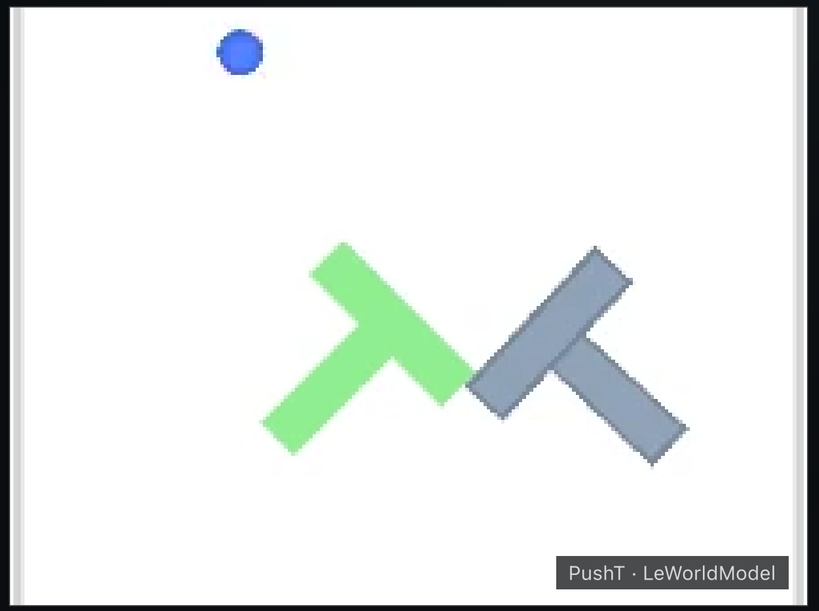
CIS 6280 · FALL 2026

WORLD MODELS

LECTURE 3

Environments · Simulators · Rollouts

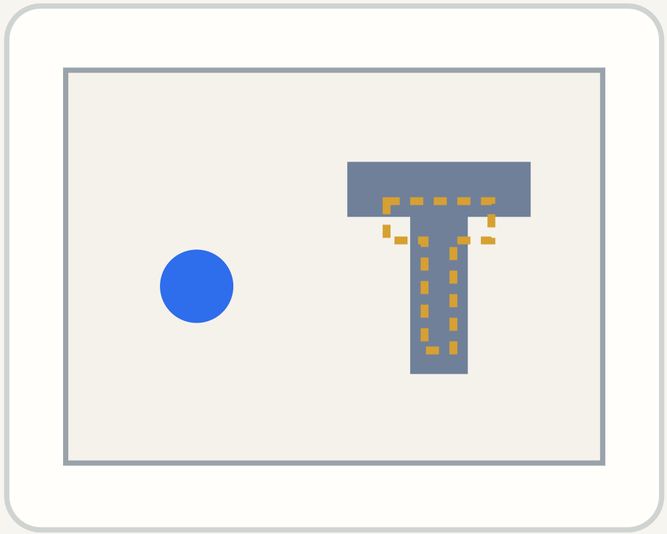


Jiatao Gu

University of Pennsylvania · September 1, 2026

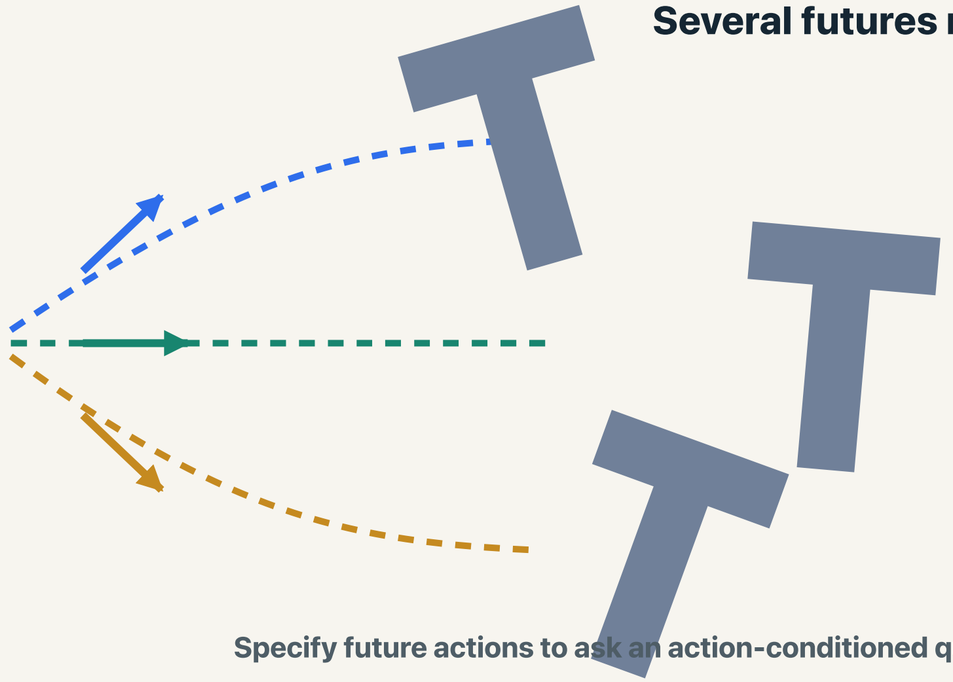
What Could Happen Next?

History available now



What futures are plausible?

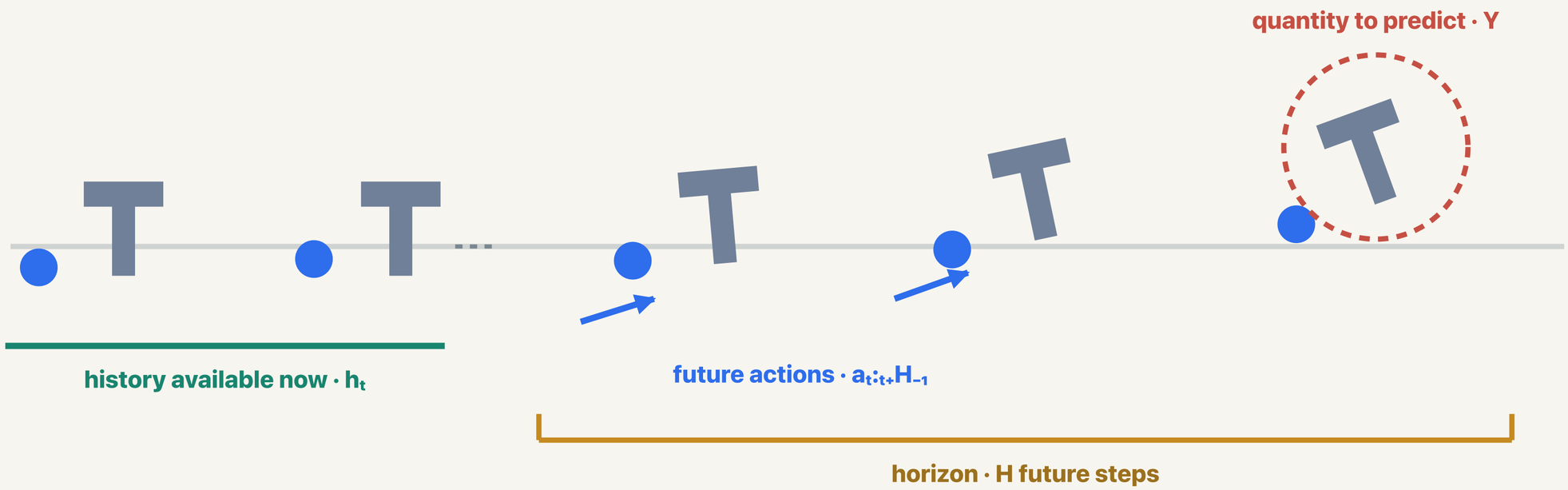
Several futures may fit the same history



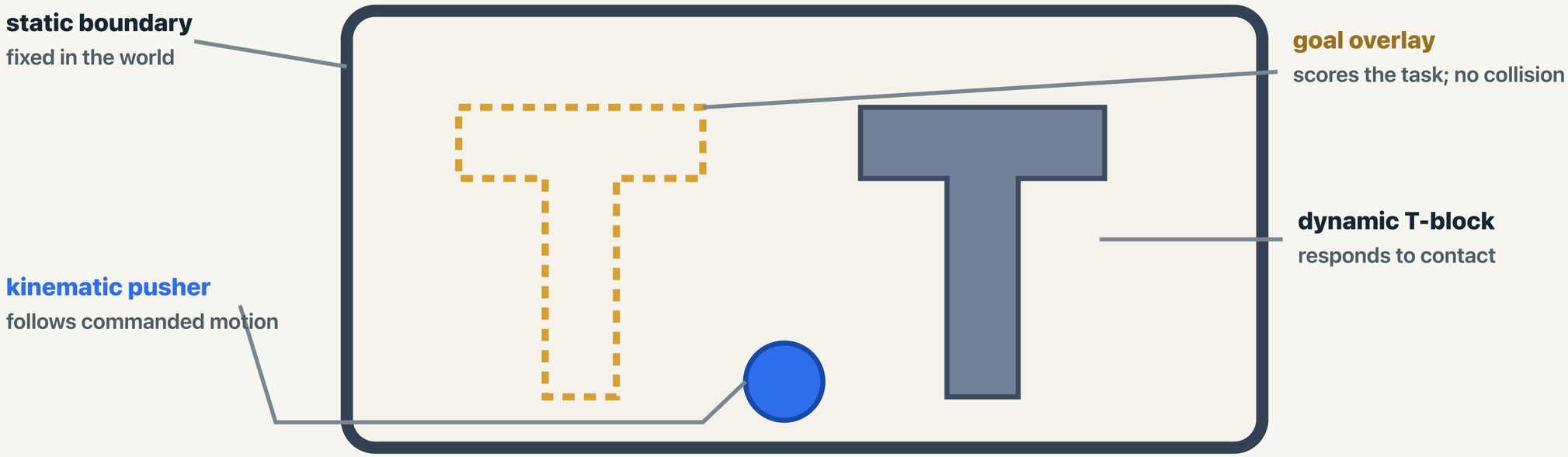
Specify future actions to ask an action-conditioned question.

Four Choices Define a Future Query

$$p_\theta(Y_{t+1:t+H} \mid h_t,\ a_{t:t+H-1})$$

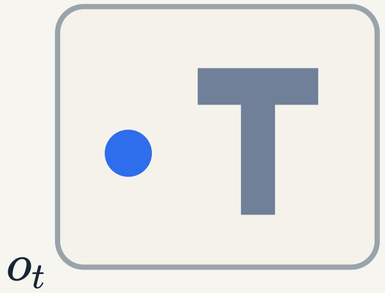


The Physical World in PushT



A rigid body keeps its shape while the whole body translates and rotates.

Where Does the Next Observation Come From?

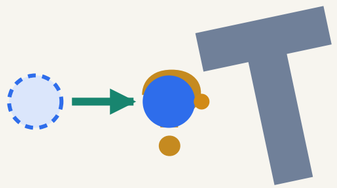


o_t

a_t

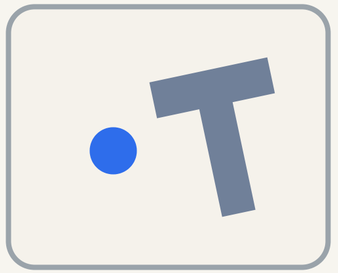
apply action

action: move pusher right



contact moves + rotates the T

o_{t+1}



Reality or a simulator must produce this transition.

