Schedule

Start	End	Section	Speaker
13:00	13:45	Overview of Embodied AI	Zhiwei Jia (video)
13:45	14:30	The Basic Frameworks and techniques for Embodied AI	Fanbo Xiang (in person)
14:30	15:15	Design Choices in Embodied AI Environments	Jiayuan Gu (video)
15:15	15:30	Break	
15:30	16:15	Experience and Practices to Debug Simulators	Fanbo Xiang (in person)
16:15	16:35	Real World Robotics and Sim2Real	Rui Chen (video)
16:35	17:00	Embodied AI Tasks in ManiSkill and Visual Learning Challenges	Fanbo Xiang (in person)


Angel Xuan Chang
Simon Fraser University


Rui Chen
Tsinghua University


Jiayuan Gu
UC San Diego


Zhiwei Jia
UC San Diego


Tongzhou Mu
UC San Diego


Yuzhe Qin
UC San Diego


Hao Su
UC San Diego


Xiaolong Wang
UC San Diego


Fanbo Xiang
UC San Diego

UC San Diego

清華大學
Tsinghua University

SFU SIMON FRASER UNIVERSITY

Building and Working in Environments for Embodied AI

CVPR 2022 Tutorial


Angel Xuan Chang
Simon Fraser University


Rui Chen
Tsinghua University


Jiayuan Gu
UC San Diego


Zhiwei Jia
UC San Diego


Tongzhou Mu
UC San Diego


Yuzhe Qin
UC San Diego


Hao Su
UC San Diego


Xiaolong Wang
UC San Diego


Fanbo Xiang
UC San Diego

Schedule

Start	End	Section	Speaker
13:00	13:45	Overview of Embodied AI	Zhiwei Jia (video)
13:45	14:30	The Basic Frameworks and techniques for Embodied AI	Fanbo Xiang (in person)
14:30	15:15	Design Choices in Embodied AI Environments	Jiayuan Gu (video)
15:15	15:30	Break	
15:30	16:15	Experience and Practices to Debug Simulators	Fanbo Xiang (in person)
16:15	16:35	Real World Robotics and Sim2Real	Rui Chen (video)
16:35	17:00	Embodied AI Tasks in ManiSkill and Visual Learning Challenges	Fanbo Xiang (in person)


Angel Xuan Chang
Simon Fraser University


Rui Chen
Tsinghua University


Jiayuan Gu
UC San Diego


Zhiwei Jia
UC San Diego


Tongzhou Mu
UC San Diego


Yuzhe Qin
UC San Diego


Hao Su
UC San Diego


Xiaolong Wang
UC San Diego


Fanbo Xiang
UC San Diego

The Basic Frameworks and Techniques for Embodied AI

Building and Working in Environments for Embodied AI (part II)

CVPR 2022 Tutorial

UC San Diego

清華大學 Tsinghua University

SFU SIMON FRASER UNIVERSITY

Overview

- We are going to talk about
    - How the embodied AI community models and solves problems
    - How simulators and environments are built
    - How to build your own environment

- This section is for all people who want to build or use embodied AI environments.

Outline

- Modeling and approaches for Embodied AI
- Simulation technology for Embodied AI
- Building an environment from scratch

Outline

- Modeling and approaches for Embodied AI
  - World model
  - Learning-based methods to solve tasks
  - Classic robotics

- Simulation technology for Embodied AI
- Building an environment from scratch

Outline

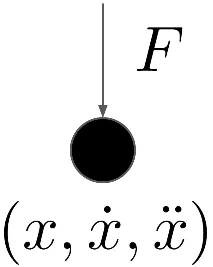
- Modeling and approaches for Embodied AI
  - World model
  - Learning-based methods to solve tasks
  - Classic robotics
- Simulation technology for Embodied AI
- Building an environment from scratch

How do People Model the World?

x Position

$\dot{x} := dx/dt$ Velocity

$\ddot{x} := d\dot{x}/dt$ Acceleration



Newton’s second law of motion

$$F = m\ddot{x}$$

How do People Model the World?

If we call $(x, \dot{x})$ the **state** of the world,

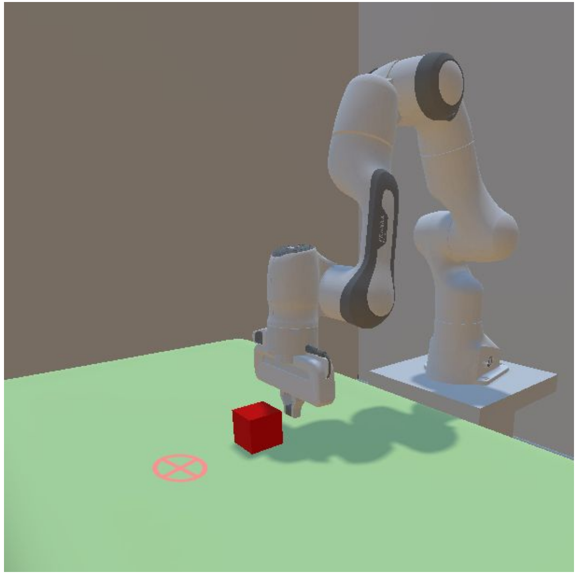
And F an **action** on the world.

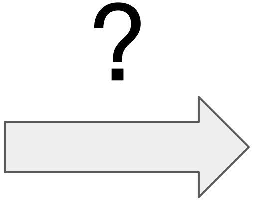
Newton's second law models the **transition** of state under action over time.

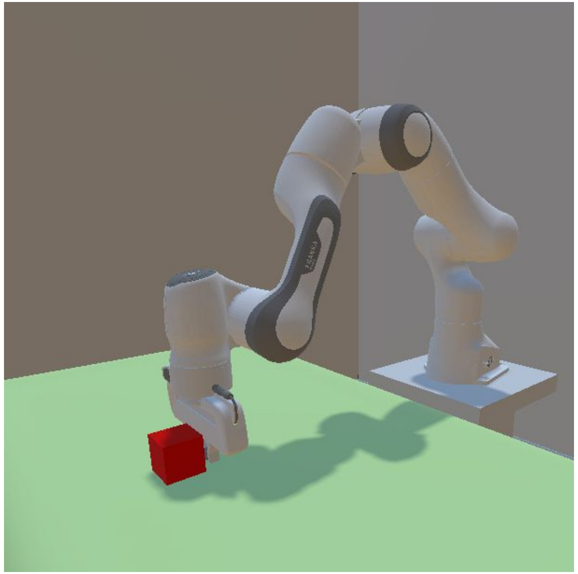
$$\frac{d}{dt}(x, \dot{x}) = (\dot{x}, \frac{F}{m})$$

An Embodied AI Example

Task: push block to target location.





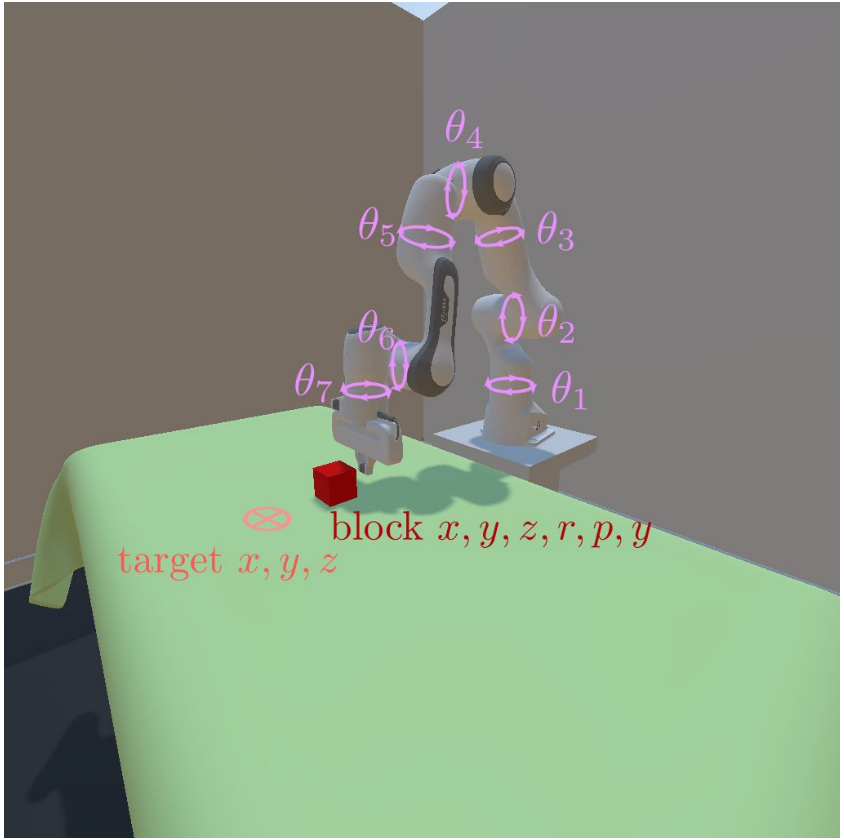


States

A **state** is a configuration of the world.

- In this example
  - Joint angles θ1-θ7
  - block position and orientation
  - target position

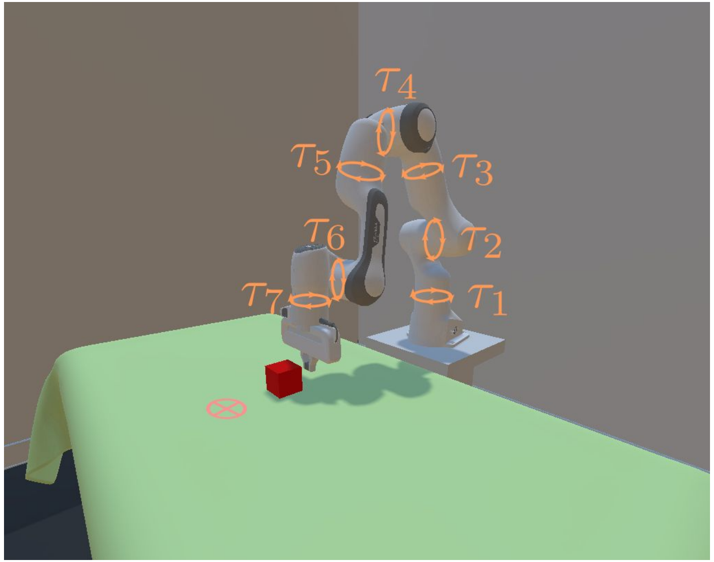
The collection of all states is called the state space $\mathcal{S}$.



Actions

An **action** is a robot command.

- For example
  - Motor torque



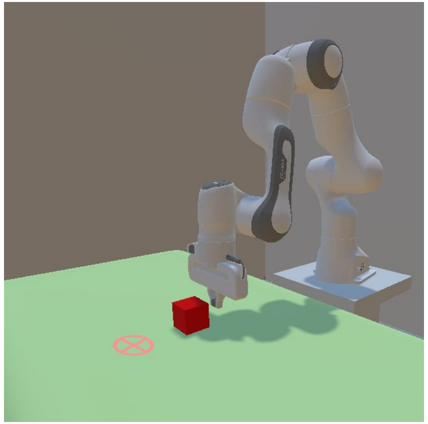
The collection of all actions is called the action space $\mathcal{A}$.

Transition

The **transition function** $\mathcal{T}$ describes how the **state** changes over time according to an **action.**

Formally, $\mathcal{T}$ describes the rate of change of the state given the current state and action.

$$\dot{s} := \frac{ds}{dt} = \mathcal{T}(s, a)$$



Transition function: classical mechanics

The Forward Model

The forward model is a 3-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T})$

$\mathcal{S}$: State Space	all possible world states
$\mathcal{A}$: Action Space	all possible control signals
$\mathcal{T}$: Transition	environment dynamics

Modeling Transition on a Computer

On a computer, things are discrete.

$$\dot{s} = \mathcal{T}(s, a) \xrightarrow{\text{Discretize over time}} s_{t+1} = \hat{\mathcal{T}}(s_t, a_t)$$

We call 1/Δt as the action frequency

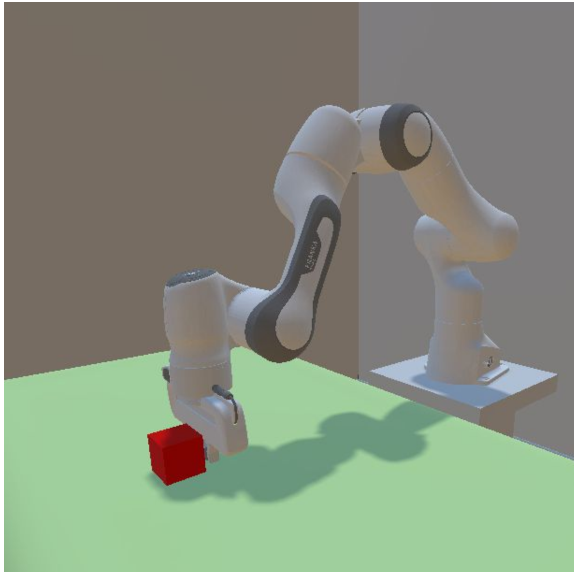
In general, the transition can be stochastic.

$$s_{t+1} \sim \mathcal{T}(\cdot | s_t, a_t)$$

Note: one may model stochasticity in the continuous time case (stochastic differential equations) but it is out of scope in this tutorial.

When is a Task Successful?

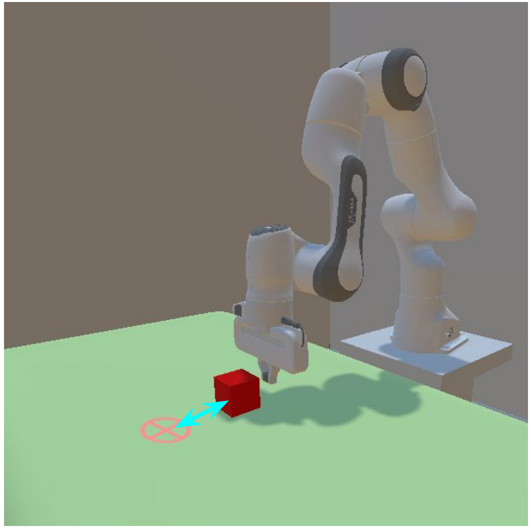
- How do we know if a task is complete?
- Idea: define success on states
  - Box xyz is close to target xyz
  - Box velocity is close to 0
  - Robot velocity is close to 0



When is a Task Successful?

- More generally, we can introduce a **reward** function $\mathcal{R}$ to measure how successful the current state/action is.

- For example
  - The environment gives a reward of 1 when the block is close to the target, 0 otherwise.



When is a Task Successful?

- More generally, we can introduce a **reward** function $\mathcal{R}$ to measure how successful the current state/action is.

- The 4 tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$ is formally known as a **Markov Decision Process (MDP)**.

Markov Decision Process

Markov Decision Process is a 4-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$

$\mathcal{S}$	: State Space	all possible world states
$\mathcal{A}$	: Action Space	all possible control signals
$\mathcal{T}$	: Transition	environment dynamics
$\mathcal{R}$	: Reward	how successful is the state/action

How to Solve Embodied AI Tasks

To solve an embodied AI task, the agent needs to know what action to take given the current state.

