
  #elif defined(GLASS)
     computeGlass(materialPar

  #if defined(SPOT_LIGHT)
     computeSpotLight(lightParams);
  #elif defined(POINT_LIGHT)
     computePointLight(lightParams);
  #elif defined(SKY_LIGHT)
     computeSkyLight(lightParams);
}
```

lightParams

```
struct SkylightParams
{
  TextureCube lightProbe;
  float strength;
  TextureCube shadowMap
};
```

## Create Parameter Blocks

**Block0**

| | |
|---|---|
| lightProbe | |
| strength | 2.0 |
| shadowMap | |
| displacementMap | |
| normalMap | |

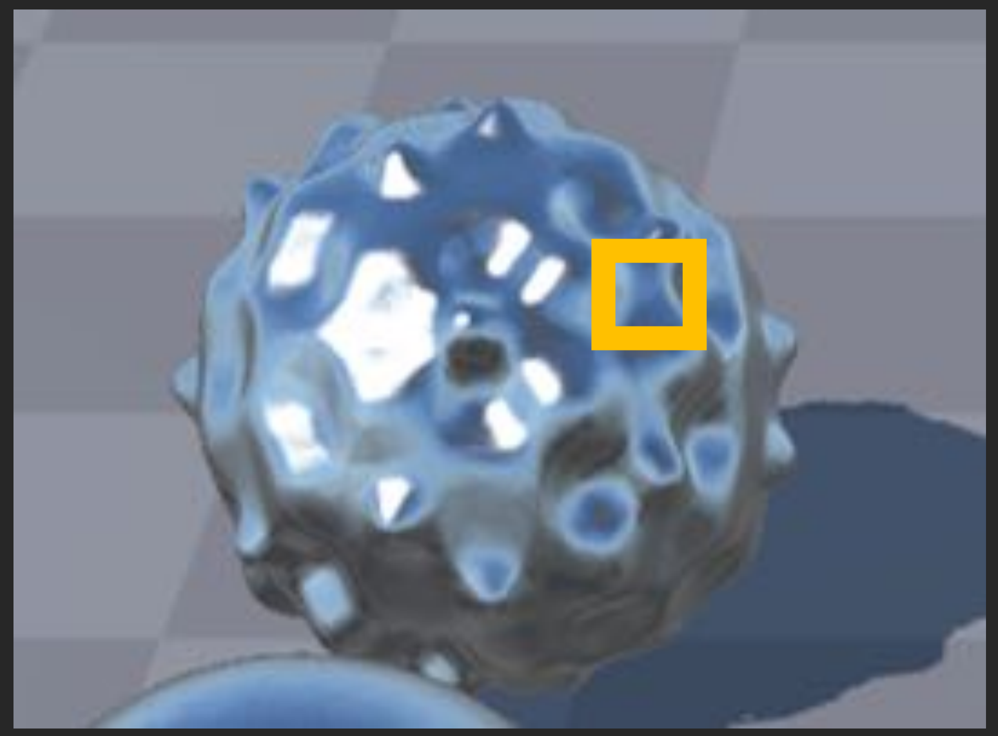
**Block 1**

| | |
|---|---|
| roughness | |
| tint | [0.4 0.4 0.4] |

**Block 2**

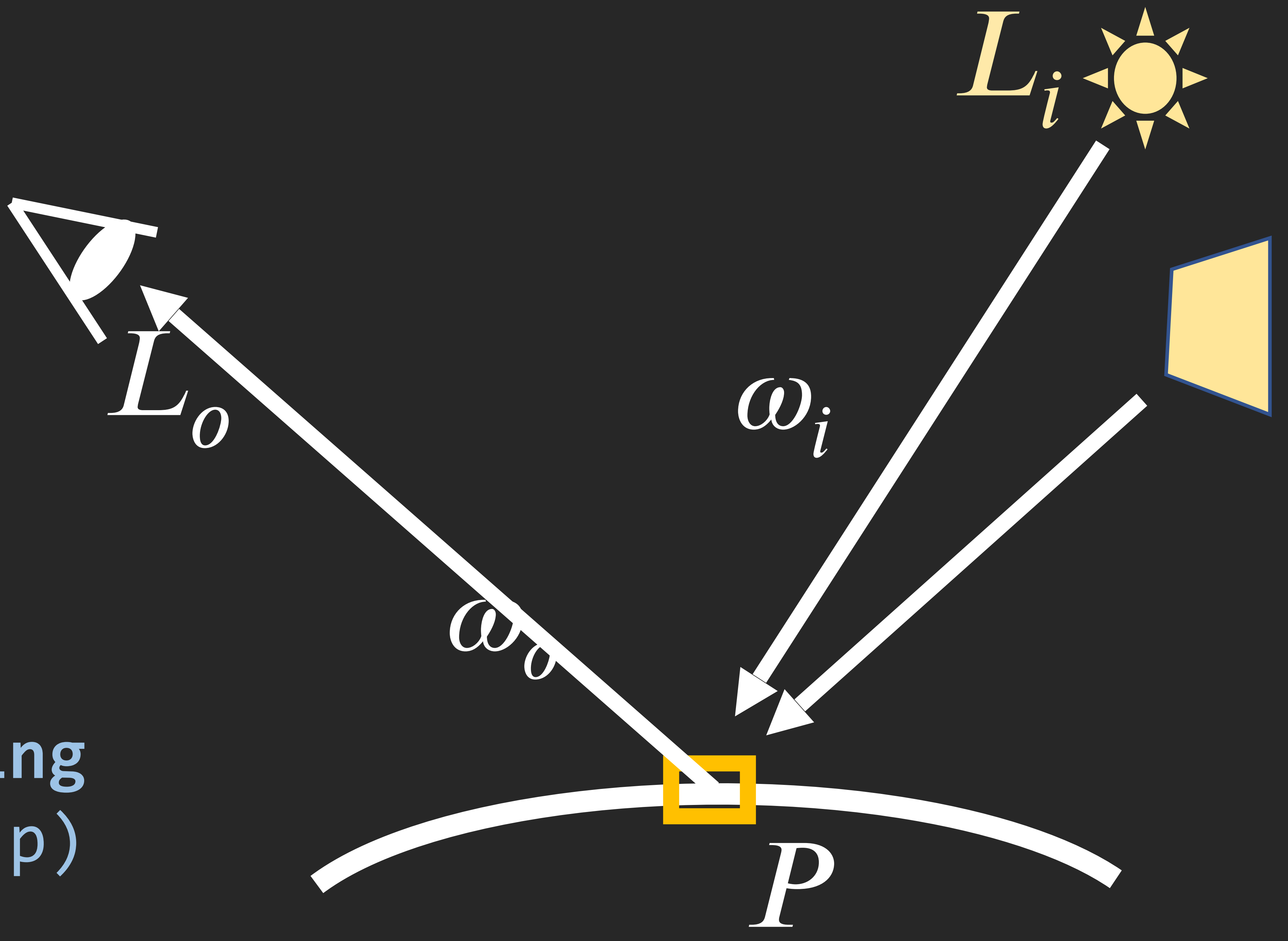
| | |
|---|---|
| roughness | |
| tint | [0.5 0.5 0.5] |

# Recall: basic physics model



bidirectional reflectance function (BxDF)

$$L_o = \sum_i L_i f(\omega_i, \omega_o)$$




1. Material Shading