The Simulator Advances State, Then Produces Pixels

COMPLETE INTERNAL SIMULATOR STATE

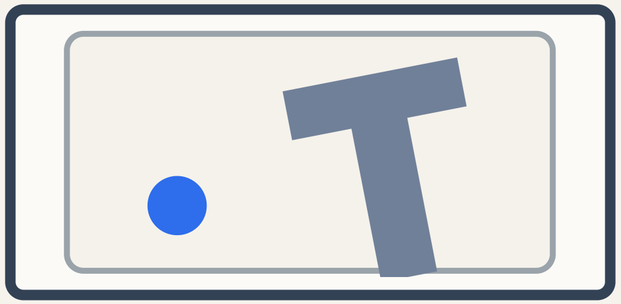
s_t contains different bodies with different roles

Kinematic pusher	position $\mathbf{p}_t$ and velocity $\mathbf{v}_t$ **controller** prescribes its motion
Dynamic T-block	pose (x_t, y_t, ϕ_t), linear velocity, angular velocity ω_t **contact physics** determines its response

render / sense →

AGENT OBSERVATION

o_t = pixels

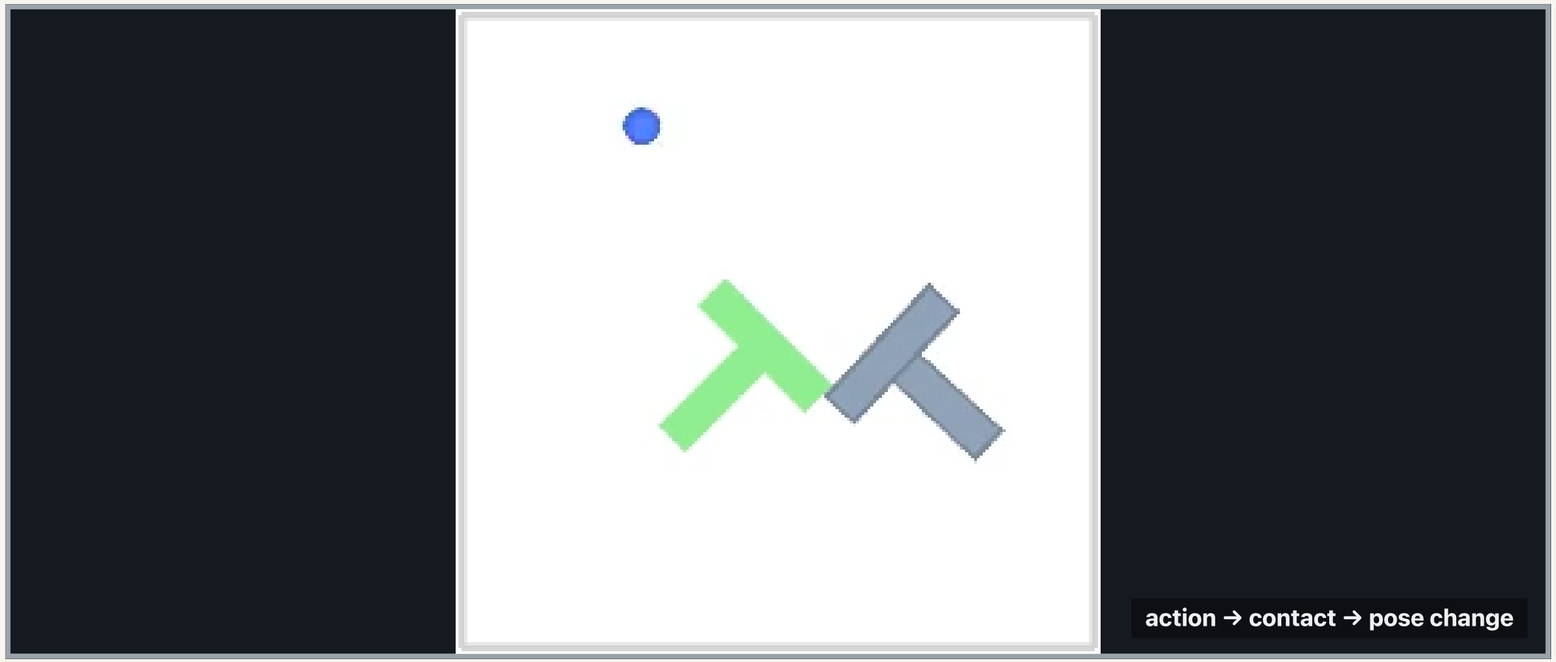


Shows appearance, but may hide velocity and body role.

physics: $(s_t, a_t) \rightarrow s_{t+1}$

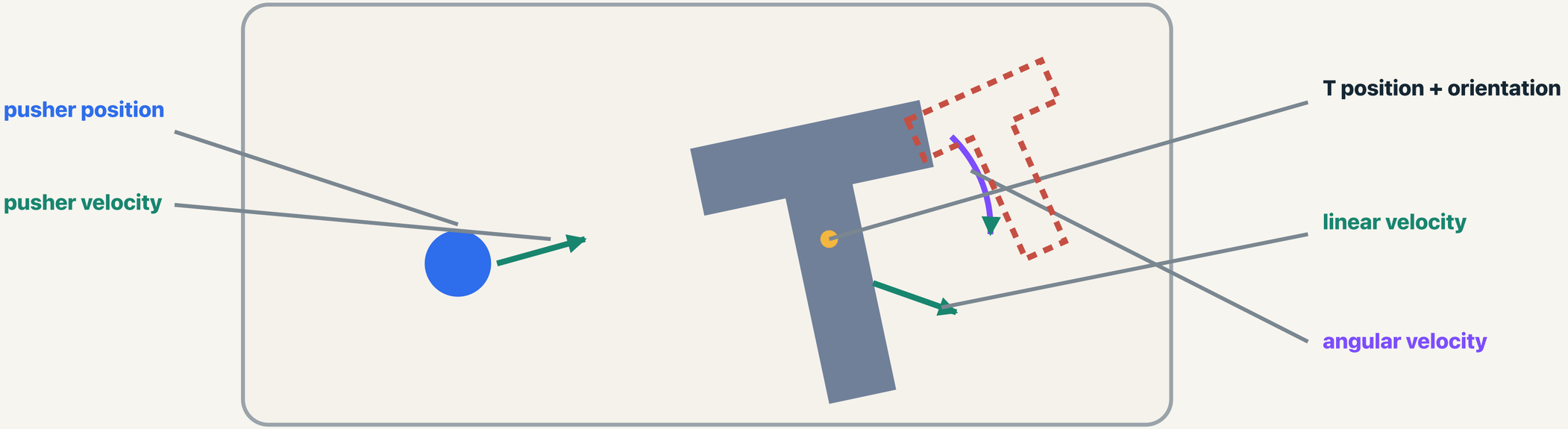
observation: o_t = render / sense(s_t)

One Push in PushT



Watch the contact: why does the T both translate and rotate?

Pose and Motion

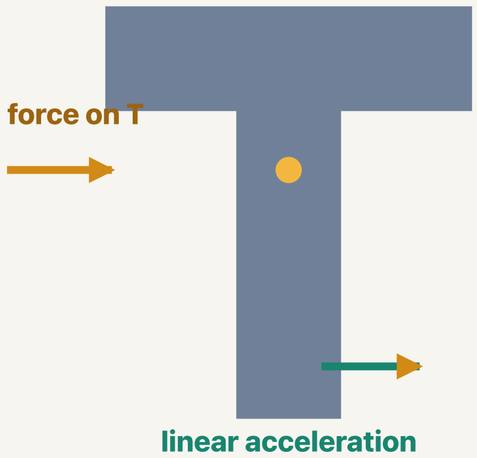


same pose now + different velocity → different next pose

For this simulator, s_t stores both pose and motion.

Translation and Rotation

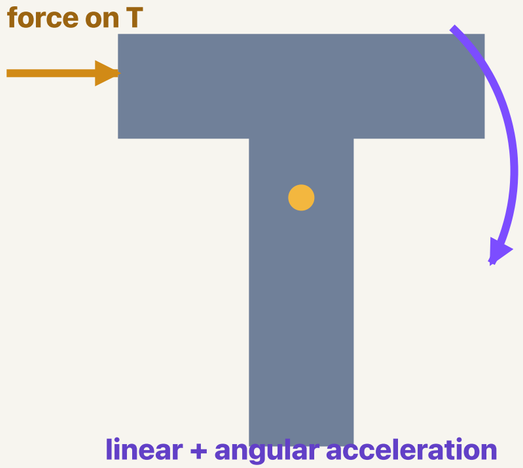
force through the T center



$\mathbf{F} = m\mathbf{a}$

Force changes the T's linear motion; mass resists that change.

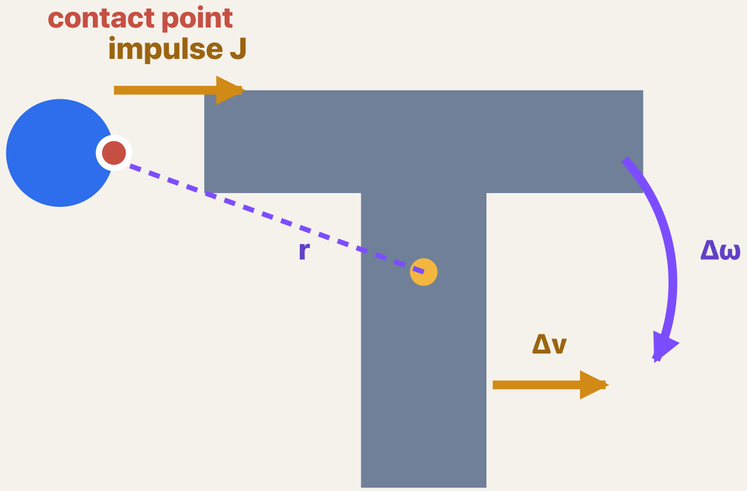
off-center force



$\tau = I\alpha$

Torque is the turning effect of force; rotational inertia resists angular acceleration.

Contact Transfers Motion



$$J = \int F \, dt$$

Impulse is the contact force accumulated over a very short time.

$$\Delta v_T = J / m_T$$
$$\Delta \omega_T = (r \times J) / I_T$$

J changes translation. An off-center impulse, with lever arm **r**, also changes spin.

Rigid-body simulation compresses a short, deformable real contact into an instantaneous impulse approximation.

`env` Is an Object That Owns a World

```
import gymnasium as gym
class PushTEnv(gym.Env):
    def reset(self, *, seed=None, options=None):
        self.state = make_initial_state()
        return observe(self.state), {}
    def step(self, action):
        self.state = transition(self.state, action)
        obs = observe(self.state)
        return obs, reward(self.state), \
               False, False, {}
env = PushTEnv()
```

CLASS

`PushTEnv(gym.Env)`

Defines a world behind Gymnasium's standard reset/step interface.

OWNED STATE

`self.state`

The environment object stores the world that actions will change.

INSTANCE

`env = PushTEnv()`

`env.step(action)` calls the step method on this particular world.