This is called a **policy**.

A policy π takes a **state** and outputs an **action** (can be stochastic).

$$a \sim \pi(\cdot|s)$$

A good policy should eventually complete the task (reach a successful state or accumulate a great amount of reward).

How to Solve Embodied AI Tasks

- Imitate an expert.
  - Imitation learning
  - Both $\mathcal{T}$ and $\mathcal{R}$ are not needed
- Learn to accumulate reward in an MDP
  - Reinforcement learning
  - Model-free: $\mathcal{T}$ is not modeled
  - Model-based: $\mathcal{T}$ is learned in the process
- Design rules based on mechanics
  - Classic robotics
  - $\mathcal{T}$ is modeled in advance (including learned models)

Outline

- Modeling and approaches for Embodied AI
  - World model
  - Learning-based methods to solve tasks
    - Imitation learning, reinforcement learning
  - Classic robotics

- Simulation technology for Embodied AI
- Building an environment from scratch

Optimal Policy

For a given policy $a \sim \pi(\cdot|s)$

We run the policy on the environment for H steps and collect rewards

$$a_t \sim \pi(\cdot|s_t) \quad s_{t+1} \sim \mathcal{T}(\cdot|s_t, a_t) \quad r_{t+1} \sim \mathcal{R}(\cdot|s_t, a_t, s_{t+1})$$

An optimal policy is the one that maximizes the expected cumulative reward

$$\mathbb{E}[\sum_{t=1}^{H} r_t]$$

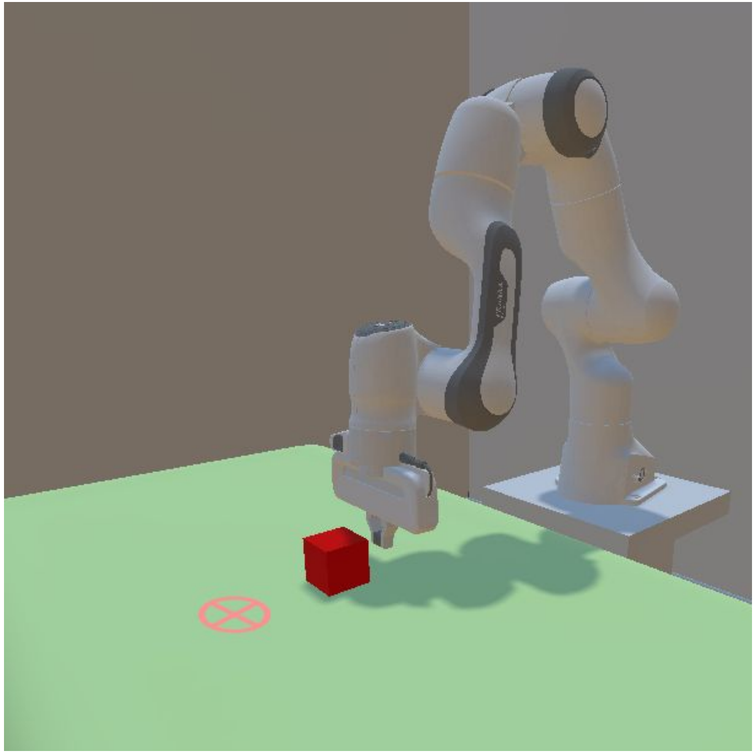
Note: in practice, a discount factor is often used to handle the case H=∞. It is not discussed here for simplicity.

Example of Optimal Policy

The environment gives a reward of 1 when the block is close to the target, 0 otherwise.

Let's also assume the system is terminated when the reward is 1.

An optimal policy is one that moves the block to the target eventually.



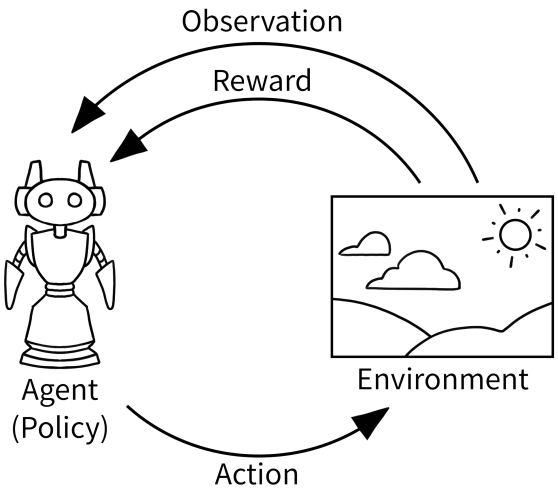
Partially-Observable MDP

In practice, the **state** is not always known.

Instead, we get some **observation.**

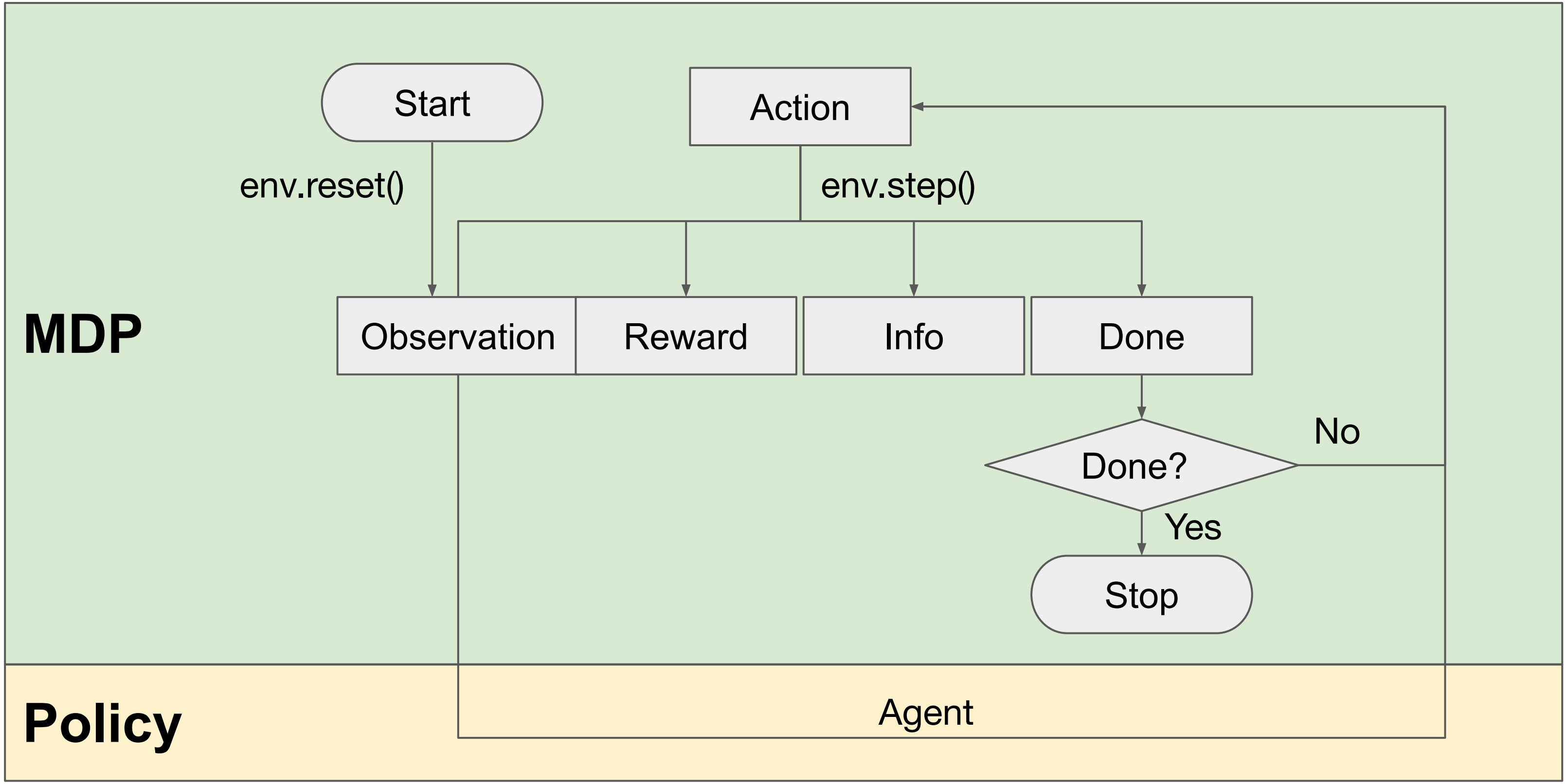
E.g., position of the cube vs an image of the cube

- Common observations
  - RGB-D image
  - Position & velocity of objects and robots
  - Task information (e.g. goal)
  - Other sensory readings



OpenAI Gym https://www.gymlibrary.ml/content/api/

Simulating MDP on a Computer



https://www.gymlibrary.ml/

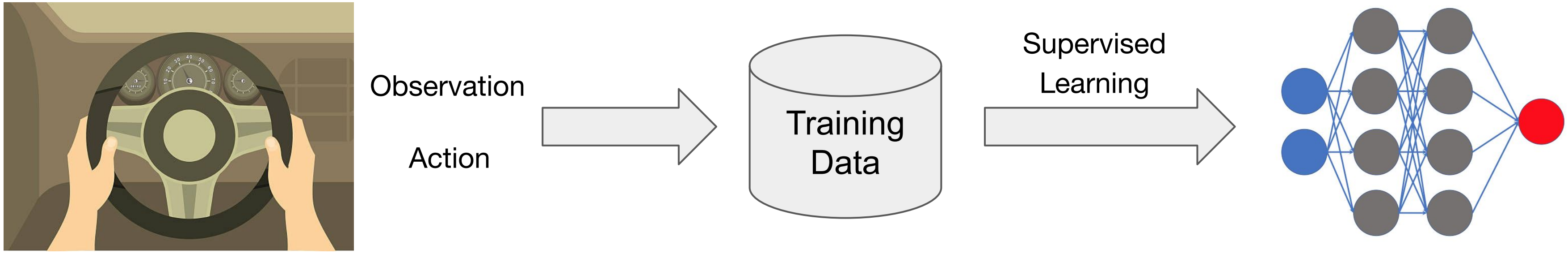
How to get a Good Policy

Now how do we find a good policy?

- Idea 1: assume an expert (e.g., human) has solved the task; mimic this behavior — **imitation learning.**
- Idea 2: interact with the environment and try to improve the policy with reward —**reinforcement learning.**

Imitation Learning

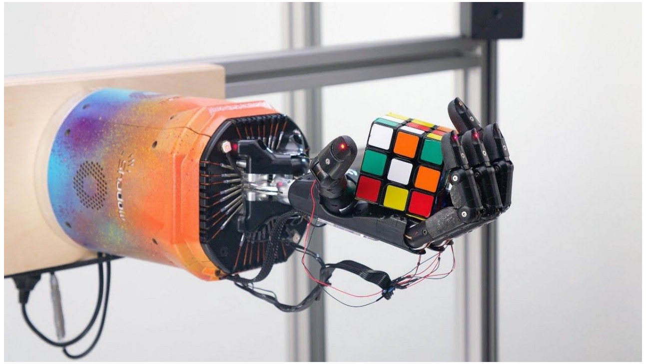
- Input: expert demonstrations $\{(s_t, a_t)\}$
- Output: policy $a \sim \pi_\theta(\cdot|s)$



Expert

Reinforcement Learning

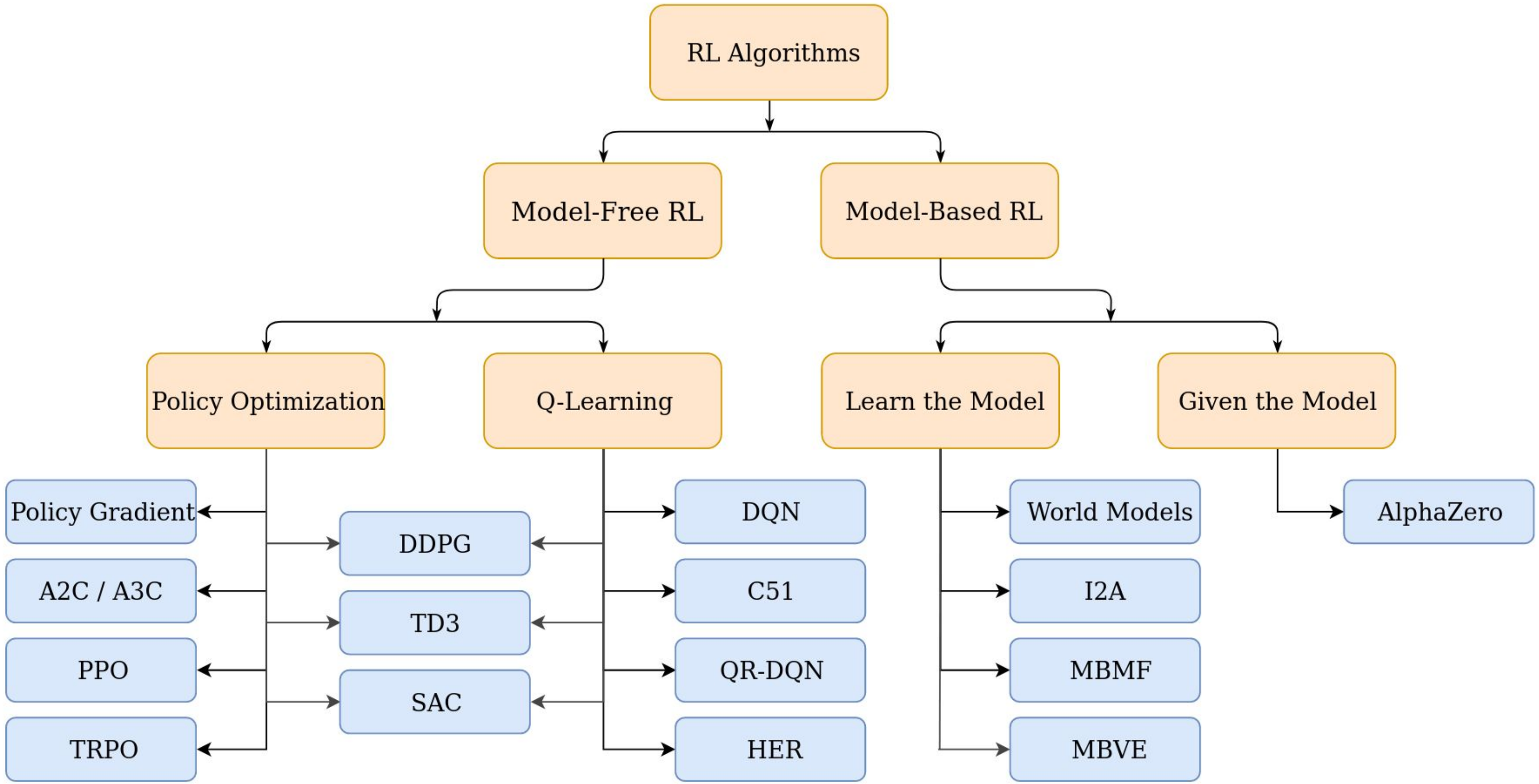
- What if we do not have expert data?
- Learn from interaction experience.
    a. Interact with environment (env.step) to collect experience.
    b. Use collected experience to improve the current policy.
    c. Repeat ab.