```
f = evalMaterial(p)
```

2. Light Shading

```
Li, Wi = light[i].illum(p)
```

3. Lighting Integration

```
Lo = integrate(Li, f, Wi, Wo);
```

# Achieving modularity: implement shading features in separate files

**Materials**

- MetalMaterial.hlsl
- BrickMaterial.hlsl

**Lights**

**Light Integration**

```
struct MetalMaterial {…}
struct MetalBxDF {…}
MetalBxDF evalMaterial(MetalMaterial mat) {…}
float bxdf(MetalBxDF f) {…}
```

# Specialize shader by linking different files via #include

**MyShader_Variant1.hlsl**

```
#include "MetalMaterial.hlsl"
typedef MetalMaterial Material;
typedef MetalBxDF BxDF;
#include "LightEnv.hlsl"
#include "MyShader.hlsl"
```

**MetalMaterial.hlsl**

```
struct MetalMaterial {…}
struct MetalBxDF {…}
MetalBxDF evalMaterial(MetalMaterial mat) {…}
float bxdf(MetalBxDF f) {…}
```

**MyShader.hlsl**

```
float3 myShader(Material mat, LightEnv lightEnv)
{
    BxDF f = evalMaterial(mat);
    return evalLighting(lightEnv, f);
}
```

# Specialize shader by linking different files via #include

**MyShader_Variant1.hlsl**

```
#include "MetalMaterial.hlsl"
typedef MetalMaterial Material;
typedef MetalBxDF BxDF;
#include "LightEnv.hlsl"
#include "MyShader.hlsl"
```

**MetalMaterial.hlsl**

```
struct MetalMaterial {…}
struct MetalBxDF {…}
MetalBxDF evalMaterial(MetalMaterial mat) {…}
float bxdf(MetalBxDF f) {…}
```

- **No compiler help to ensure correctness**
  Shader entry point is not checked until a specialized variant is compiled

**MyShader.hlsl**

```
float3 myShader(Material mat, LightEnv lightEnv)
{
    BxDF f = evalMaterial(mat);
    return evalLighting(lightEnv, f);
}
```

# Specialize shader by linking different files via #include

**MyShader_Variant1.hlsl**

```
#include "MetalMaterial.hlsl"
typedef MetalMaterial Material;
typedef MetalBxDF BxDF;
#include "LightEnv.hlsl"
#include "MyShader.hlsl"
```

**MetalMaterial.hlsl**

```
struct MetalMaterial {…}
struct MetalBxDF {…}
MetalBxDF evalMaterial(MetalMaterial mat) {…}
float bxdf(MetalBxDF f) {…}
```

**MyShader.hlsl**

```
float3 myShader(Material mat, LightEnv lightEnv)
{
    BxDF f = evalMaterial(mat);
    return evalLighting(lightEnv, f);
}
```

- **No compiler help to ensure correctness**
  Shader entry point is not checked until a specialized variant is compiled

- **Assumptions to make a valid entry point is never explicitly stated in code**
  What types and functions should I provide to implement a new material?

# Next time (Foley and He visiting from NVIDIA)

- **Can we do better?**
  - **Can we achieve modularity and type safety of modern languages**
  - **But retain the performance expectations of modern GPU code?**
    - **No overhead of dynamic dispatch / worst-cast thread register allocation**
    - **Efficient bulk CPU-GPU communication**