Inside One `env.step(action)`

STATE + ACTION

(s_t, a_t)

Describe the world and choose a push.

→

TARGET

$p^\star$

Convert the action into a pusher target.

→

CONTROL

a_{PD}

Repeatedly correct the pusher motion.

→

MOTION

integrate

Advance velocity and position by `dt`.

→

CONTACT

detect

Find where the pusher touches the T.

→

RESPONSE

J

Update translation and rotation.

→

NEXT STATE

s_{t+1}

Return observation, reward, and termination.

> PushT is our compact route through a full physics-simulation pipeline.

A Physics Simulator Is a World Model

$$s_{t+1} = F_{\text{physics}}(s_t, a_t; \theta)$$

state + action + physical parameters → next state

Predictive: it answers what happens after an action.

Executable: we can call it repeatedly to generate a future.

Physics-based: its transition rule is specified by mechanics, geometry, and numerical algorithms—not learned from data.

Still a model: rigid bodies, friction, time steps, and parameters are approximations of reality.

Why Run the World in Software?

CONTROL

Choose the initial state

Repeat exactly the same condition, including hidden velocity and random seed.

INTERVENTION

Try another action

Branch from one state and compare counterfactual futures.

SCALE

Generate rollouts

Run many trials faster, cheaper, and more safely than on a robot.

INSPECTION

See every variable

Read pose, velocity, contacts, and forces that a camera may hide.

> Physics laws make a compact transition rule possible; numerical simulation turns that rule into repeated experiments.

Roles Persist Across Different Software Implementations

DECISION MAKER

Policy

Reads an observation and selects an action.

ROLE 1

Environment interface

Defines actions, observations, reward, reset, and termination.

This lecture: a Gymnasium-compatible PushT environment
Could instead be: another API or a robot bridge

ROLE 2

Transition mechanism

Advances the internal state using physics or another world process.

This lecture: Pymunk rigid-body simulation
Could instead be: another engine or the real world

> Gymnasium and Pymunk are today’s concrete choices—not the definition of an environment or simulator.

The Action Is a 2D Relative Target Vector

$$a_t = (a_x, a_y)$$
$$p^\star = p_t + 100\, a_t$$

Two components: horizontal and vertical target displacement.

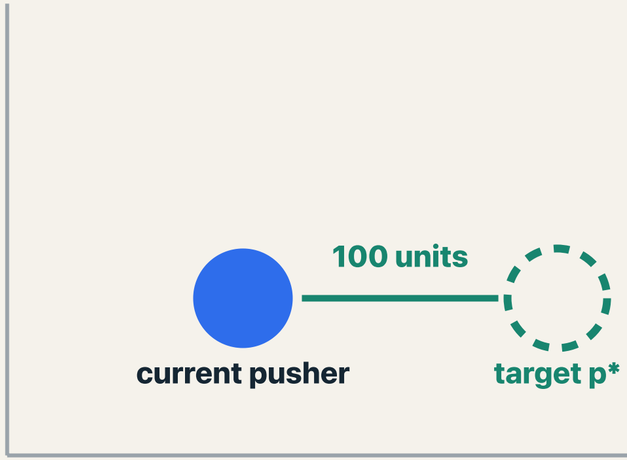
Dimensionless: the values are scaled by the environment; they are not pixels, Newtons, or an angle.

Relative: the target starts from the pusher's current position.

Held fixed: one target is used throughout the ten physics substeps of this action.

From Action to Pusher Target

$$p_t = (273, 195)$$
$$a_t = (1, 0)$$
$$p^\star = (273, 195) + 100(1, 0)$$
$$p^\star = (373, 195)$$



Pymunk Stores Bodies, Shapes, and a Space

BODY

Motion state

Position, angle, linear velocity, angular velocity, mass, and inertia.

SHAPE

Collision geometry

Circle or polygon geometry plus friction and elasticity, attached to a body.

SPACE

The simulated world

Holds bodies and shapes; `space.step(dt)` advances motion and resolves contacts.

BLUE PUSHER

kinematic body + circle

Controller prescribes motion.

→

T-BLOCK

dynamic body + polygons

Contact determines response.

→

PUSHT WORLD

one space

Every substep advances all registered objects.

Build the PushT World in Colab

```
import pymunk
# 1. The space is our 2D simulated world.
space = pymunk.Space()
# 2. The pusher: motion prescribed by control.
pusher_body = pymunk.Body(
    body_type=pymunk.Body.KINEMATIC)
pusher_shape = pymunk.Circle(pusher_body, 15)
# 3. The T-block: motion determined by physics.
t_body = pymunk.Body(mass=1.0, moment=moment)
t_shape = pymunk.Poly(t_body, vertices)
# 4. Register objects before stepping the world.
space.add(pusher_body, pusher_shape,
          t_body, t_shape)
```

WATCH FOR

Body + Shape

Motion state and collision geometry are created separately.

THEN

Add to Space

The engine can only update objects registered in the simulated world.

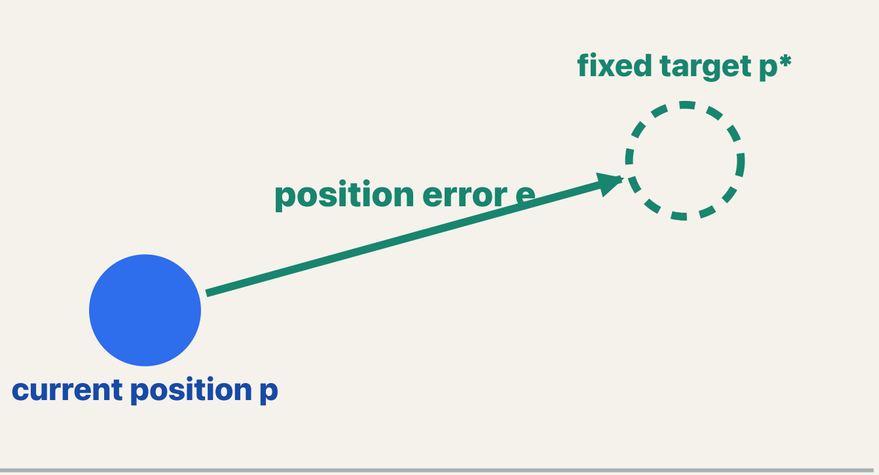
NEXT ACT

Move toward p*

We will add the feedback controller and step the same world.

Open the course Colab ↗

How Does the Pusher Reach the Target?



$$e = p^{\star} - p$$

The error points from the current pusher position toward the target.

A target is a reference

The controller must repeatedly turn the remaining error into motion.

measure error → correct motion → measure again

Feedback Corrects the Motion at Every Step

target p* stays fixed

OBSERVE
Read p and v
Where is the pusher now, and how fast is it moving?

→

COMPARE
Compute error
Use the latest position.

→

CORRECT
Update motion
Use the current error and velocity.

→

ADVANCE
Step the world
Obtain a new p and v.

read the new state and repeat

One action holds p* fixed; feedback is recomputed from the latest state.

P Responds to Position Error

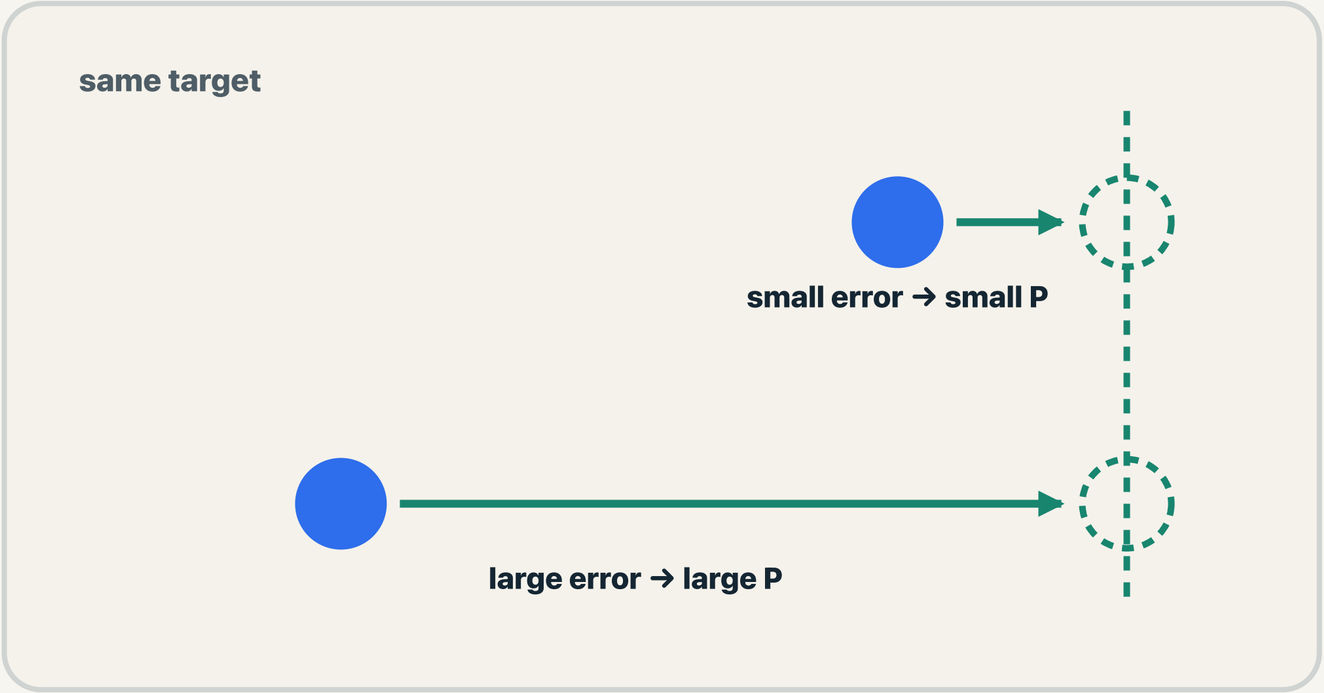
P MEANS

Proportional

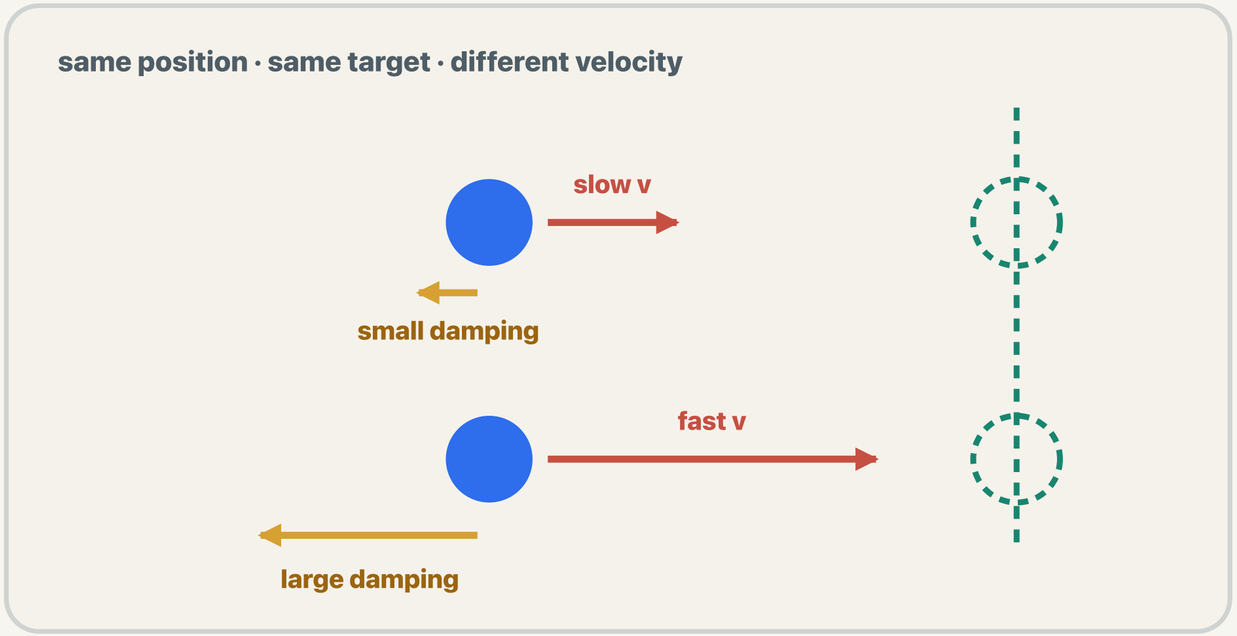
$$a_P = k_p e = k_p(p^\star - p)$$

The correction points toward the target and scales with the remaining distance.

k_p is a gain. A larger value makes the controller pull toward the target more aggressively.