https://openai.com/blog/solving-rubiks-cube/

Recommended reading:
https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver

Reinforcement Learning Taxonomy



OpenAI Spinup https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

Combining Reinforcement Learning and Expert Demonstrations

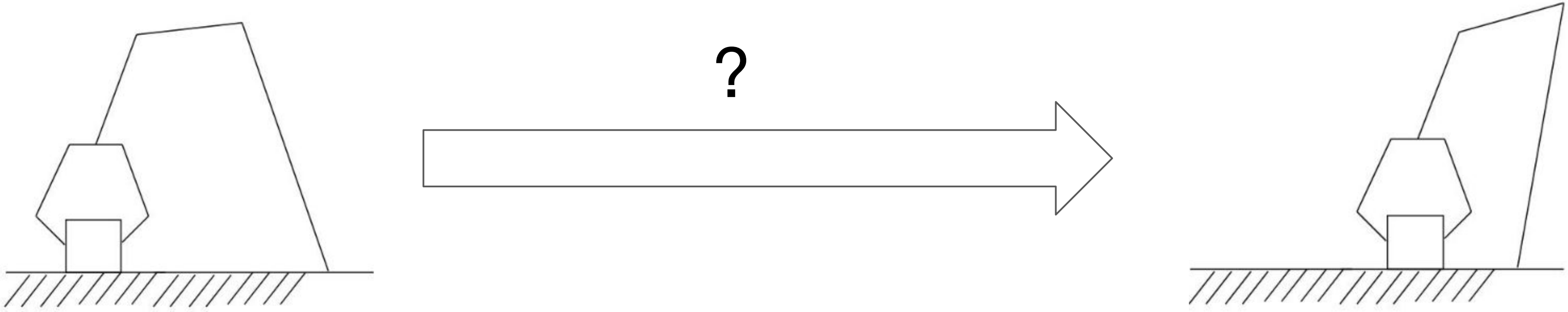
- “Learning from demonstrations”
  - Offline RL: train RL with given experience without further interactions
  - Augmenting online RL training with demonstrations
  - Dynamic movement primitives
  - Learning transition model from demonstrations

Outline

- Modeling and approaches for Embodied AI
  - World model
  - Learning-based methods to solve tasks
  - Classic robotics
- Simulation technology for Embodied AI
- Building an environment from scratch

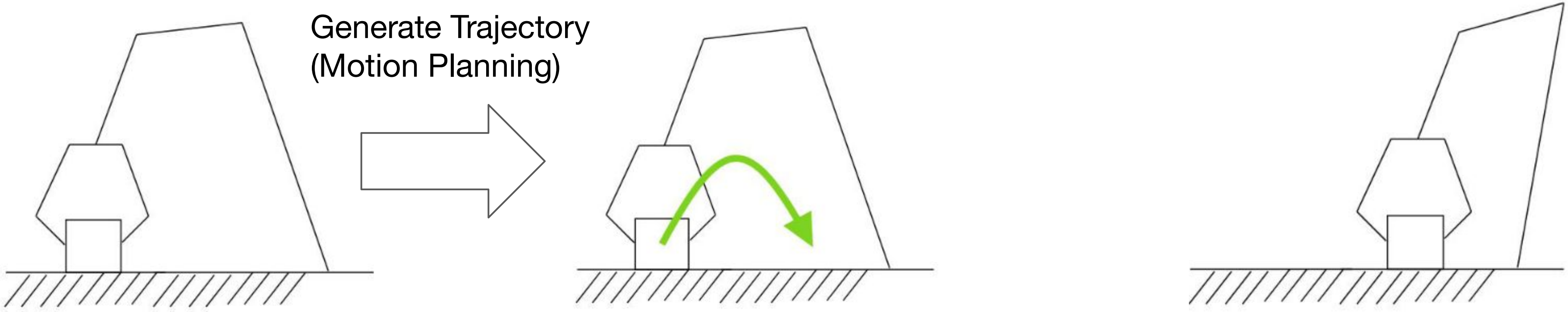
Plan and Control

A popular pipeline in classic robotics is planning and control.



Plan and Control

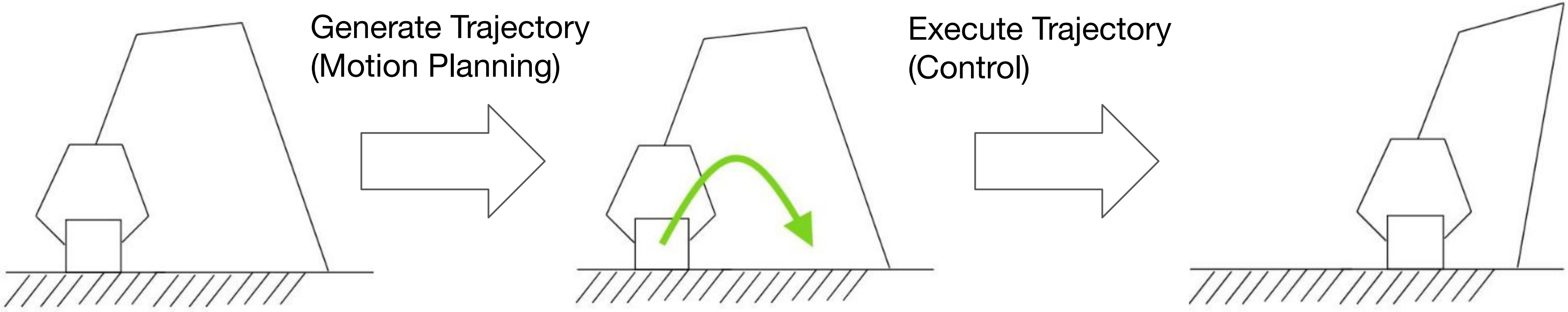
A popular pipeline in classic robotics is planning and control.



Motion planning generates a trajectory (position, velocity, and acceleration) of the robot.

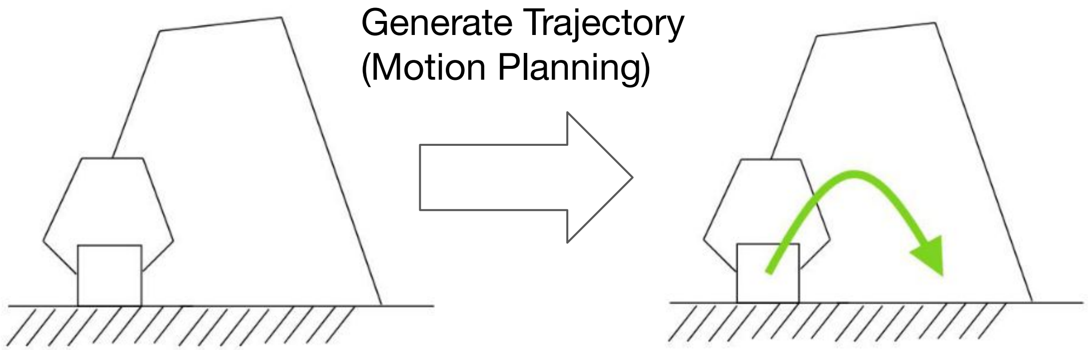
Plan and Control

A popular pipeline in classic robotics is planning and control.



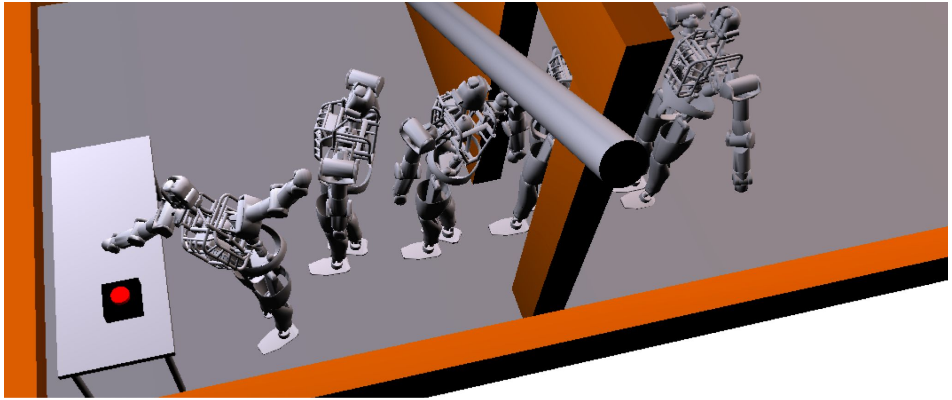
Motion planning generates a trajectory (position, velocity, and acceleration) of the robot.
Control executes the trajectory.

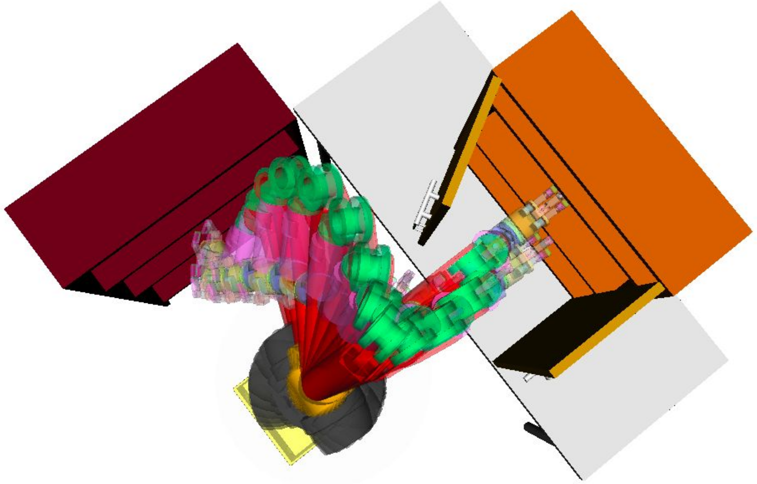
Motion Planning



Motion Planning

- Task: move a robot from one pose to another





Ratliff N, Zucker M, Bagnell J A, et al. CHOMP: Gradient optimization techniques for efficient motion planning, ICRA 2009
Schulman, John, et al. Finding Locally Optimal, Collision-Free Trajectories with Sequential Convex Optimization, RSS 2013

Motion Planning

- Task: move a robot from one pose to another
- Assumptions
  - We know the start and goal pose
  - We can verify if a given pose is valid (usually means collision-free)
  - We can verify whether a pose is reachable from another pose using some simple control strategy

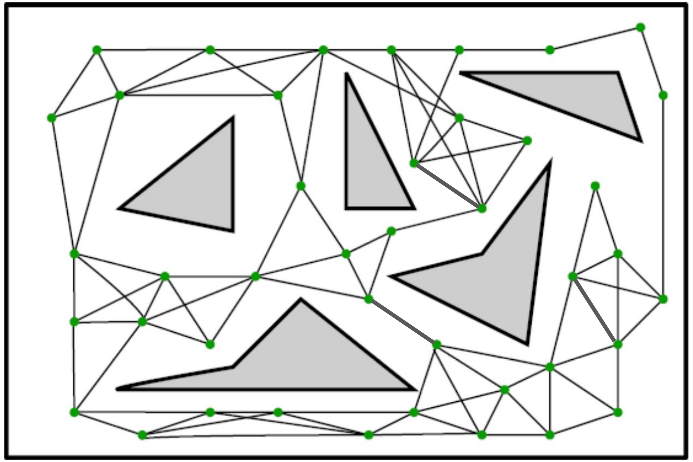
Motion Planning

- Task: move a robot from one pose to another
- Assumptions
  - We know the start and goal pose
  - We can verify if a random pose is valid (usually means collision-free)
  - We can verify whether a pose is reachable from another pose using some simple control strategy
- Algorithms
  - Rapidly-exploring random tree (RRT)
  - Probabilistic roadmap method (PRM)

Motion Planning Example: PRM

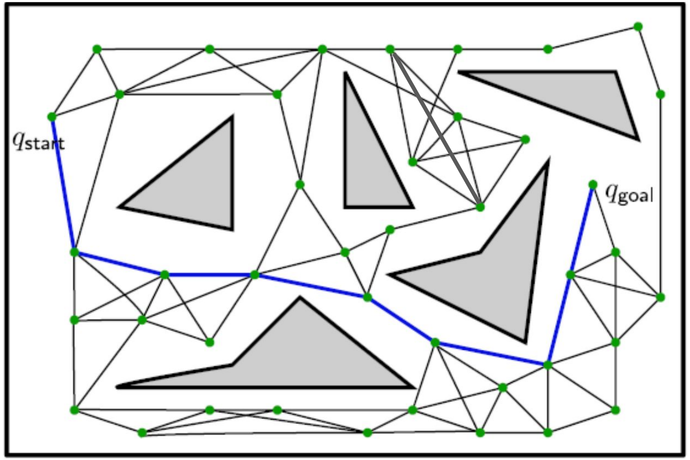
Motion Planning Example: PRM

- Phase 1: Map construction
  - Randomly sample collision-free configurations
  - Connect every sampled state to its neighbors
  - Connect the start and goal states to the graph



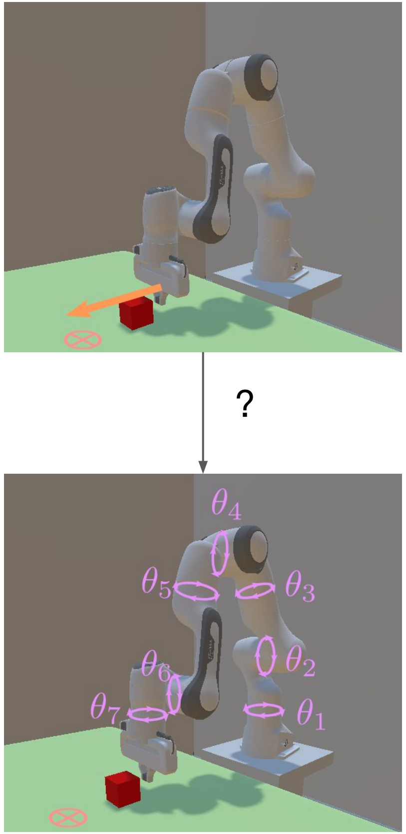
Motion Planning Example: PRM

- Phase 2: Query
    - Run path finding algorithms like Dijkstra



How to Find a Robot Pose For Grasping?

- Some tasks (such as grasping) require moving the gripper to a position.
- How do we find the robot pose of a given gripper pose?

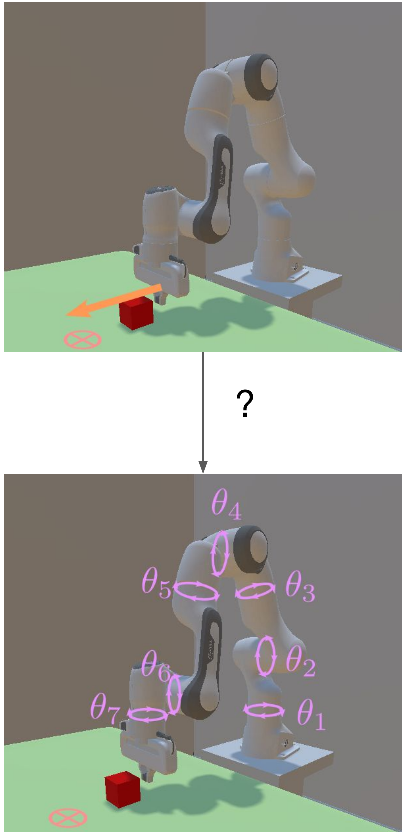


How to Find a Robot Pose For Grasping?

- Some tasks (such as grasping) require moving the gripper to a position.
- How do we find the robot pose of a given gripper pose?
    - **Inverse Kinematics (IK)**

```
robot_model = robot.create_pinocchio_model()

joint_positions, success, error = robot_model.compute_inverse_kinematics(
    link_idx,
    target_pose,
    active_qmask = joint_mask  # joints with mask value 1 are allowed to move
    max_iterations = 100
)
```

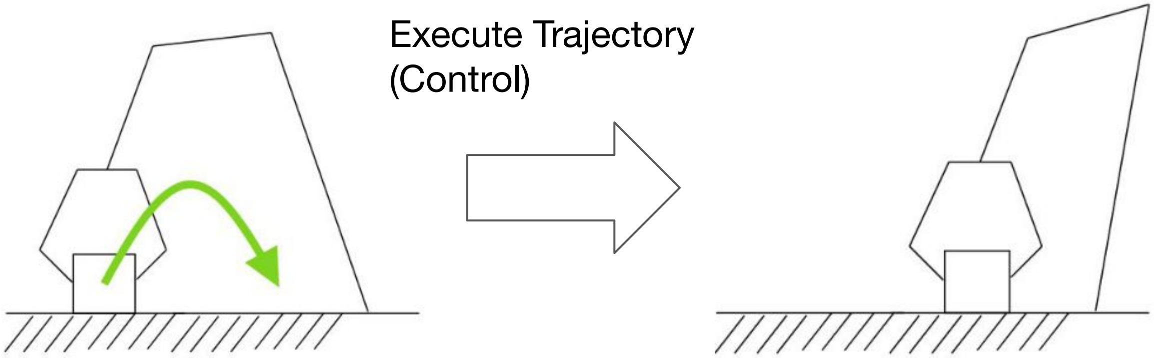
Code in SAPIEN



Time Parameterization

- PRM/RRT gives a path with discrete joint positions q_d
- A time parameterization algorithm converts the path q_d to a joint **trajectory** $(q_d, \dot{q}_d, \ddot{q}_d)$ with time.

Control



Control

- Robotic control executes a given trajectory $(q_d, \dot{q}_d, \ddot{q}_d)$ by controlling the joint torques τ

Control

- Robotic control executes a given trajectory $(q_d, \dot{q}_d, \ddot{q}_d)$ by controlling the joint torques τ
    - q represents the joint positions of a robot
- Similar to $F = ma$, the dynamic model of a robot is known.
    - Forward dynamics: $\ddot{q} = \mathrm{FD}(\tau; q, \dot{q})$
    - Inverse dynamics: $\tau = \mathrm{ID}(\ddot{q}; q, \dot{q}) = M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q)$

Inertia matrix   Coriolis matrix   Gravity & other forces

Control

- What we have
  - Trajectory $(q_d, \dot{q}_d, \ddot{q}_d)$
  - Inverse dynamics: $\tau = \mathrm{ID}(\ddot{q}; q, \dot{q})$
- Ideally, using τ computed from $\ddot{q}_d$ gives a perfect trajectory.
- However, the real world is not perfect. What if there is some error?

$$e = q - q_d$$

PD Control

- The PD control law has the form

$$\tau = -K_v \dot{e} - K_p e \quad \text{where} \quad K_v, K_p \in \mathbb{S}^+ \quad e = q - q_d$$

- Intuitively
  - When the position lags behind ($e < 0$), increase τ to catch up
  - When it is moving too slow ($\dot{e} < 0$), also increase τ to catch up
  - Inverse dynamics is not used at all!

PD Control

- PD control has no convergence guarantee in general
- When it converges, often $e \neq 0$
- How to fix it?

- Combine PD control and inverse dynamics. (Augmented PD control)

$$\tau = \mathrm{ID}(\ddot{q}; q, \dot{q}) - K_v \dot{e} - K_p e$$

PID Control

- To mitigate steady-state errors, an integral term is often added.

PID: $$\tau = -K_v \dot{e} - K_p e - K_i \int_0^t e(t) dt$$

Augmented PID: $$\tau = \mathrm{ID}(\ddot{q}; q, \dot{q}) - K_v \dot{e} - K_p e - K_i \int_0^t e(t) dt$$

$$K_v, K_p, K_i \in \mathbb{S}^+ \qquad e = q - q_d$$

Example: PD Velocity Controller

- Velocity controller
  - Constant velocity trajectory; acceleration is 0
  - Do not care about position error; $K_p = 0$

$$\tau = \mathrm{ID}(0; q, \dot{q}) - K_v \dot{e}$$

```
for joint in robot.get_active_joints():
    # stiffness: diagonal of Kp
    # damping: diagonal of Kv
    joint.set_drive_property(stiffness=0, damping=10.0)

robot.set_drive_velocity_target(joint_velocity_target)  # set PD control velocity
passive_force = robot.compute_passive_force(gravity=True, coriolis_and_centrifugal=True)  # ID(0;q,q̇)
robot.set_qf(passive_force)  # augment PD control with ID
```

Code in SAPIEN

Use Control in MDP Modeling

- When an RL work says: *we use "velocity control" or "position control" as action*. What does that mean?

Use Control in MDP Modeling

- The action in an MDP can be “target joint velocity” or “target joint position” for a controller.

Use Control in MDP Modeling

- The action in an MDP can be “target joint velocity” or “target joint position” for a controller.
- A controller (such as PD) is used to convert this velocity or position signal to joint torques, which are then used to drive the robot.

Use Control in MDP Modeling

- The action in an MDP can be “target joint velocity” or “target joint position” for a controller.
- A controller (such as PD) is used to convert this velocity or position signal to joint torques, which are then used to drive the robot.
- Joint velocity/position may be a better choice for MDP action (than torque) due to learnability and sim-to-real transferability.

More About Control

- Control focuses on stability and robustness
- There is a huge literature
  - Optimal control
  - Feedforward/feedback control (including PD)
  - Robust control
  - Self-organized control
  - Stochastic control
  - …
- Optimal control has a strong connection with RL

Summary

- Embodied AI Approaches
  - Learning-based methods
    - Imitation learning
    - Reinforcement learning
    - …
  - Classic robotics
    - Planning
    - Control
    - …
- In-depth discussion of these topics
  - Course: machine learning for robotics
  - https://haosulab.github.io/ml-for-robotics/SP22/index.html

How do we Study
Embodied AI Algorithms?

- An environment is required to develop approaches
- Real robot?
  - High costs
  - Safety concerns
- Simulation environment?
  - Physical simulation
  - Camera simulation
  - Assets loading
  - Sim-to-real gaps

Outline

- Modeling and approaches for Embodied AI
- Simulation technology for Embodied AI
  - From simulator to environment
  - Rigid body simulation
  - Camera simulation
  - Assets
- Building an environment from scratch

Outline

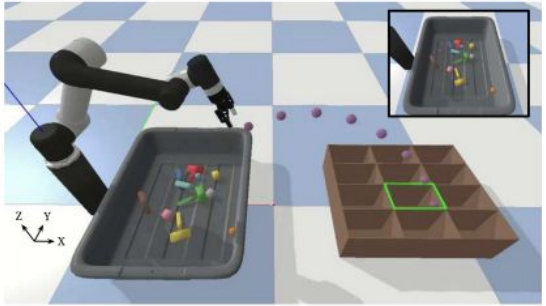
- Modeling and approaches for Embodied AI
- Simulation technology for Embodied AI
  - From simulator to environment
  - Rigid body simulation
  - Camera simulation
  - Assets
- Building an environment from scratch

Simulator

- A library (often a simple SDK) that simulates one or more physical processes.
  - Rigid body
  - Particle system
  - Light transport (renderer)

MuJoCo Engine

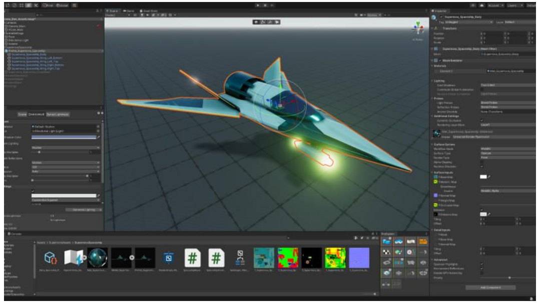
Nvidia Flex Solver



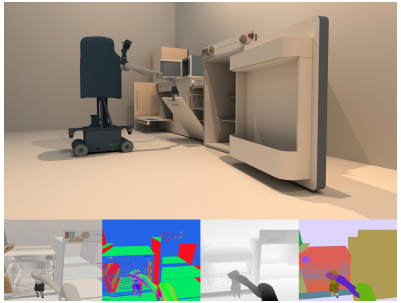
Bullet Physics SDK

Engine

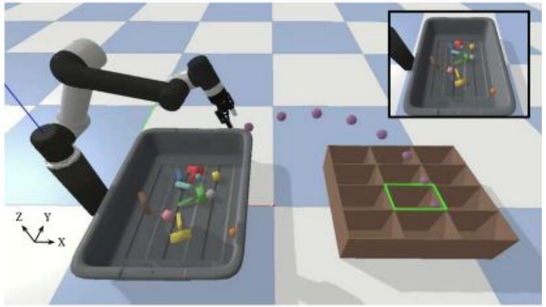
- A software that bundles together simulators to help developers.
  - E.g., “Game engine”
  - Sometimes also called a simulator



Unity



SAPIEN

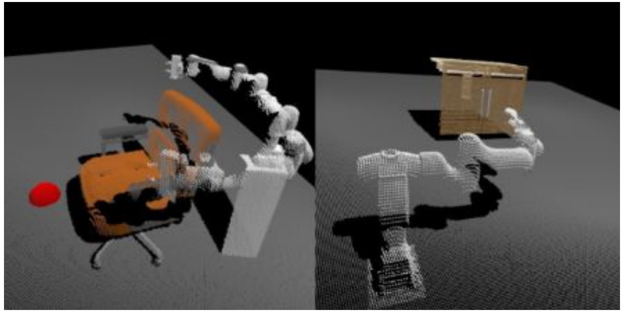


PyBullet

Environment

- Bundles of engines/simulators, assets, and tasks for studying specific embodied AI problems.
  - Some environments also call themselves a simulator.

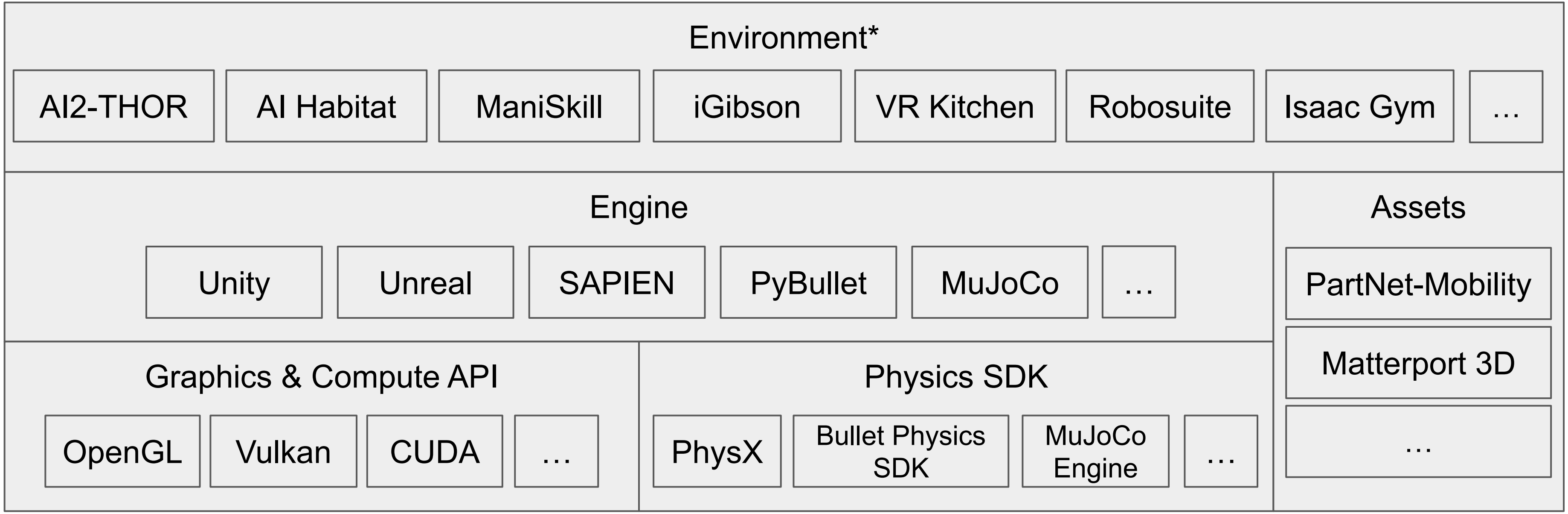
Ai2-THOR



ManiSkill

AI Habitat

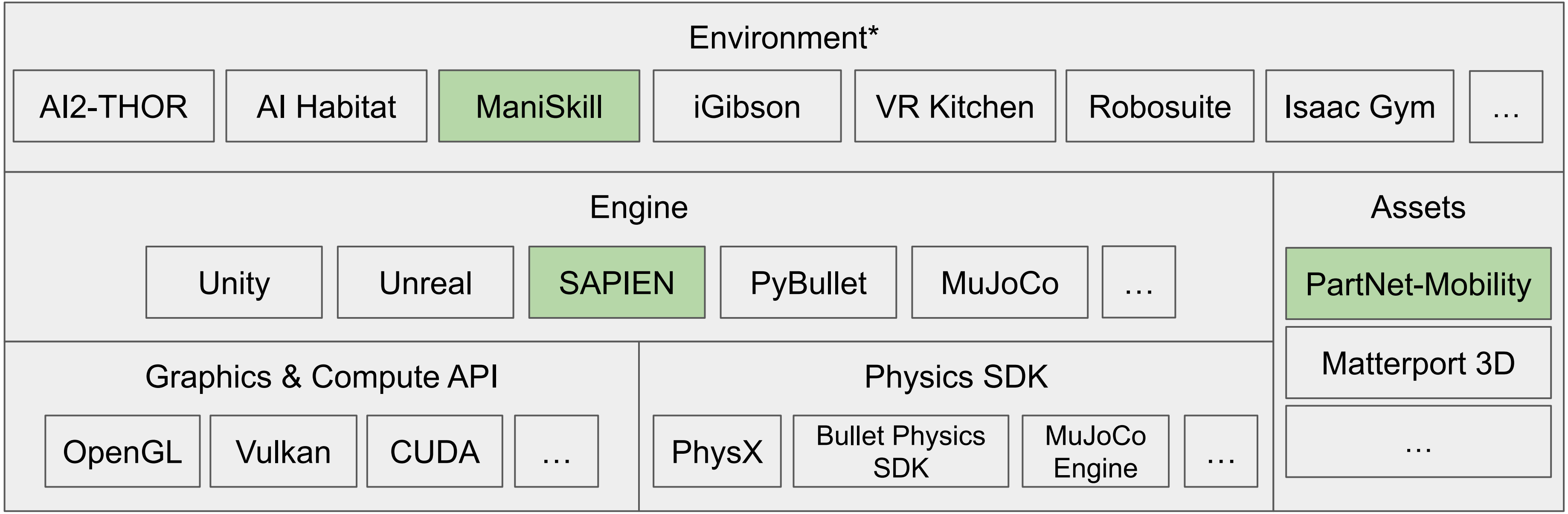
Dependency



*Some environments may not have an engine and are developed directly on low-level graphics & physics SDKs.