D Responds to the Current Velocity



D MEANS

Derivative

$$\dot{e} = -v$$

$$a_D = k_d \dot{e} = -k_d v$$

Because the target is fixed, moving toward it reduces the error. The D term pushes against that velocity.

k_d is a gain. A larger value applies stronger damping and suppresses overshoot.

The PD Controller Produces an Acceleration Command

This PushT implementation combines the two corrections:

$$a_{\text{cmd}} = k_p(p^\star - p) - k_d v$$

P · target pull

$k_p(p^\star - p)$

D · velocity damping

$-k_d v$

A visible first step

$e = 50\ \text{px}, \quad v = 0$

$k_p = 100\ \text{s}^{-2}, \quad k_d = 20\ \text{s}^{-1}$

$a_{\text{cmd}} = 100(50) - 20(0) = 5000\ \text{px/s}^2$

Controller ownership

The controller reads the current pusher state and computes a command. The simulator has not advanced yet.

PD is a feedback structure

Acceleration is the output chosen by this implementation. Other robotics controllers can use the same P and D structure to command:

torque | **velocity** | **force**

The Pusher Is Kinematic in Pymunk

KINEMATIC BODY

Blue pusher

The controller prescribes its motion. Contact does not determine the pusher's acceleration.

```
# course implementation
pusher.velocity += a_cmd * dt
```

For the pusher, the code updates velocity directly. It bypasses solving pusher acceleration from a force using $F = ma$.

DYNAMIC BODY

Gray T-block

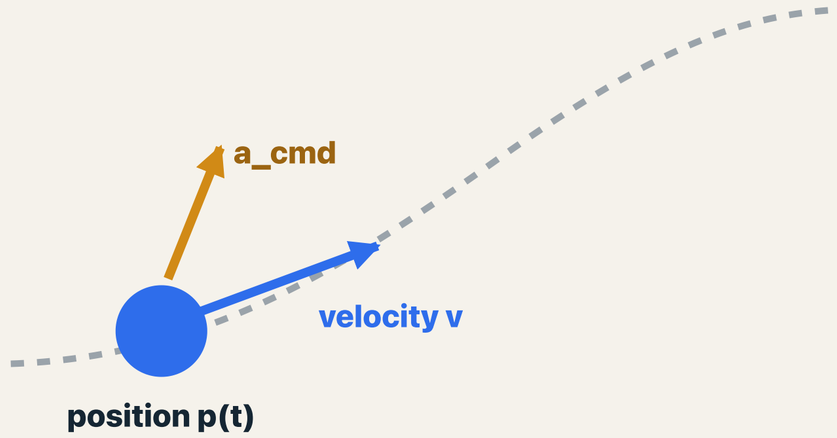
Pymunk determines its response from mass, inertia, existing motion, and contact impulses.

$$\Delta v_T = J / m_T$$
$$\Delta \omega_T = (r \times J) / I_T$$

The PD command is not applied directly to the T-block. The pusher must first make contact.

PD command → pusher velocity → contact impulse → T-block motion

Velocity and Position Evolve Over Time



A continuous curve describes what happens between stored frames.

Velocity is the rate of position change.

$$\dot{p} = v$$

Acceleration is the rate of velocity change.

$$\dot{v} = a_{\text{cmd}}$$

The dot means “change per second.” These two equations are the controller’s continuous-time motion model.

How a Computer Advances One Small Step

1 · UPDATE VELOCITY

$$v \leftarrow v + a_{\text{cmd}}\,dt$$

Acceleration accumulates over a short duration.

2 · UPDATE POSITION

$$p \leftarrow p + v\,dt$$

The updated velocity advances the pusher position.

WORKED FIRST STEP

$$e = 50\ \text{px}, \quad v = 0$$

$$a_{\text{cmd}} = 100e - 20v = 5000\ \text{px/s}^2$$

$$\Delta v = a_{\text{cmd}}\,dt = 50\ \text{px/s}$$

```
dt = 0.01 s
```

In the course code, the visible line updates velocity; `space.step(dt)` performs the position advance.

We Use Semi-Implicit Euler: Velocity First, Then Position

Same initial state, acceleration, and `dt`—but a different discrete transition.

Forward Euler

COMPARISON

EXPLICIT EULER · POSITION FIRST

$$p \leftarrow p + vdt$$
$$v \leftarrow v + a\,dt$$

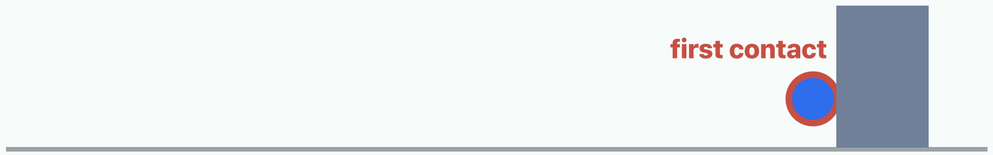
Acceleration changes *v* now, but position responds in the next substep.

Semi-Implicit Euler

USED HERE

SYMPLECTIC EULER · VELOCITY FIRST

$$v \leftarrow v + a\,dt$$
$$p \leftarrow p + vdt$$



The new velocity moves the pusher immediately and is usually more stable for mechanical motion at the same `dt`.

Why one substep matters: crossing a contact boundary switches $J = 0$ to $J \neq 0$, changing impulse, torque, and the later rollout.

One Action Contains Ten Physics Steps

1	2	3	4	5	6	7	8	9	10
PD step(dt)	PD step(dt)	PD step(dt)	PD step(dt)	PD step(dt)	PD step(dt)	PD step(dt)	PD step(dt)	PD step(dt)	PD step(dt)

action begins · target fixed

action ends after 100 ms

CONTROL CLOCK

10 Hz → 100 ms

The environment asks the policy for ten actions per second.

PHYSICS CLOCK

dt = 10 ms · K = 10

During one action, feedback is recomputed and `space.step(dt)` is called ten times.

100 ms is an interface choice, not a law of physics. A shorter interval reacts more often; a longer interval gives each action a longer effect and changes the rollout.

Who Updates What?

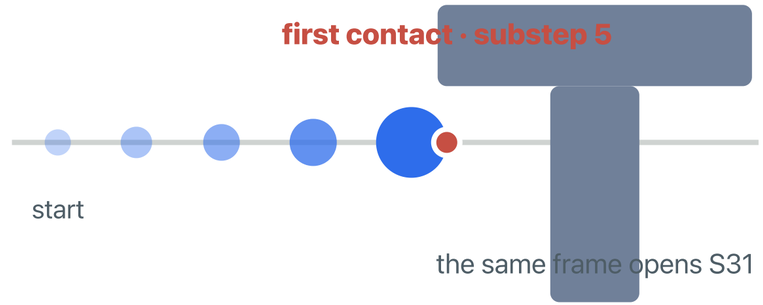
CONTROLLER	**Read p, v and compute a_{cmd}**	Uses the latest pusher state and the fixed target.
COURSE WRAPPER	`pusher.velocity += a_cmd * dt`	Integrates the acceleration-form controller.
PYMUNK SPACE	`space.step(dt)`	Advances every body, finds contacts, and changes the dynamic T through impulses.
ENVIRONMENT	**Read the next state**	The pusher and T-block both have updated pose and motion.

> Every substep contains one controller update and one complete `Space` step; `space.step(dt)` does not update only the T-block.

From Target to First Contact

```
# Once per 100 ms action:
target = pusher.position + 100 * Vec2d(*action)
dt = 0.10 / 10
# Ten physics substeps:
for step in range(10):
    # Feedback uses the latest pusher state.
    error = target - pusher.position
    a_cmd = kp * error - kd * pusher.velocity
    # Acceleration command changes velocity.
    pusher.velocity += a_cmd * dt
    # Pymunk advances every body and contact.
    space.step(dt)
    if first_contact():
        break  # freeze this frame for S31
```

FROZEN FIRST-CONTACT FRAME



Open step_action() + show_rollout() in Colab ↗

LIVE SIMULATION · PYMUNK 2D

Real Pymunk 7.3.0 ready

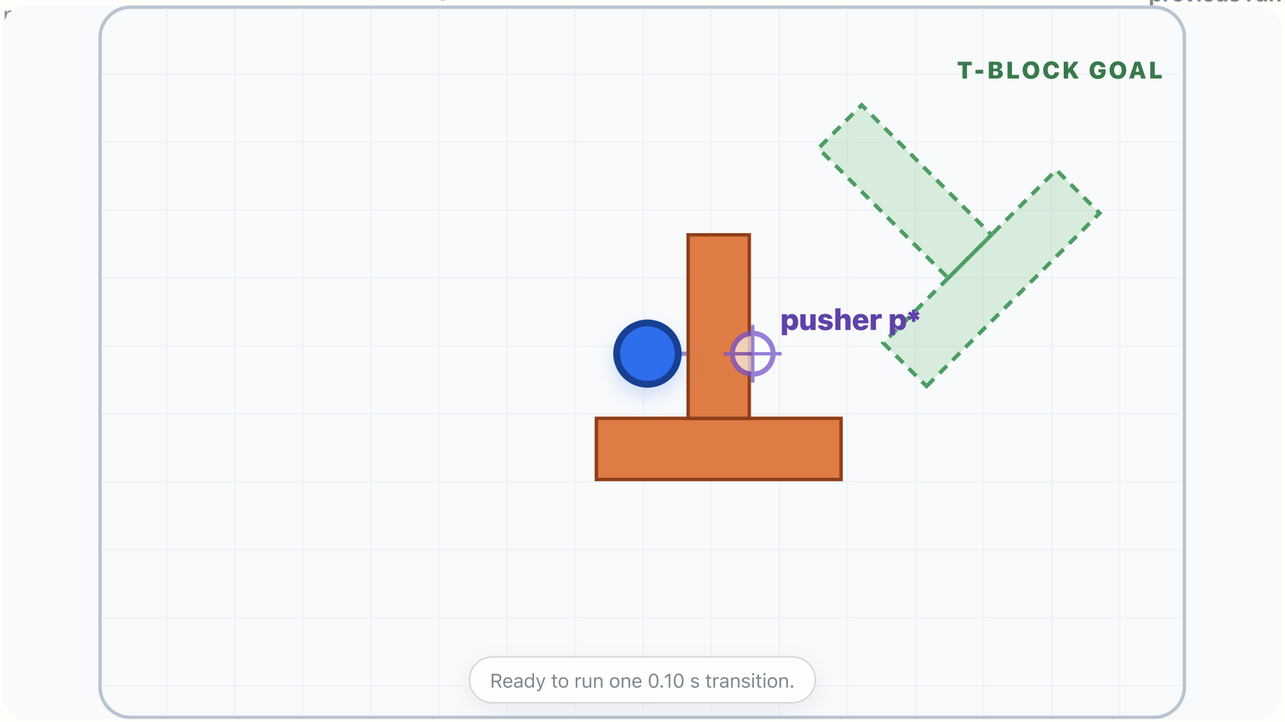
One action becomes a target; K physics steps produce the next state