Note: simulator, engine, framework, environment, etc. do not have formal definitions and are often used interchangeably, so always use context to understand the software.

Dependency



*Some environments may not have an engine and are developed directly on low-level graphics & physics SDKs.

Note: simulator, engine, framework, environment, etc. do not have formal definitions and are often used interchangeably, so always use context to understand the software.

What to Choose?

- Graphics/physics SDK
  - If you are creating a new engine or modifying an engine.
- Engine
  - If you are creating a new environment.
- Environment
  - If you want to solve a predefined embodied AI problem.

Outline

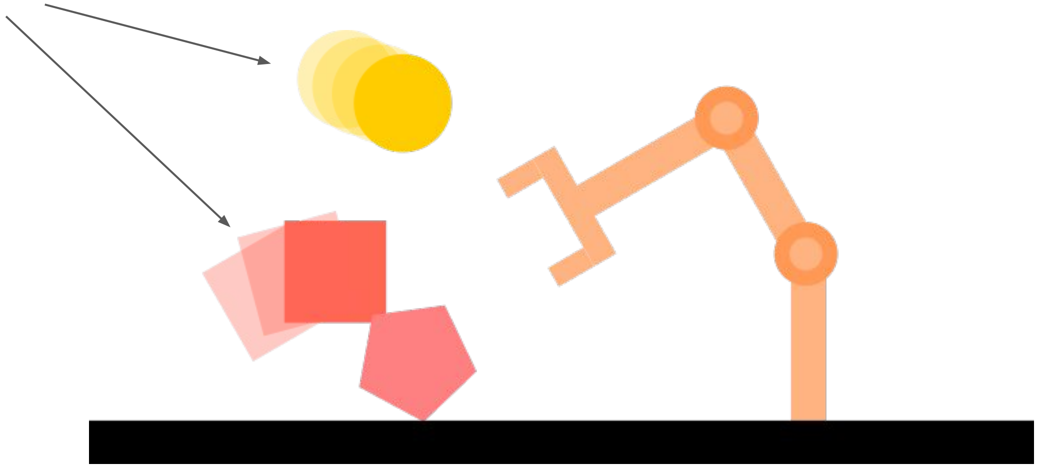
- Modeling and approaches for Embodied AI
- Simulation technology for Embodied AI
  - From simulator to environment
  - Rigid body simulation
  - Camera simulation
  - Assets
- Building an environment from scratch

Rigid Body Simulation

- You probably have heard of physical simulators
  - MuJoCo, Bullet, PhysX
  - What do they do?

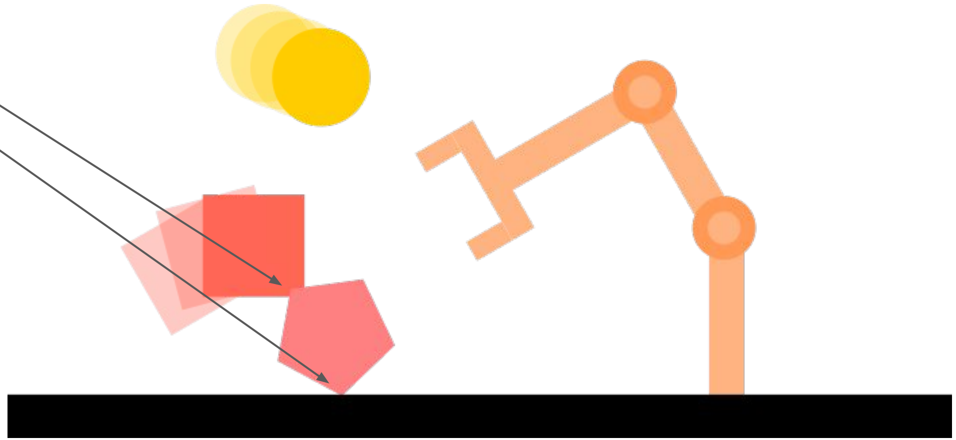
Rigid Body Simulation

- You probably have heard of physical simulators
  - MuJoCo, Bullet, PhysX
  - What do they do?
    - Model motion of bodies



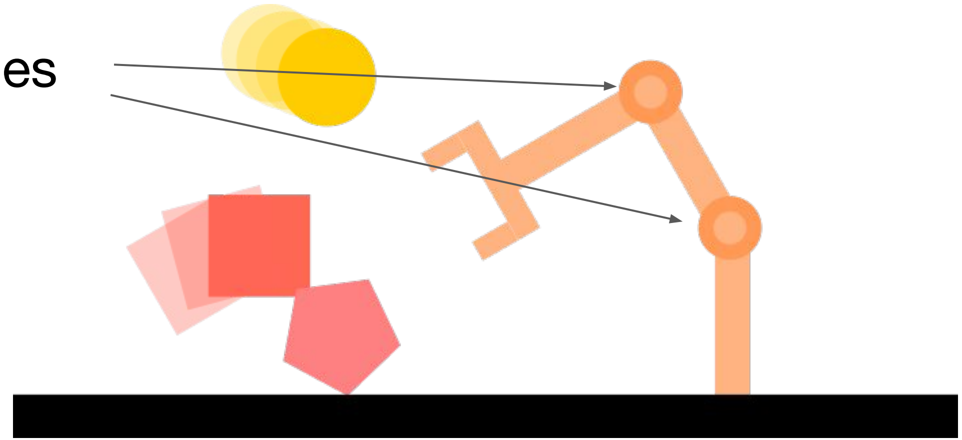
Rigid Body Simulation

- You probably have heard of physical simulators
  - MuJoCo, Bullet, PhysX
  - What do they do?
    - Model motion of bodies
    - Handle collisions



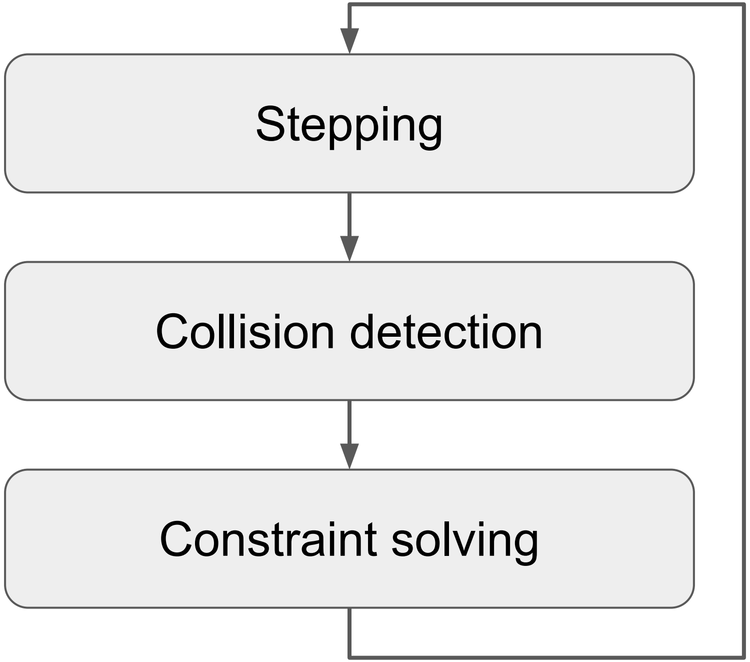
Rigid Body Simulation

- You probably have heard of physical simulators
  - MuJoCo, Bullet, PhysX
  - What do they do?
    - Model motion of bodies
    - Handle collisions
    - Handle connected bodies



Rigid Body Simulation

- Most rigid body simulations repeat the following steps

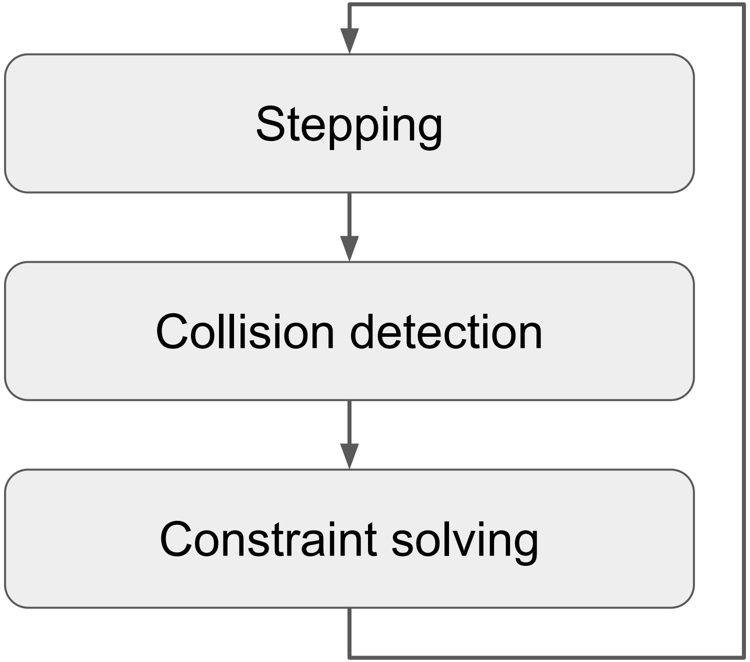


Rigid Body Simulation

- Most rigid body simulations repeat the following steps

pybullet.stepSimulation

sapien.core.Scene.step



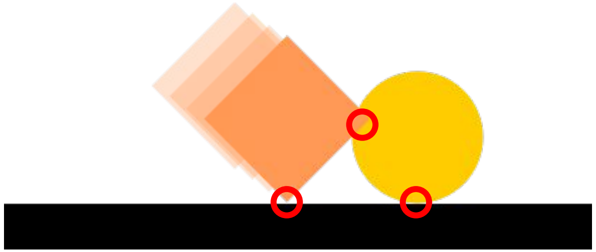
Stepping

- Advance the simulation time
- Most common choice: semi-implicit Euler

$$v_{t+1} = v_t + a_t \Delta t$$
$$x_{t+1} = x_t + v_{t+1} \Delta t$$

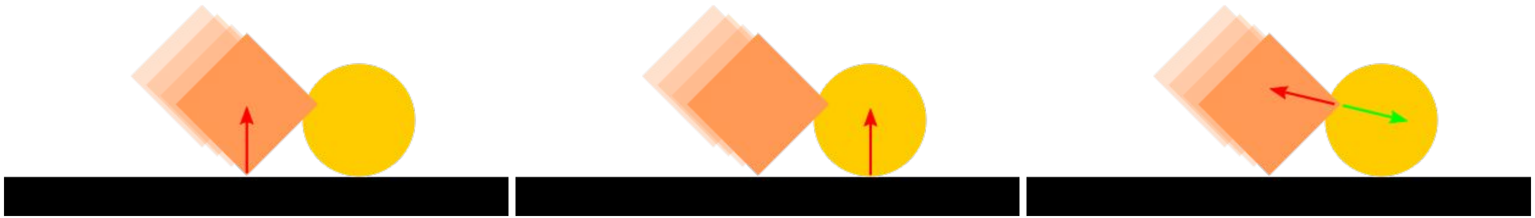
Collision Detection

- Collision detection tries to find contacts
  a. Run a collision detection algorithm to find contact point positions, normals, and penetration/separation distances.
  b. Add each contact to the constraints
  c. (Constraints are solved later)



Constraint Solving

- Constraints are physical restrictions, such as
  - Bodies connected with joints
  - No penetration between contact bodies
- The solver solves a system of equations or an optimization problem to figure out a proper force/impulse to add for each constraint.
  - Key parameter **solver iterations**: how long the solver is allowed to run



Outline

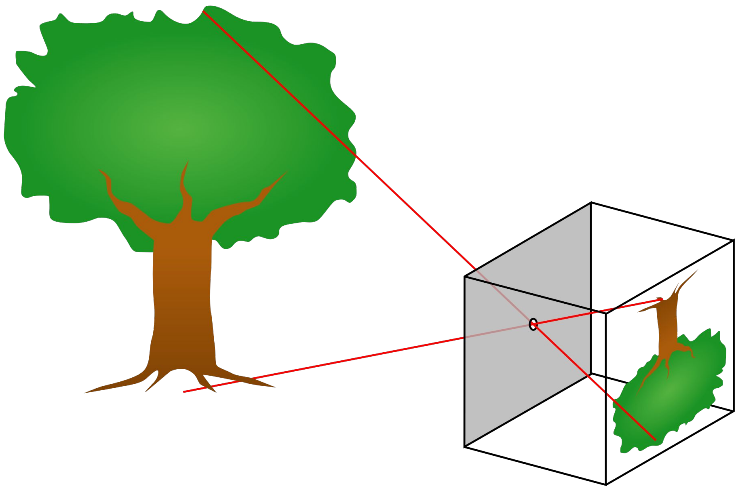
- Modeling and approaches for Embodied AI
- Simulation technology for Embodied AI
  - From simulator to environment
  - Rigid body simulation
  - Camera simulation
  - Assets
- Building an environment from scratch

Camera Simulation

- Camera simulation is achieved by **rendering**
  - Modeling light transport
- Components in camera simulation
  - Cameras
  - Lights
  - Geometries
  - Materials and textures

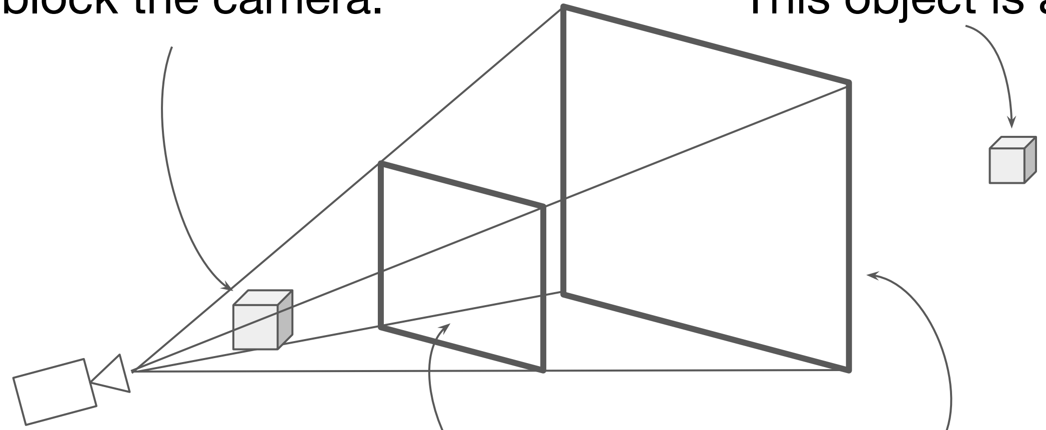
Camera Model

- Real cameras use lenses, which can cause defocus blur
- Simulations typically use the simplified pinhole model