One persistent world · one 0.10 s action at a time

pusher T-block T goal pusher p*

previous run

Run continues from the current state; Run again branches from the r

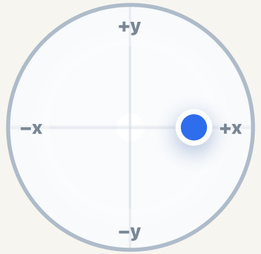


frame 0 / 10 · t = 0.00 s

PUSHER TRACKING	FIRST CONTACT	T OUTCOME
setpoint 62 px away · not en...	—	waiting

1 Action vector

direction + magnitude · release to select



ACTION

(0.62, 0.00)

center = zero action

2 Experiment code

edit Python values, then run the same world

editable config.py — reset code

```
# Edit values, then Run
action_scale = 100  # px / action
substeps = 10       # K in 0.10 s
block_mass = 1.0    # T mass
friction = 0.85     # contact
damping = 0.0       # 0=stop, 1=no drag
kp = 100            # pull to p*
kd = 20             # brake
```

p* shift = (62, 0) px — dt = 0.10 / 10 = 0.010 s

Run | Run again from previous state | Reset world

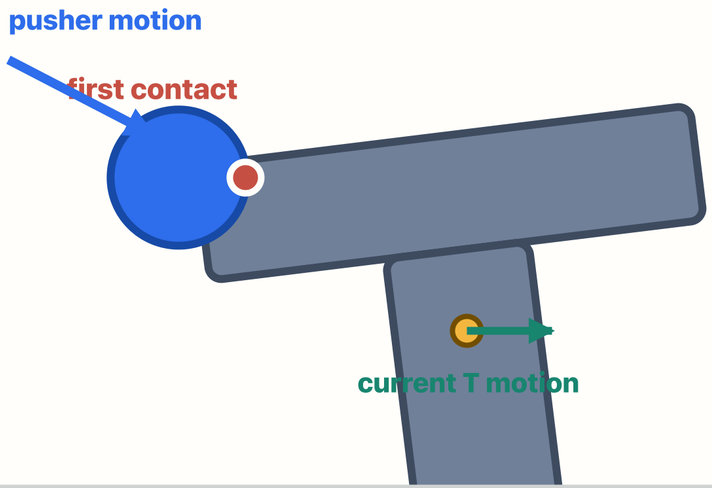
World reset. The next action starts from the canonical initial state.

action → pusher target → K × space.step(dt) → contact → next state

What Must Pymunk Know at First Contact?

Frozen at the first contact

simulated clock paused



Same scene, same instant, same state as S30

Now zoom from the visible motion into the engine.

GEOMETRY

Circle + T polygons

Their exact shapes and current poses tell Pymunk where contact can occur.

MOTION

Linear + angular velocity

The touching points may be moving toward each other, apart, or along the surface.

PHYSICAL PROPERTIES

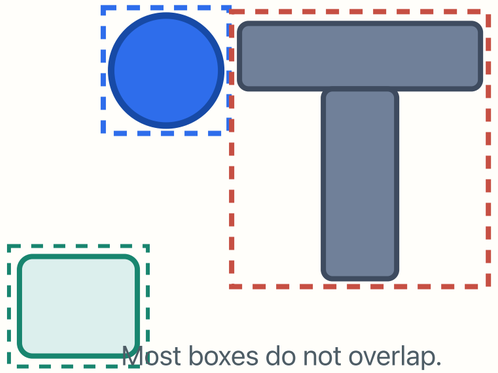
Mass, inertia, friction, elasticity

These determine how strongly each body responds after contact is found.

Pymunk reads geometry, motion, and material properties from the frozen state before it computes a response.

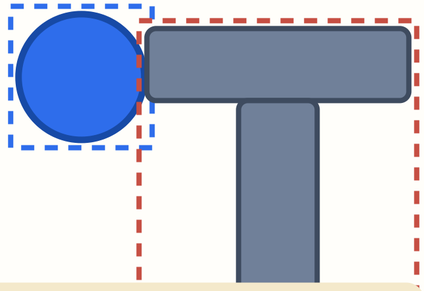
How Does Pymunk Locate the Contact?

1 · Give each shape a simple outer box



Most boxes do not overlap.

2 · Keep possible pairs

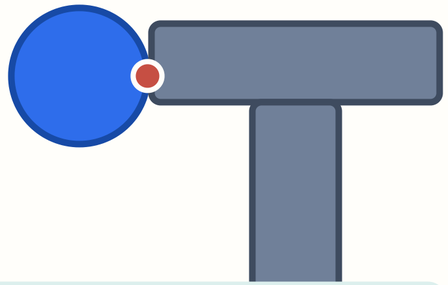


overlapping boxes → check this pair

possible contact, not proof

3 · Check the real boundaries

exact circle–polygon contact



output: local contact geometry

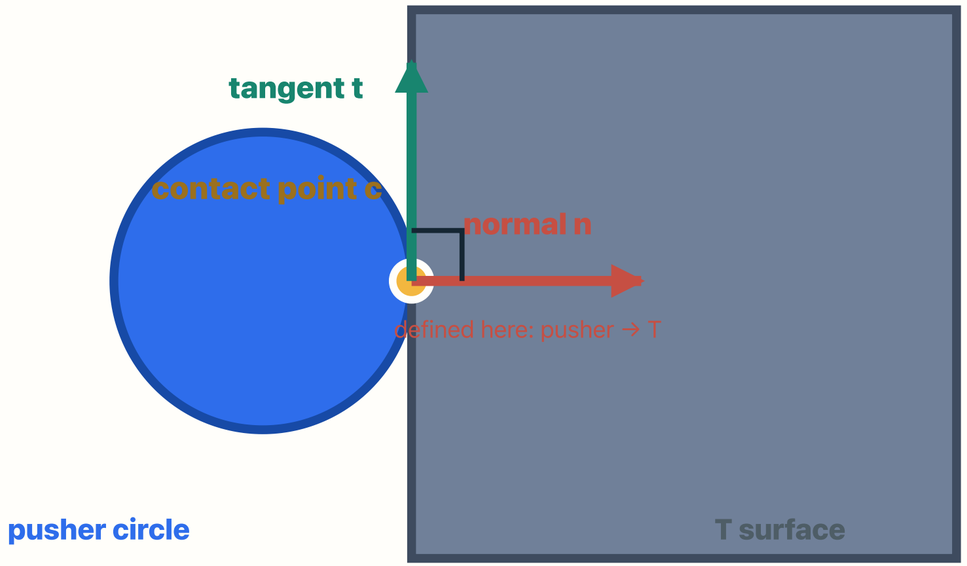
recognition term: **broad phase** = possible pairs

recognition term: **narrow phase** = exact shapes

> Pymunk spends exact geometry work only on pairs that might touch.

The Contact Point and Surface Normal

Microscope view of the exact same contact



Point c

Where the two exact boundaries meet.

Normal n

The local direction that separates the surfaces. Its sign follows the chosen body order.

Tangent t

A direction along the touching surface, perpendicular to n.

The normal is a geometry direction.
It does not report where either body is moving.

Are the Shapes Closing or Separating?

Velocity of the point on the T

$$v_{T,c} = v_T + \omega_T \times r$$

Translation and rotation both move the contact point.

Relative velocity at contact

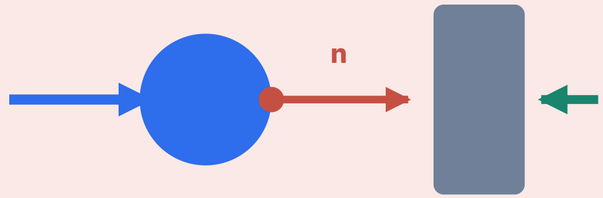
$$v_{\text{rel}} = v_T + \omega_T \times r - v_{\text{pusher}}$$

Keep only the normal component

$$v_n = v_{\text{rel}} \cdot n$$

One signed number tells whether this contact is approaching.

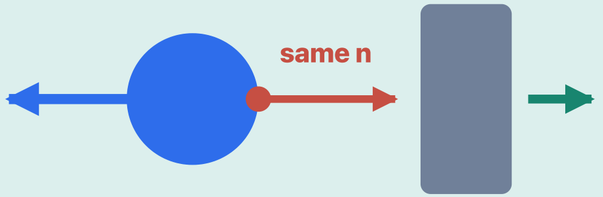
Closing · $v_n < 0$



distance is shrinking

normal response is needed

Separating · $v_n \geq 0$



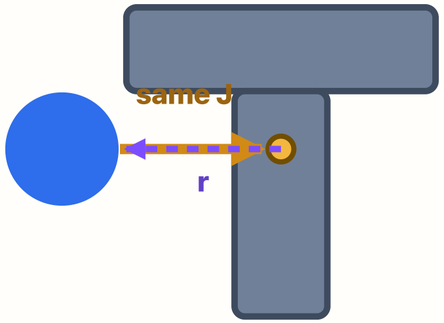
distance is growing

contact does not pull them back

The geometry supplies n; the current state supplies motion. Their dot product decides whether the contact is closing.

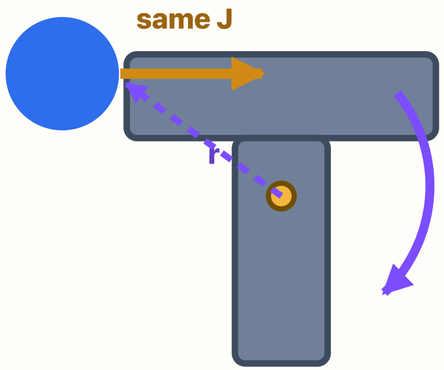
Where the Pusher Hits Changes the Rotation

Center hit



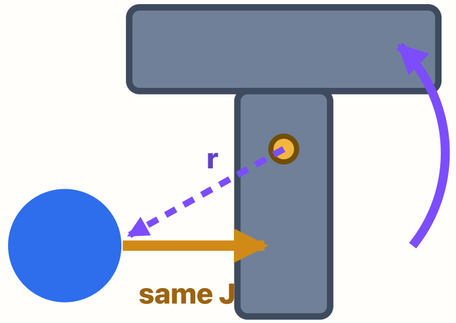
r × J ≈ 0 → almost no rotation

Upper hit



r × J < 0 → clockwise

Lower hit

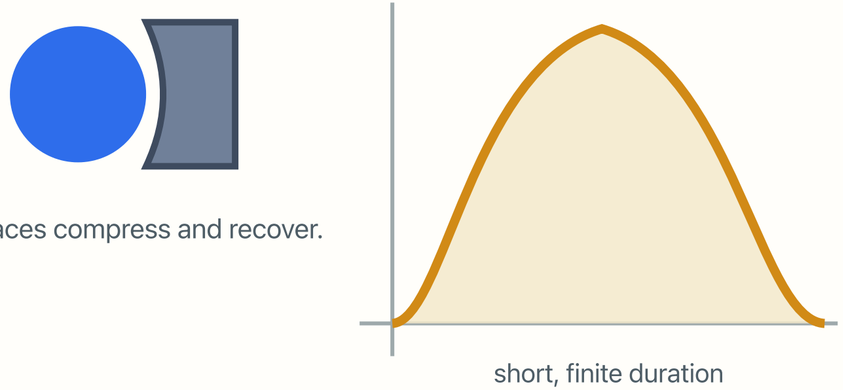


r × J > 0 → counter-clockwise

Contact location enters through r: the same rightward impulse can translate the T equally while changing its rotation.

From a Short Force Pulse to an Impulse

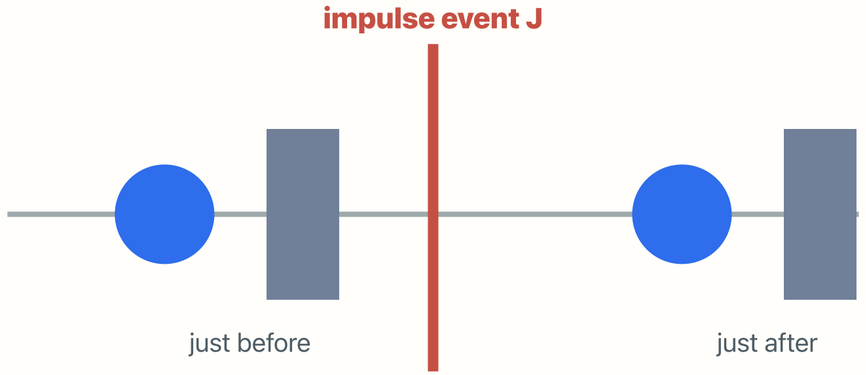
Real deformable contact



Surfaces compress and recover.

short, finite duration

Rigid-body approximation



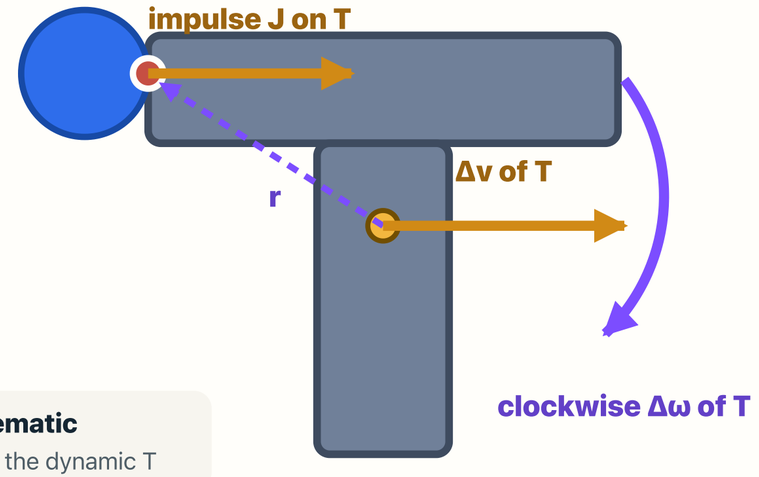
same pose at the event · velocity may jump

$$J = \int F(t)\,dt$$

same area, same accumulated push

A velocity jump is a reasonable rigid-body approximation. Real materials deform over time; this model preserves the total impulse while omitting deformation, vibration, sound, and heat.

The Impulse Updates Translation and Rotation



pusher is kinematic
these equations update the dynamic T

$$\Delta v_T = J / m_T$$

The T's center velocity changes in the impulse direction.

$$\Delta \omega_T = (r \times J) / I_T$$

Because the hit is above center, the T also turns clockwise.

One contact impulse updates both parts of the T state: linear velocity and angular velocity.

How Much Impulse Is Needed?

START WITH A CENTER HIT

closing speed	$v_n = -2$ m/s
T mass	$m = 2$ kg
rebound	$e = 0$

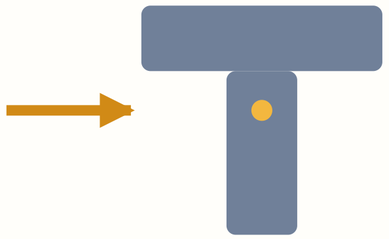
For a center hit, the impulse changes only the contact's straight-line speed:

$$j = -m\,v_n = 4\ \mathrm{N\,s}$$

That removes the 2 m/s inward speed without adding rebound.

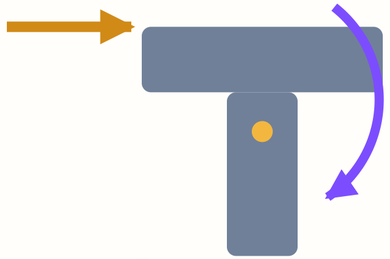
Contacts can push, not pull. The max with zero makes j = 0 when the bodies are already separating.

Center contact



all response is translation

Off-center contact



translation + rotation

Effective mass at this contact means: how easily can this touching point move along n?

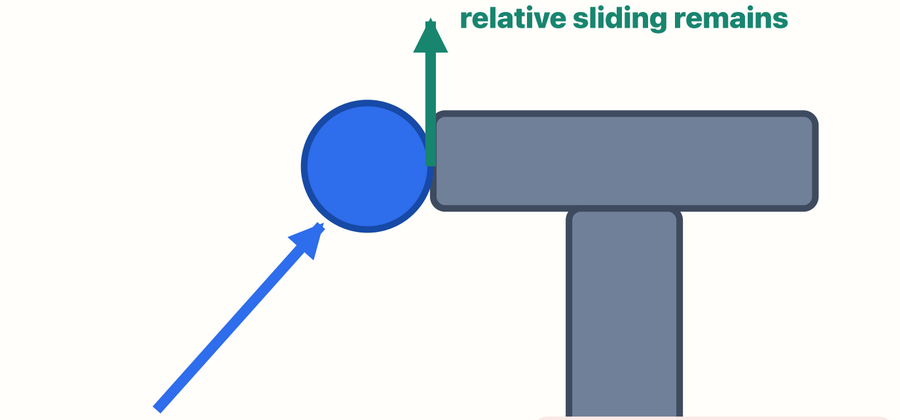
$$j = \max\left(0,\ \frac{-(1+e)v_n}{1/m + (r \times n)^2/I}\right)$$

The rotational term appears because an off-center impulse can also turn the T.

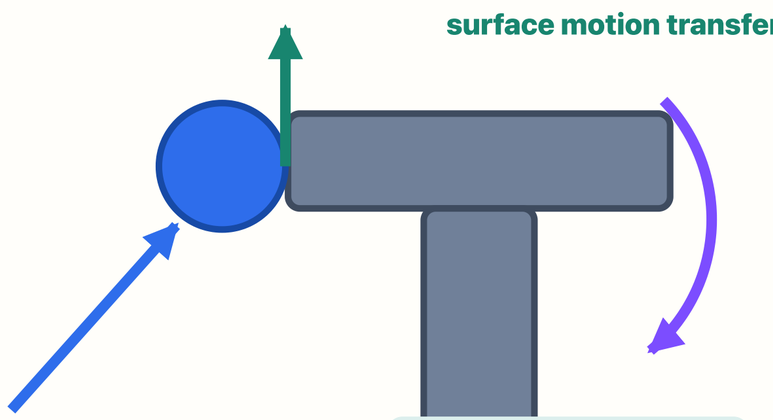
The engine chooses enough impulse to remove inward relative motion, accounting for both translation and rotation.

What Happens Along the Surface?

Low friction



Higher friction



Along the surface

Tangent t

The direction of relative sliding at this contact.

Brief sideways push

Friction impulse J_t

It acts against the current relative slide and transfers surface motion.

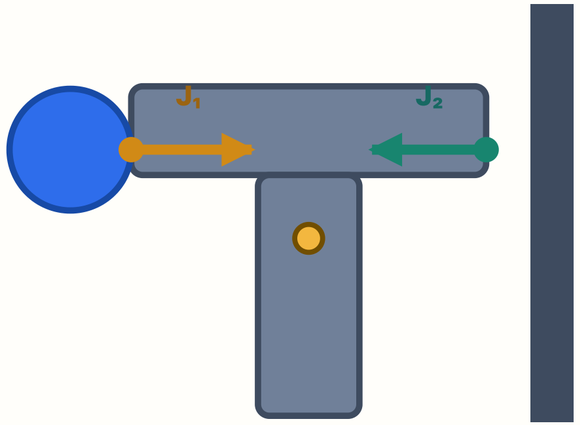
$$|j_t| \leq \mu j_n$$

Sticking: enough to stop sliding now. **Sliding:** reaches the limit and motion remains.

> Friction changes what motion the pusher transmits along the T surface—and therefore changes the future trajectory.