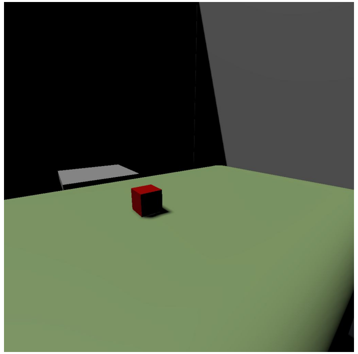
Camera Model

- Camera frustum
  - Many simulated cameras (rasterizers like OpenGL) have a range limit, forming a frustum

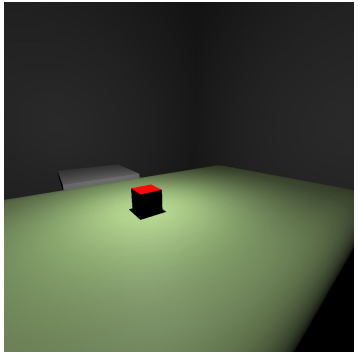


Lights

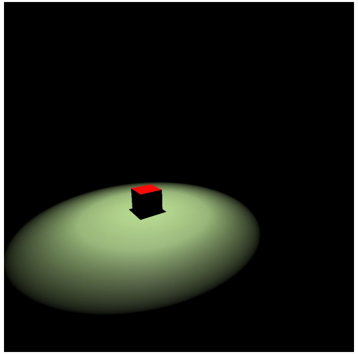
- Common types of lighting
  - Directional light (e.g. sun)
  - Point light (e.g. lamp)
  - Spot light (e.g. flashlight)
  - Ambient light (e.g. environment map)
  - Area light (e.g. bright screen)
  - Indirect light (from inter-reflections)



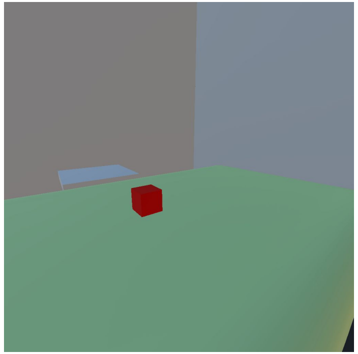
Directional light



Point light



Spot light

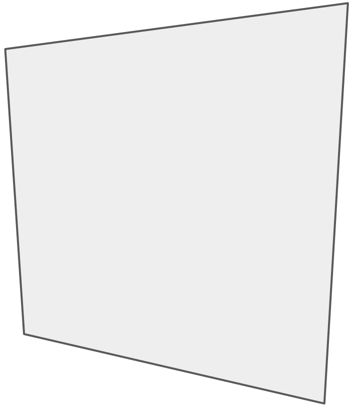


Ambient light

Geometry, Material, and Texture

- Geometry
  - Mesh, curve, volume, etc. representing shape of objects
- Material
  - Material describes the reflection and refraction properties of objects
    - Physically based rendering (PBR) model. E.g., microfacet models
    - Phong model
- Texture
  - Describes spatially varying material parameters over the geometry

Geometry, Material, and Texture



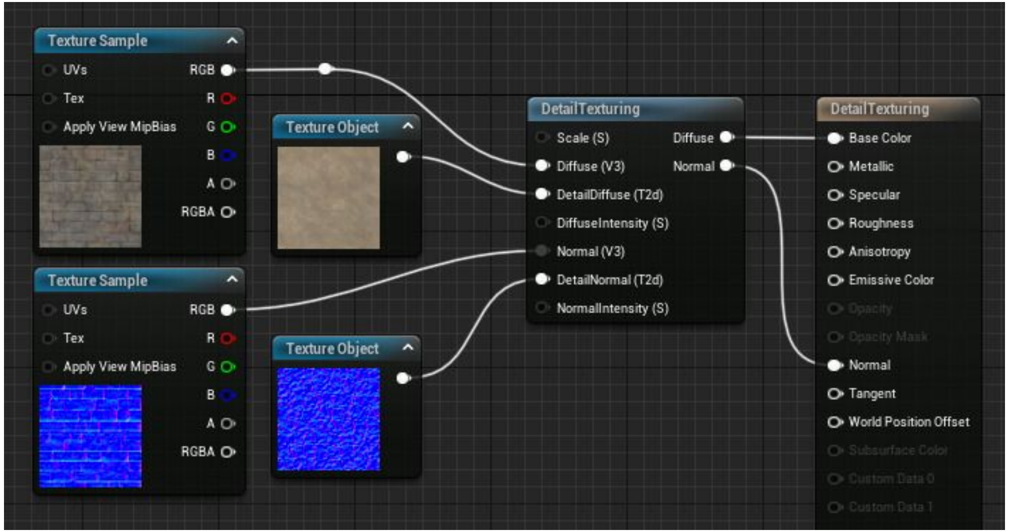
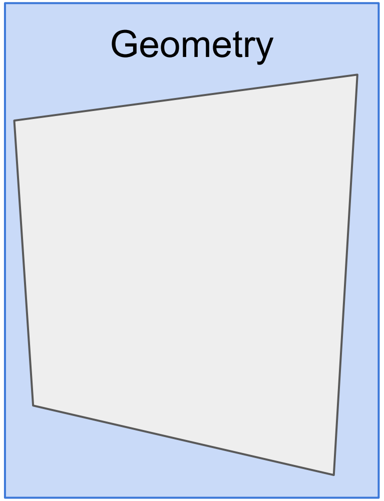


Image source: Unreal Engine 5 Documentation
https://docs.unrealengine.com/5.0/en-US/adding-detail-textures-to-unreal-engine-materials/

Geometry, Material, and Texture



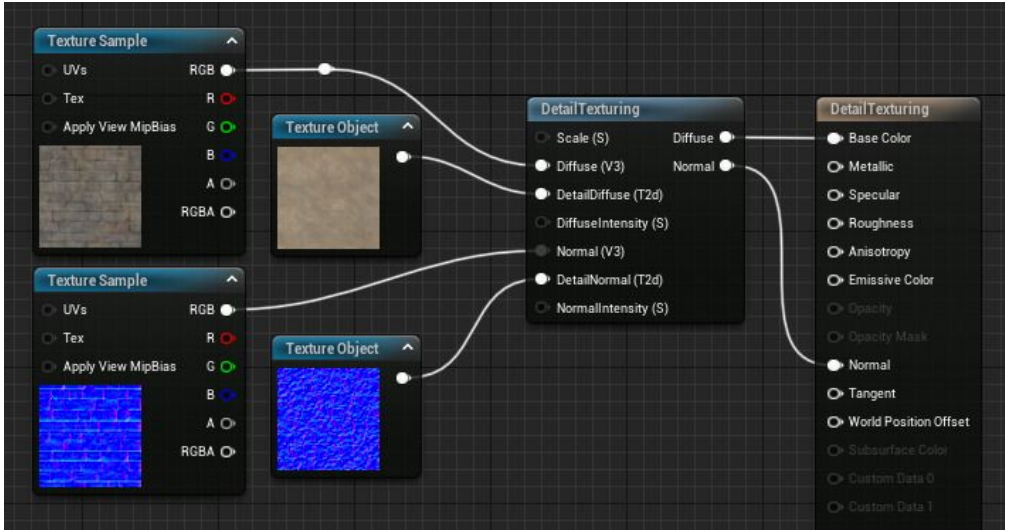
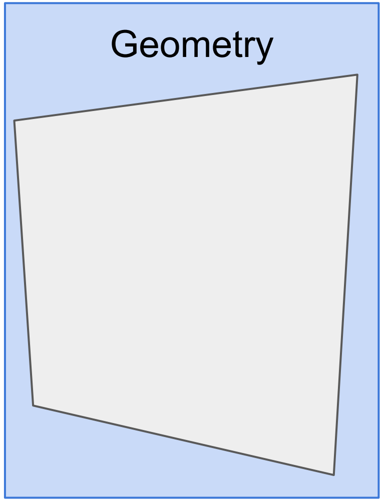


Image source: Unreal Engine 5 Documentation
https://docs.unrealengine.com/5.0/en-US/adding-detail-textures-to-unreal-engine-materials/

Geometry, Material, and Texture



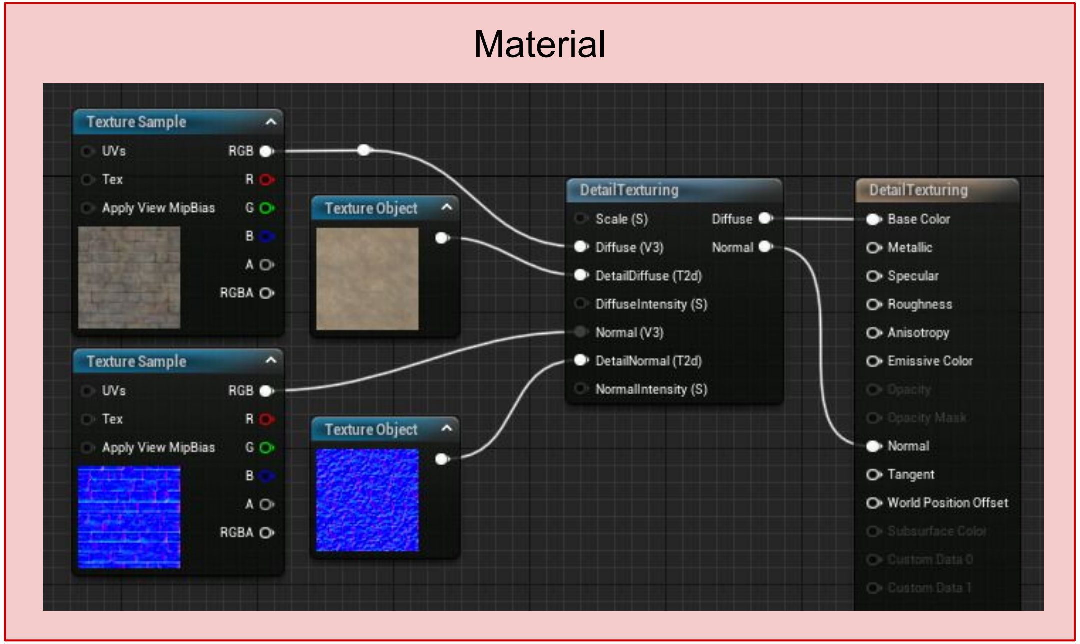


Image source: Unreal Engine 5 Documentation
https://docs.unrealengine.com/5.0/en-US/adding-detail-textures-to-unreal-engine-materials/

Geometry, Material, and Texture

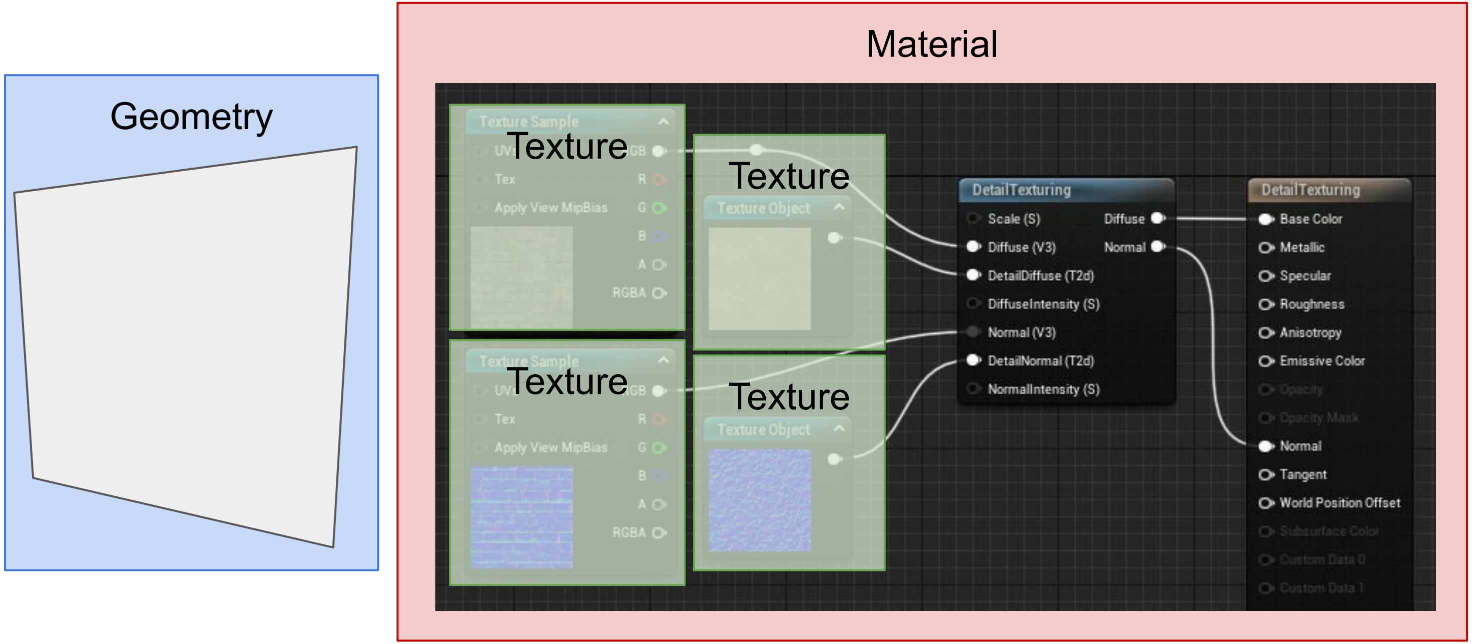


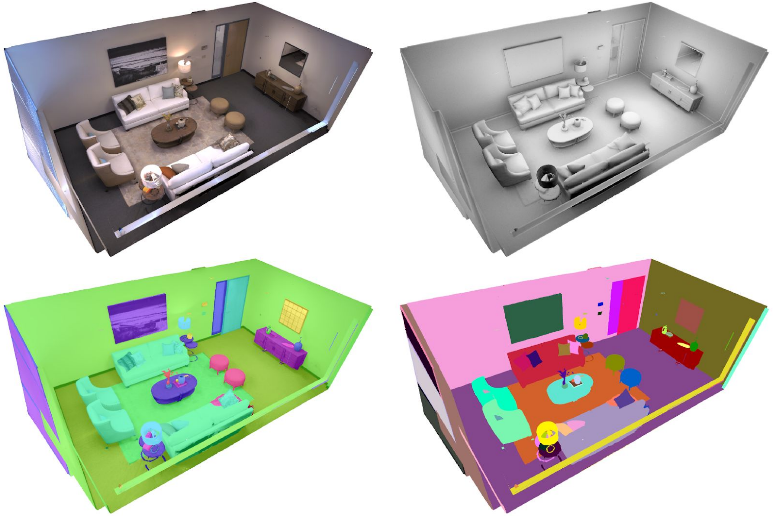
Image source: Unreal Engine 5 Documentation
https://docs.unrealengine.com/5.0/en-US/adding-detail-textures-to-unreal-engine-materials/

Outline

- Modeling and approaches for Embodied AI
- Simulation technology for Embodied AI
  - From simulator to environment
  - Rigid body simulation
  - Camera simulation
  - Assets
- Building an environment from scratch