Two Contacts Act on the Same T-Block

One T, two simultaneous contacts



both contacts read and update the same v_t, ω_t

Why one independent answer per contact fails

J_1 changes the T's v and ω

the wall now sees a different relative contact velocity

J_2 changes the T's v and ω again

the pusher contact now sees a different relative velocity

the two answers are coupled through the T

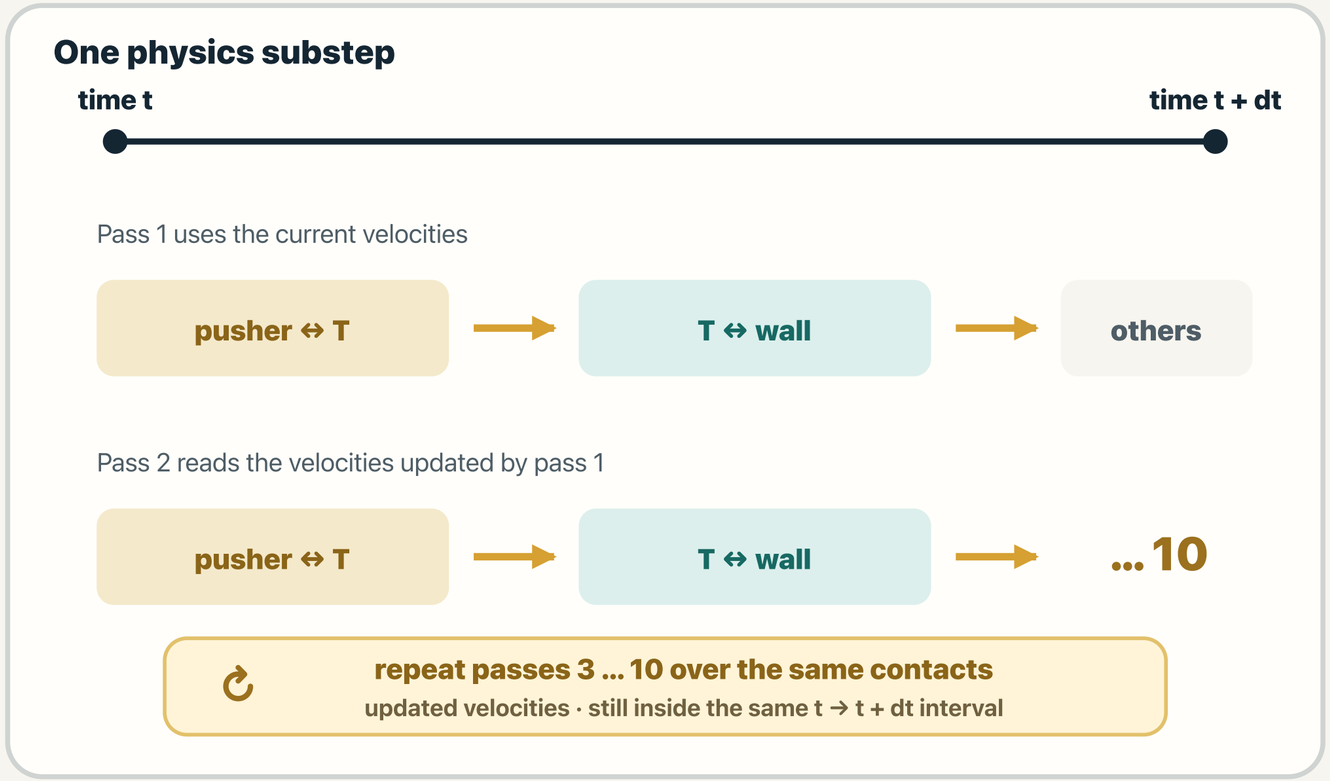
> Contacts that share a rigid body also share its velocity and angular velocity; correcting one contact changes the others.

How Pymunk Coordinates Several Contacts

```
space.iterations = 10
space.step(dt)
```

One outer call advances the simulated clock by dt.

The ten internal passes do not add physical time.



Physics substep
Calls `space.step(dt)` and advances time by dt.

Solver iteration
Refines contact and joint impulses within that same step.

Does Our Prediction Match Pymunk?

```
# Freeze the same upper-contact case.
trace = show_contact(case="upper")
# Read the quantities in causal order.
print(trace.geometry)
print(trace.relative_motion)
print(trace.impulse)
print(trace.state_change)
```

Before running: predict three signs—closing or separating, impulse zero or positive, clockwise or counter-clockwise.

This is a teaching trace around Pymunk outputs, not Pymunk's internal source code.

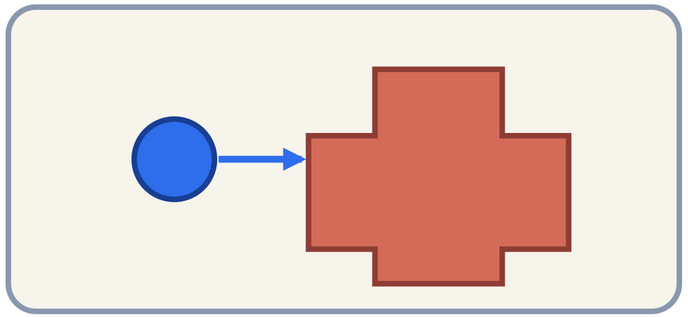
Open the course Colab · show_contact ↗

```
STATIC TRACE · case = "upper"

geometry
contact = (384.0, 195.0)
normal = (+1.000, +0.000)

relative motion
v_normal = -0.82 m/s  # closing

contact impulse on T
J = (+0.91, +0.06) N·s

state change of T
Δv = (+0.91, +0.06) m/s
Δω = -0.37 rad/s  # clockwise
```

geometry locates contact | v_n confirms closing | J prevents overlap | Δv, Δω update state

PushT Gives Us a 2D View of Robot Pushing

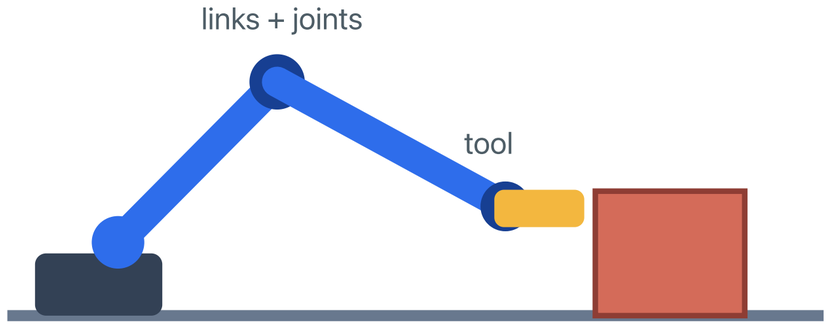
PushT · top-down plane



pusher = robot tool T-block = object

restore 3D structure

Robot pushing · table in 3D

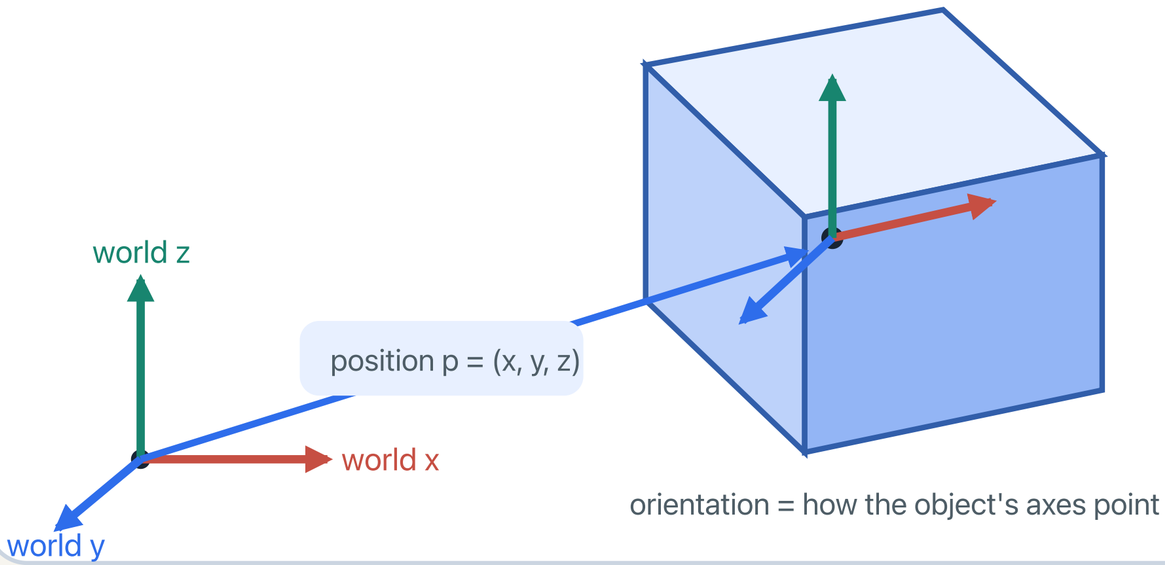


Still present: action → tool motion → contact → object response

Added in 3D: height · full orientation · links · joints · actuators

A 3D Object Has a Position and an Orientation

Where is the object, and which way does it face?



3D configuration

position

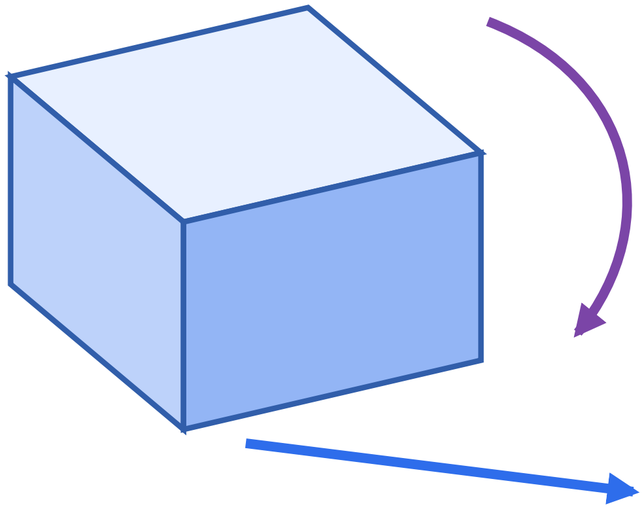
p = (x, y, z)

orientation

stored as a rotation matrix R
or a quaternion

representation names only — no derivation today

Motion in 3D Has Linear and Angular Velocity



Linear velocity

$v = (v_x, v_y, v\_z)$

direction and speed of translation

Angular velocity

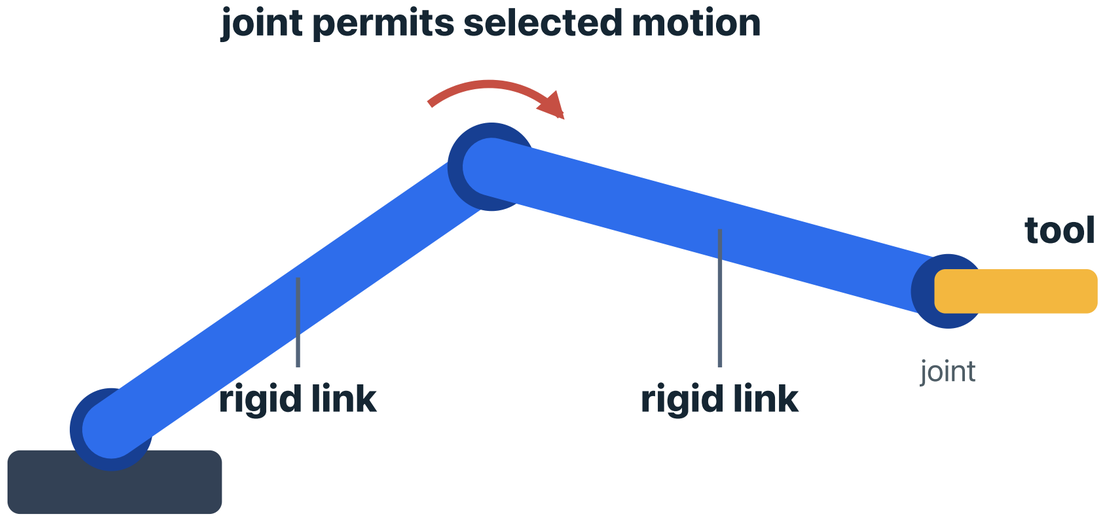
$\omega = (\omega_x, \omega_y, \omega\_z)$

axis of rotation + rotation rate

how orientation is changing now

A free rigid body's state records configuration (p, R) and motion (v, ω).