Objects & Scenes

PartNet-Mobility Dataset



Replica Dataset

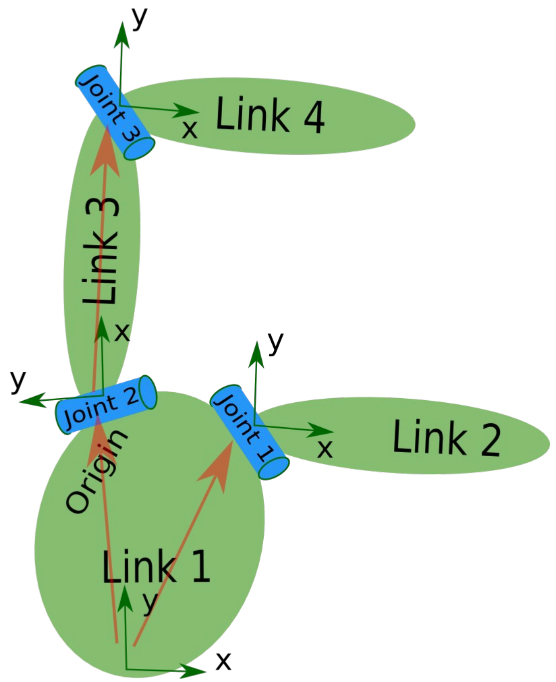
Common 3D Model Exchange Formats

Format	Material	Notes
obj+mtl	Phong, PBR (extension)	Plain text; can be edited manually.
stl	None	Suitable for collision shapes.
ply	Vertex color	Vertex color is rarely used for material.
dae	Phong	Many inconsistencies. Different software seems to disagree on its standard.
fbx	Phong	
glb/gltf	PBR, Phong	Most powerful

Recommended model loader: Assimp
https://github.com/assimp/assimp

URDF

- Unified Robot Description Format
- Designed for robotics, including kinematics and dynamics
- XML describing an articulated body
  - <link>: a rigid part
    - <inertial>: mass and inertia of the link
    - <collision>: collision geometry
    - <visual>: rendering geometry
  - <joint>: a connector for 2 links
    - Revolute, continuous, prismatic, fixed, floating, plannar



Importing Assets Into…

Importing Assets Into…

- Low-level simulators/renderers
  - Use assimp

Importing Assets Into…

- Low-level simulators/renderers
  - Use [assimp]
- Engines
  - SAPIEN and PyBullet load assets dynamically.
  - MuJoCo loads URDF or their own format (MJCF) all at once
  - Game engines have intuitive UI and drag-and-drop. Loading assets programmatically is often much harder but can be done.

Importing Assets Into…

- Low-level simulators/renderers
  - Use assimp
- Engines
  - SAPIEN and PyBullet load assets dynamically.
  - MuJoCo loads URDF or their own format (MJCF) all at once
  - Game engines have intuitive UI and drag-and-drop. Loading assets programmatically is often much harder but can be done.
- Environments
  - Customizability of environments is a design choice.
  - Environments also inherit asset loading procedure from their engines.

Summary

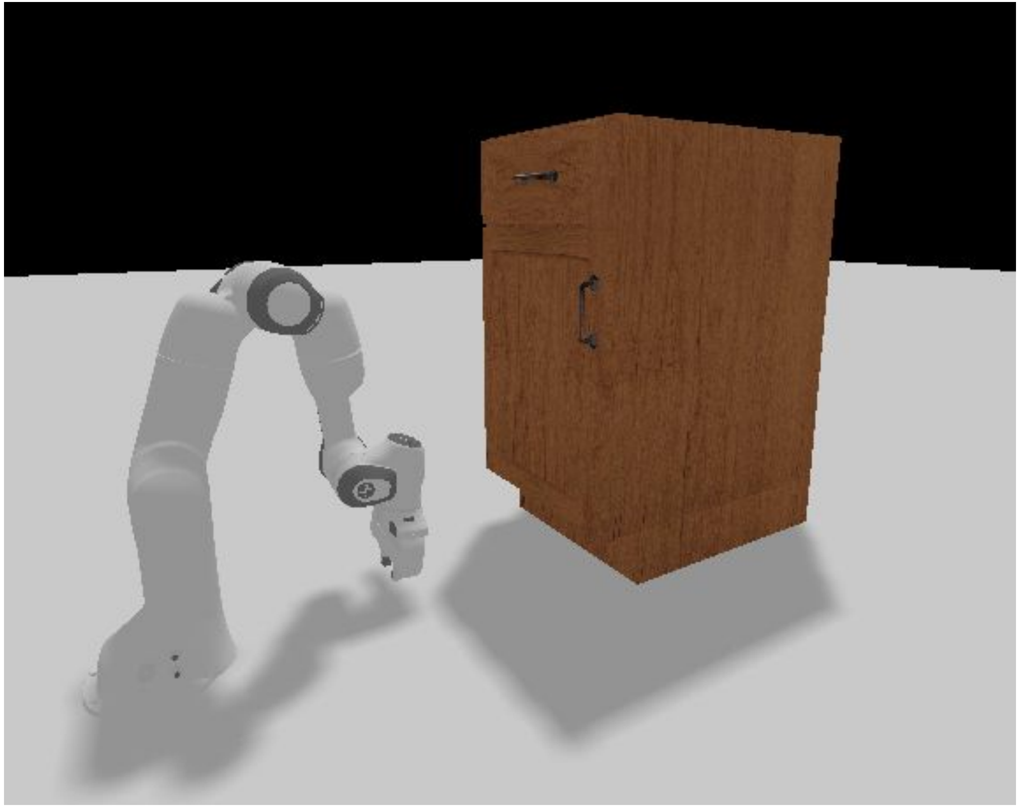
- Simulators, engines, environments
- Rigid body simulation
  - Stepping, collision detection, constraint solving, repeat
- Camera simulation
  - Camera, light, geometry, material, texture
- Assets
  - 3D model formats, URDF

Outline

- Modeling and approaches for Embodied AI
- Simulation technology for Embodied AI

- Building an environment from scratch

Build “Open Cabinet” From Scratch

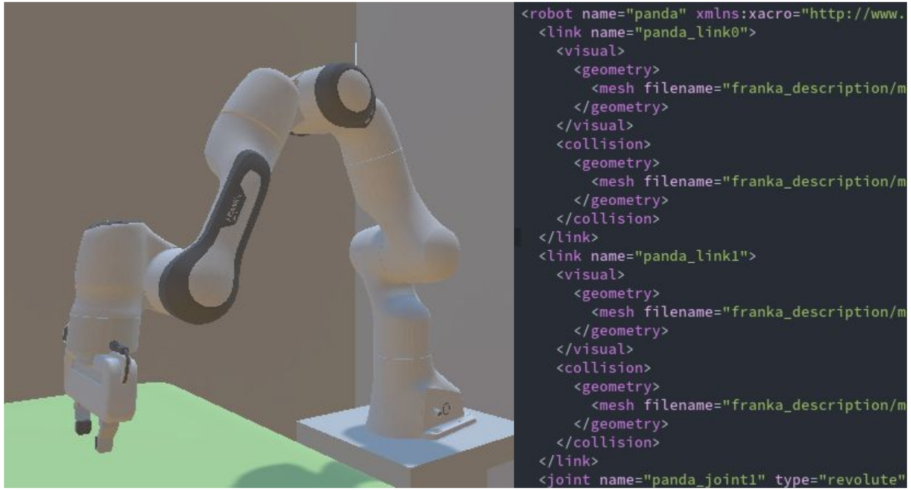


Decide the Task

- Task: open the door of a cabinet with a robot arm

Decide the Assets

- I will use this cabinet and this robot
    - They are both in the URDF format



Decide which Controller to Use

- Torque control?
- PD velocity control?
- A combination of controllers?
- Teleport (non-physical)?
- …

In this tutorial, I will show PD velocity control.

Decide Object Interactions

- How will the robot open the door?
  - Through a physical process: force and friction
  - Use a simplified model: the robot is “glued” to the door when it is close to the door.
  - Use an even simpler model: the door automatically opens if the robot gripper is within range.

- This tutorial demonstrates using the physical process.

Decide the Observation Space

- Simulation state?
- RGB-D from a camera?
- RGB-D + robot state?

This tutorial uses an RGB-D camera and the robot state as observation.

Decide the Framework

- Choose a framework
  - SAPIEN, PyBullet, MuJoCo, Unity, etc.

- This tutorial uses SAPIEN
  - We made it.
  - Very good debug viewer.
  - Clean API, type hint, and code completion, suitable for education.

Start Coding the MDP

- Write down the interface for an MDP.
- The render function is mainly for visualization

```
from gym import Env


class OpenCabinetEnv(Env):
    def __init__(self):
        pass

    def reset(self):
        raise NotImplementedError

    def step(self, action):
        raise NotImplementedError

    def render(self, mode="human"):
        raise NotImplementedError
```

Inheriting gym.Env is not required.

Initialize Simulator and Renderer

Just some boilerplate code

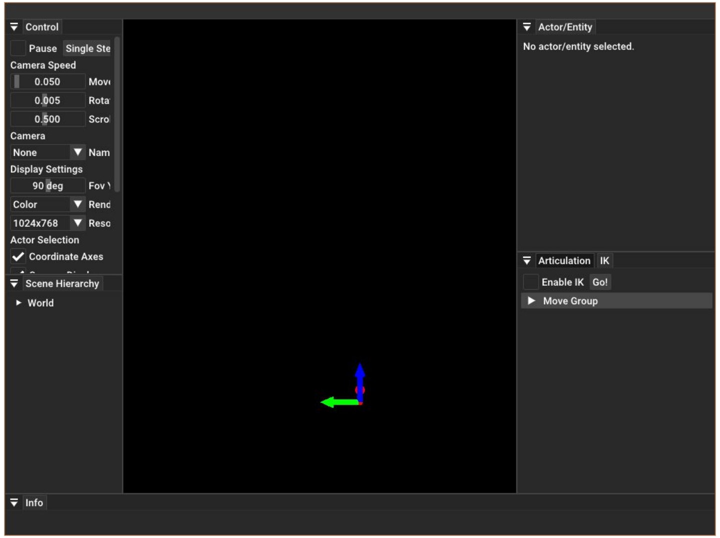
```python
import sapien.core as sapien
```

```python
def __init__(self, simulation_frequency=500):
    self.engine = sapien.Engine()
    self.renderer = sapien.VulkanRenderer()
    self.engine.set_renderer(self.renderer)
    self.scene = self.engine.create_scene()
    self.scene.set_timestep(1/simulation_frequency)
```

Visualization!

```python
    def render(self, mode="human"):
        if not hasattr(self, 'viewer') or self.viewer is None:
            from sapien.utils import Viewer
            self.viewer = Viewer(self.renderer)
            self.viewer.set_scene(self.scene)
            self.viewer.set_camera_xyz(-1, 0, 0.5)

        self.viewer.render()
```

```python
def main():
    env = OpenCabinetEnv()
    while True:
        env.render()
        if env.viewer.closed:
            break


if __name__ == "__main__":
    main()
```



Load the Assets

```python
self.scene.set_ambient_light(color=[0.3, 0.3, 0.3])
self.scene.add_directional_light(
    direction=[-0.3, -0.3, -1], color=[1, 1, 1], shadow=True
)
```

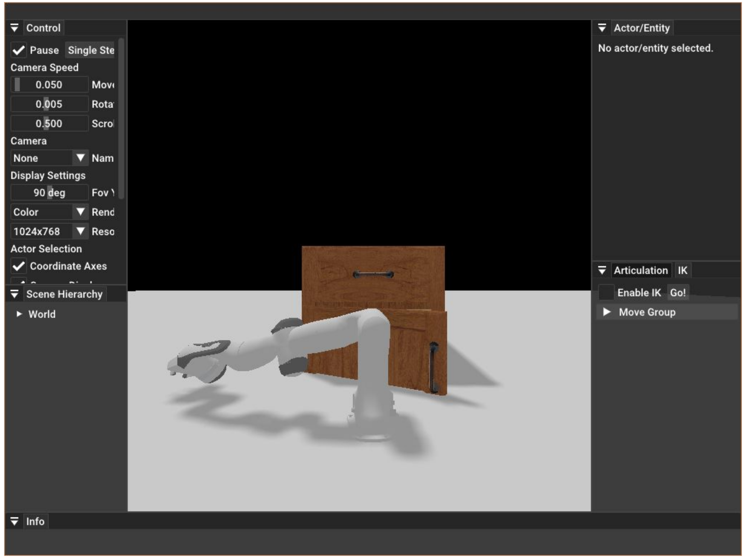
Add lights in init, so we can see the objects.

```python
self.scene.add_ground(altitude=0, render=True)
loader = self.scene.create_urdf_loader()
loader.fix_root_link = True
self.robot = loader.load("../assets/panda.urdf")
self.robot.set_pose(sapien.Pose([-1, 0, 0]))
self.cabinet = loader.load("../assets/45146/mobility.urdf")
```

Load ground, robot and cabinet into the scene.

```python
def main():
    env = OpenCabinetEnv()
    while True:
        env.scene.step()
        env.render()
        if env.viewer.closed:
            break
```

Since env.step is not implemented, add env.scene.step to the rendering loop for debugging.

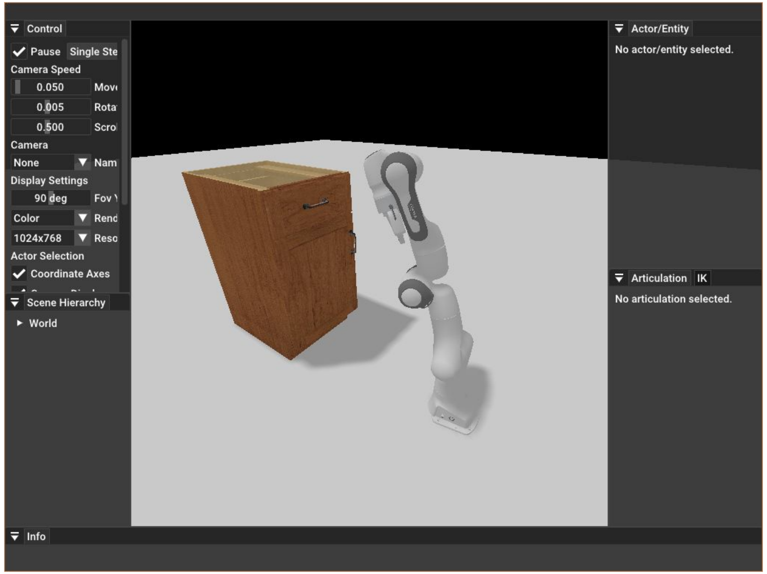


The position and scale of the cabinet is not reasonable.

Debug the Assets

It is possible to compute the bounding box in SAPIEN and "do it right". Here for simplicity I use manually-tuned numbers.

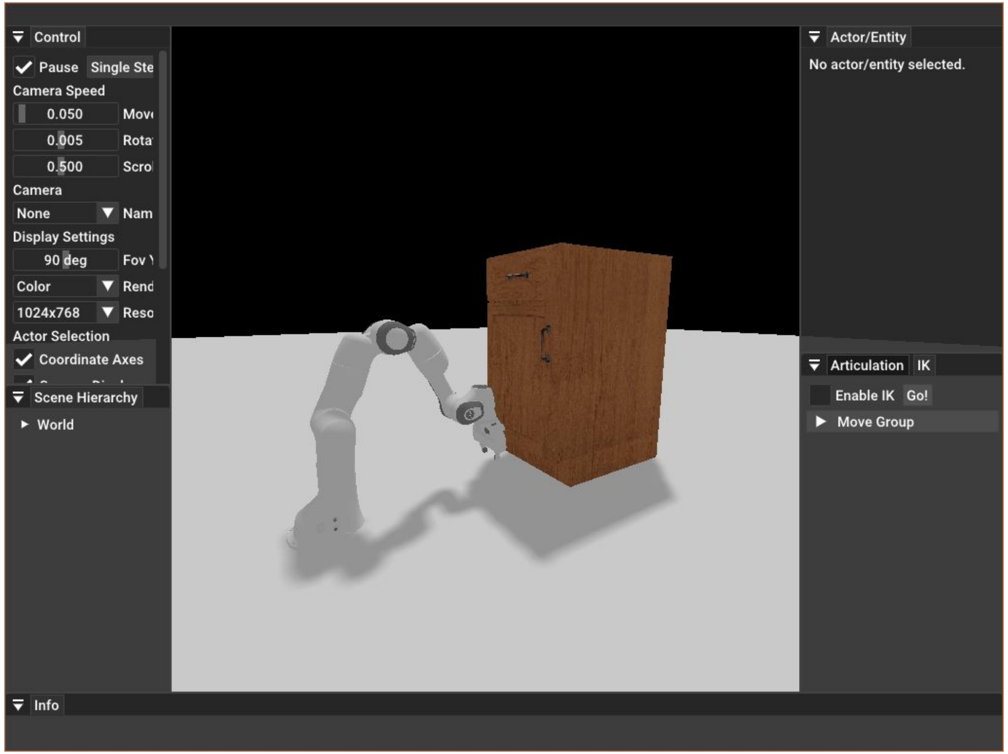
```
loader.scale = 0.6
self.cabinet = loader.load(
    "../assets/45146/mobility.urdf")
self.cabinet.set_pose(
    sapien.Pose([0, 0, 0.5]))
```



Implement Initial State (Reset)

```python
def _get_observation(self):
    return None

def reset(self):
    # initial cabinet joint positions
    self.cabinet.set_qpos([0, 0])

    # intiail robot base pose, may be randomized
    self.robot.set_pose(sapien.Pose([-1, 0, 0]))

    # initial robot joint positions, may be randomized
    qpos = [0, 0.345, 0, -2.25, 0, 2.75, 0.78, 0.04, 0.04]
    self.robot.set_qpos(qpos)

    # damping may require further fine tuning later
    joints = self.robot.get_active_joints()
    for joint in joints[:-2]:  # arm joints
        joint.set_drive_property(stiffness=0, damping=300)

    for joint in joints[-2:]:  # finger joints
        joint.set_drive_property(stiffness=0, damping=10)

    self.robot.set_drive_velocity_target(np.zeros(len(joints)))

    return self._get_observation()
```

```python
env = OpenCabinetEnv()
env.reset()
```



Implement Step Function

We also need to decide how many simulation steps to take in env.step after an action is taken.

For example, here we run the simulation at 500 Hz, if we want a 20 Hz action, we should step the simulation 25 times in the environment step function.

```
def __init__(self, simulation_frequency=500, action_frequency=20):
    self.substeps = simulation_frequency // action_frequency
```

```
def step(self, action):
    # do stuff
    for substep in range(self.substeps):
        # do stuff
        self.scene.step()
```

Implement Step Function

PD velocity control augmented with inverse dynamics

```python
def step(self, action):
    self.robot.set_drive_velocity_target(action)
    for substep in self.substeps:
        passive_force = self.robot.compute_passive_force(
            gravity=True, coriolis_and_centrifugal=True
        )
        self.robot.set_qf(passive_force)
        self.scene.step()

    return (
        self._get_observation(),
        self._get_reward(),
        self._get_done(),
        self._get_info(),
    )
```

Implement the Observation

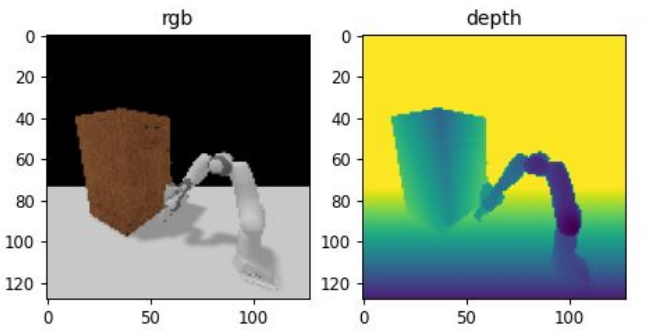
```python
self.camera = self.scene.add_camera(
    name="camera",
    width=128,
    height=128,
    fovy=np.pi / 2,
    near=0.01,
    far=2.0
)
self.camera.set_local_pose(
    sapien.Pose(
        [-1, 0.75, 0.75],
        [ 0.86988501,
          0.05311156,
          0.09607517,
         -0.48088336]))
```

Add camera

```python
if mode == "rgbd":
    self.scene.update_render()
    self.camera.take_picture()
    color = self.camera.get_float_texture("Color")[..., :3]

    # channel 2 is z depth, channel 3 is 01 depth
    depth = self.camera.get_float_texture("Position")[..., [3]]

    return np.concatenate([color, depth], 2)
```

Add to render function

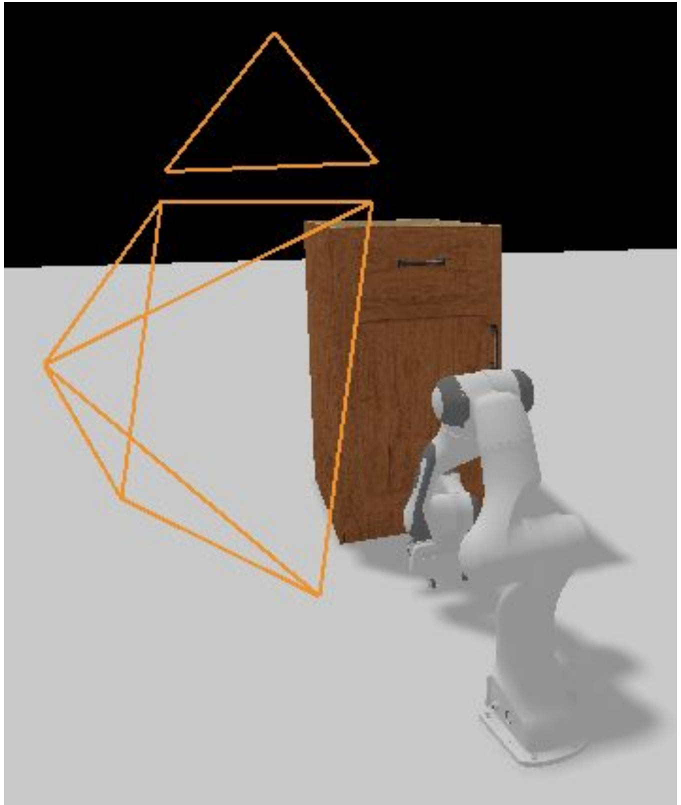
```python
rgbd = env.render("rgbd")
import matplotlib.pyplot as plt
plt.subplot(121)
plt.imshow(rgbd[..., :3])
plt.title("rgb")
plt.subplot(122)
plt.imshow(rgbd[..., 3])
plt.title("depth")
plt.show()
```



Visualization

Implement the Observation

SAPIEN also draws the added camera in the viewer for easier debugging.



Implement the Observation

Finish the observation function

```
def _get_observation(self):
    rgbd = self.render("rgbd")
    qpos = self.robot.get_qpos()
    qvel = self.robot.get_qvel()

    return np.concatenate([qpos, qvel]), rgbd
```

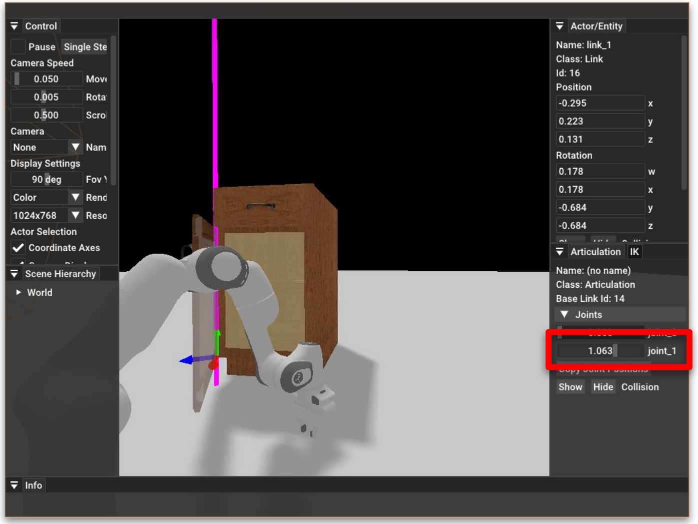
Reward

- Sparse reward
  - 1 when success, 0 otherwise
  - E.g., 1 if the door is opened to 60 degrees
- Dense reward
  - Give reward based on heuristics
  - E.g., proportional to the cabinet door angle
  - E.g., give reward if the gripper is close to the handle

Reward

Debugging: manually drag the door to success condition, and see if the reward is implemented correctly.

```python
def _get_reward(self):
    # sparse reward
    if self.cabinet.get_qpos()[1] > np.pi / 3:
        return 1
    else:
        return 0

    # dense reward
    # return self.cabinet.get_qpos()[1]
```



Drag the slider and print the reward.

Done and Info

env.step can set done to True when the task is completed.

It is also okay to never set done to True.

```
def _get_done(self):
    return self.cabinet.get_qpos()[1] > np.pi / 3

    # it is okay to never have a done
    # return False

def _get_info(self):
    return None
```

Action Space

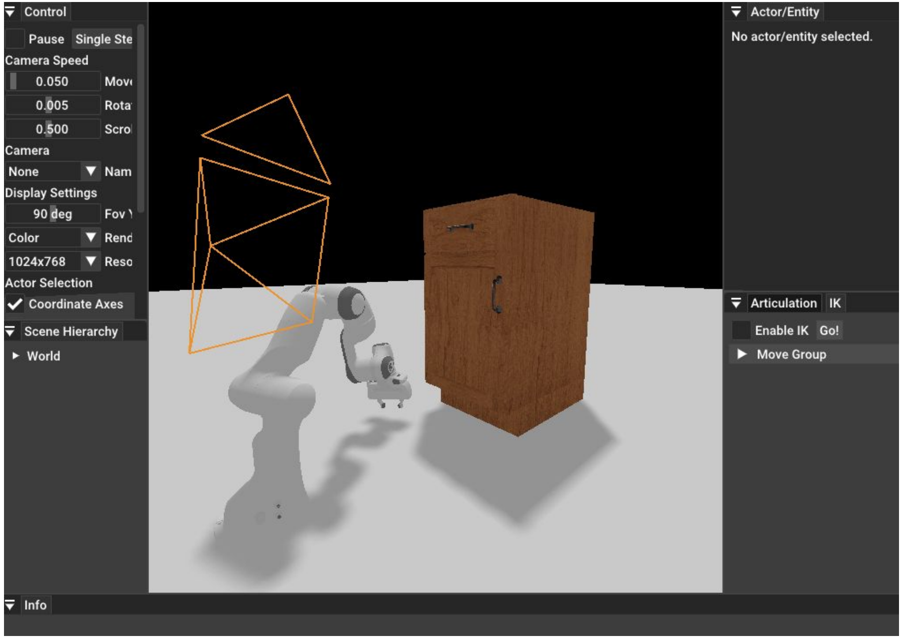
- For many algorithms, it is important to normalize the actions.
- Therefore, the environment should specify the range of the actions.

- Here the range for target velocity is [-0.2, 0.2] for all joints.

```
class OpenCabinetEnv(Env):
    action_space = gym.spaces.Box(np.ones(9) * -0.2, np.ones(9) * 0.2)
```

Test Random Actions

```
def main():
    env = OpenCabinetEnv()
    env.reset()

    while True:
        random_action = env.action_space.sample()
        obs, reward, done, info = env.step(random_action)
        env.render()
        if env.viewer.closed:
            break
```



Code will be released at https://github.com/haosulab/cvpr-tutorial-2022 after this tutorial.

Are we Done?

Are we Done?

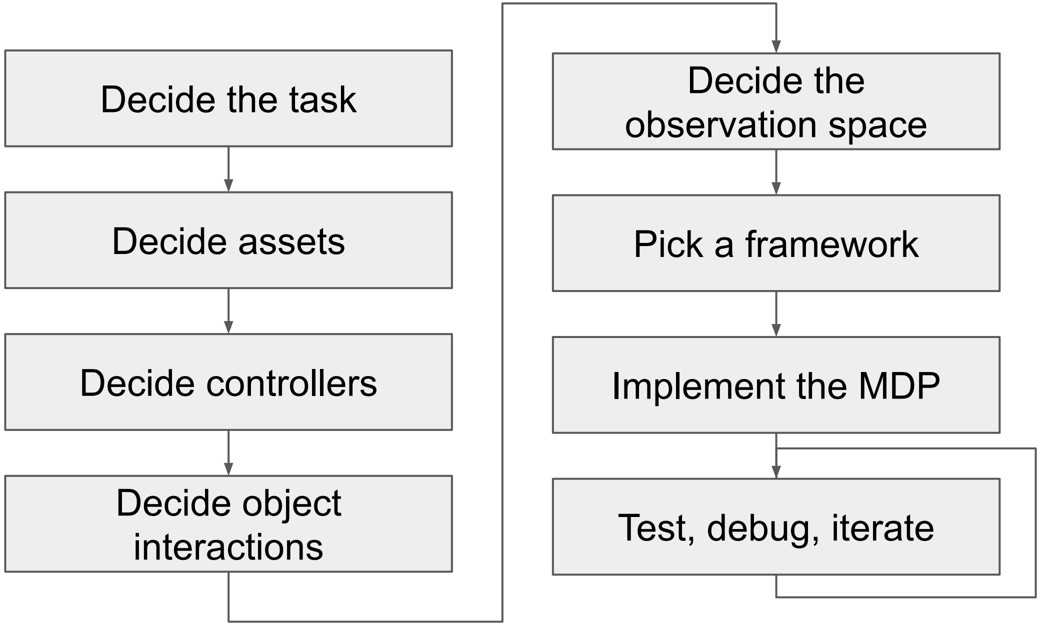
No.

Iterate

- Developing environments is an iterative process
  a. Develop the environment following this tutorial.
  b. Find issues later when solving the environment.
  c. Analyze the issue. Modify the environment if needed.
     - (E.g. The environment above probably will not work because I used default friction, which is too small for the gripper to hold firmly)
- We introduce common issues in the later section **Experiences and Practices to Debug Simulators**

Summary

- Build an environment



Q & A

- Contact: Fanbo Xiang (fxiang@eng.ucsd.edu)