Joints Connect Rigid Links into a Robot



Link

one rigid body segment

Joint

a rule for relative motion

for example: rotate or slide

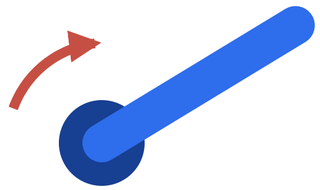
Articulated robot

links connected by joints

the tool moves with the chain

Joint Coordinates Record the Robot Configuration

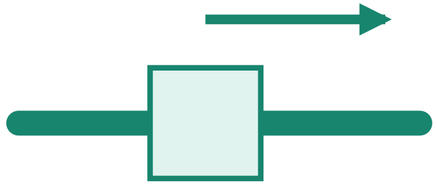
Revolute joint



q_i = joint angle

measured in radians or degrees

Prismatic joint



q_i = sliding distance

measured in meters

Collect every joint

$q = (q_1, \ldots, q_n)$

current joint configuration

$\dot{q} = (\dot{q}_1, \ldots, \dot{q}_n)$

how fast each value changes

PushT already used the same idea

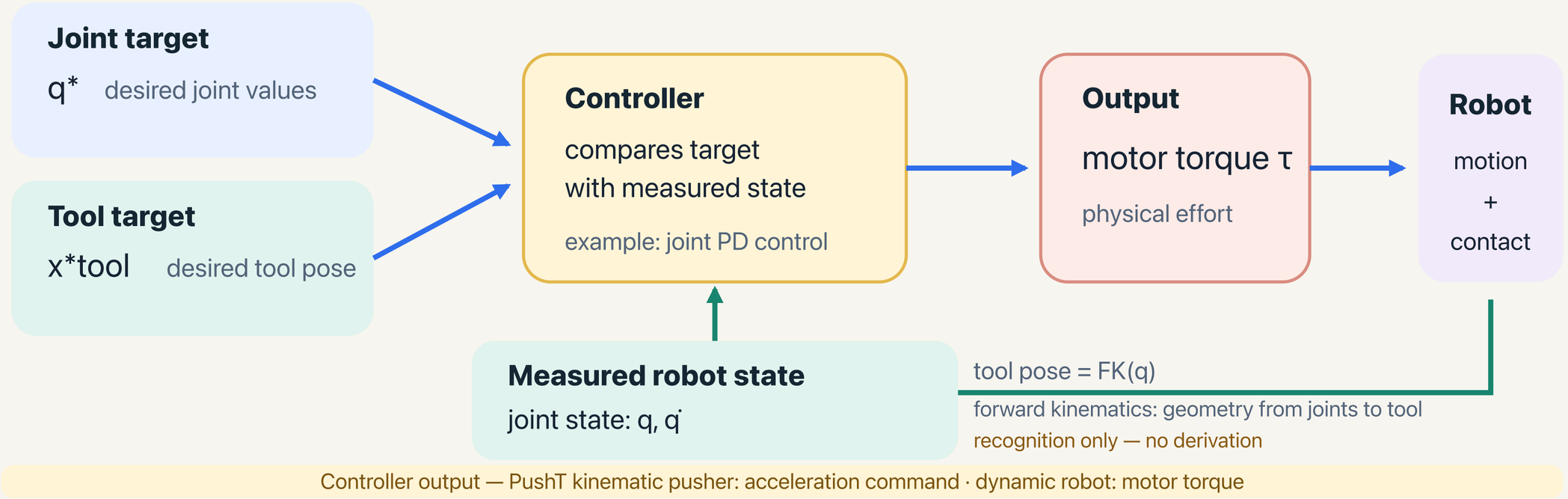
T-block configuration: (x, y, θ)

motion: (v_x, v_Y, ω)

θ is already an angle

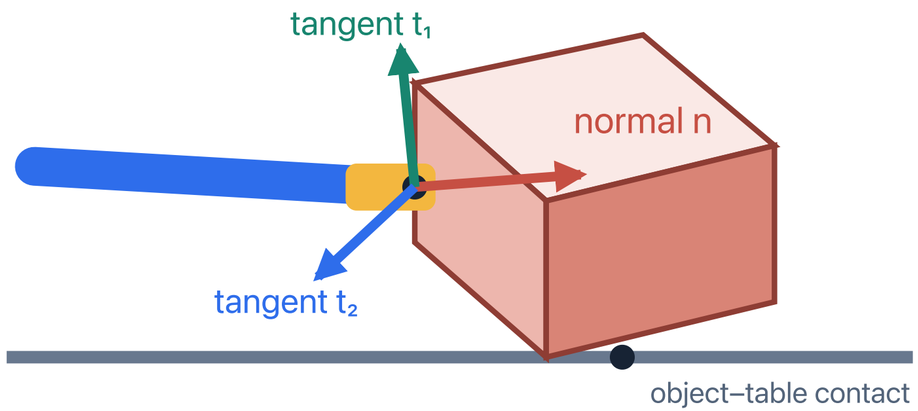
A Controller Turns a Target into Motor Torque

The action interface chooses what kind of target the policy specifies.



Controller output — PushT kinematic pusher: acceleration command · dynamic robot: motor torque

Robot Simulation Needs a 3D Articulated Engine



What changes from 2D to 3D?

- one contact normal remains
- sliding has two independent directions
  t_1 and t_2 span the tangent plane
- tool, object, table, links, and joints interact
- orientation and angular velocity are fully 3D

same simulation logic; a richer state and contact problem

Pymunk
2D rigid bodies · a good fit for PushT

MuJoCo
articulated 3D robot simulation

Genesis
another articulated 3D engine

Policy Chooses Actions, Simulator Generates Consequences

POLICY

What should I do now?

$$a_t \sim \pi_\varphi(\cdot \mid o_t)$$

The policy reads the current observation and selects an action.

observation o → action a

SIMULATOR

What happens if I do it?

$$s_{t+1} = F_{\text{sim}}(s_t, a_t)$$

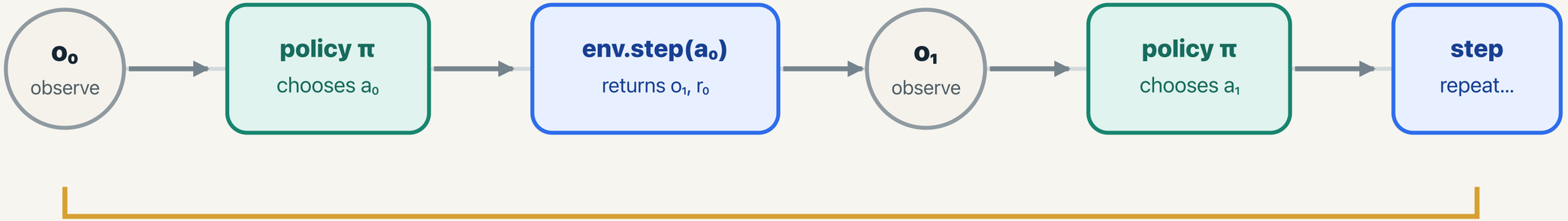
The simulator advances the physical state; the environment returns the resulting observation and reward.

state s + action a → next state

policy decides → simulator responds → policy observes

A Rollout Alternates Decisions and Transitions

policy chooses | environment advances



one rollout: continue until termination or a chosen horizon

$$\tau = (o_0, a_0, r_0, o_1, a_1, r_1, \ldots)$$

Simulation Makes Experience Repeatable and Scalable

1

Reset

Start a new rollout from a controlled initial condition or seed.

`reset(seed=7)` → o_0

2

Branch

Return to the same starting state and compare different actions.

same state → a / a′ → **two futures**

3

Repeat

Collect many episodes across starts, actions, and policy versions.

seed 1 **seed 2** **seed 3**

4

Parallelize

Advance many independent simulator instances at the same time.

env 1 **env 2** **... env N**

A simulator turns policy interaction into a controllable data-generation process: the same experiment can be repeated, varied, and scaled.

Policy Learning Uses Simulated Experience

compare_runs(policy, seeds) — static course code

```
def compare_runs(policy, seeds):
    returns = []
    for seed in seeds:
        obs, _ = env.reset(seed=seed)
        total = 0.0
        for _ in range(HORIZON):
            action = policy(obs)
            obs, reward, terminated, \
                truncated, _ = env.step(action)
            total += reward
            if terminated or truncated:
                break
        returns.append(total)
    return returns
```

Open the course Colab ↗

1 · Simulator supplies experience

Repeated transitions produce observations, actions, rewards, and next observations.

↓

2 · Reward supplies preference

Returns tell us which simulated outcomes better satisfy the task objective.

↓

3 · Learning updates the policy

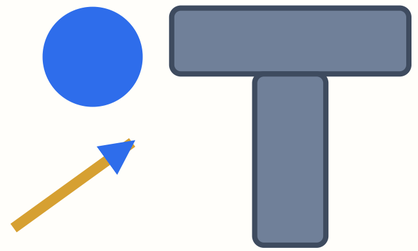
The algorithm changes policy parameters so future actions obtain better returns.

The simulator is already the transition model. Policy learning need not fit an additional learned world model.

If the target is reality, simulation mismatch matters → sim-to-real.

The Sim-to-Real Gap

Same visible start + same action

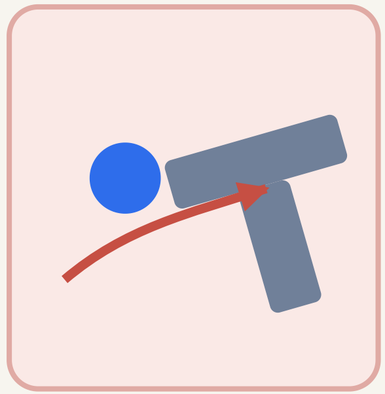


one pusher command

Simulator future

exactly follows the specified model

Physical future



follows the mechanisms and timing of reality

Where can the gap enter?

state sensing and estimation

mass, friction, gains, delay

missing physical mechanisms

timing and numerical contact

coverage and computation

A physics simulator is an oracle for its specified world. Sim-to-real error measures how that specified world differs from the physical target.

State and Observation in Simulation and Reality

INSIDE SIMULATION

The engine stores its exact internal state

pusher position + velocity	T pose + linear/angular velocity
contact cache + engine state	chosen physical parameters

A teaching environment may expose all of this directly, or render an image from it.

IN A PHYSICAL ROBOT

Sensors provide noisy, partial observations

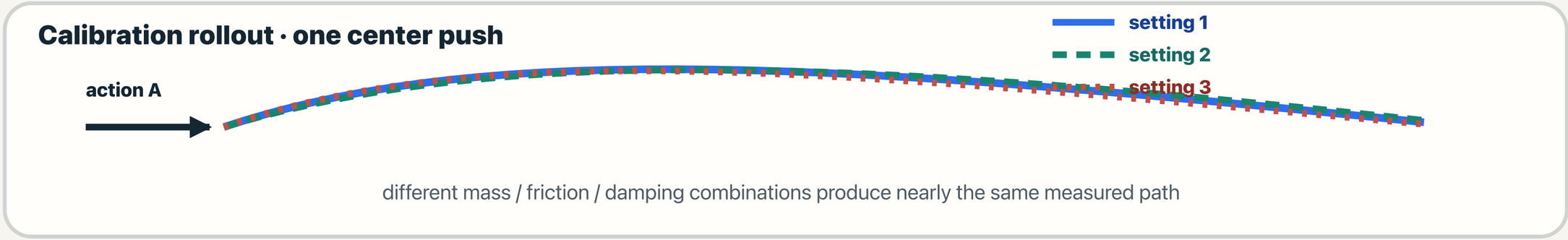
camera frames	joint encoders
force / tactile readings	timestamps and missing data

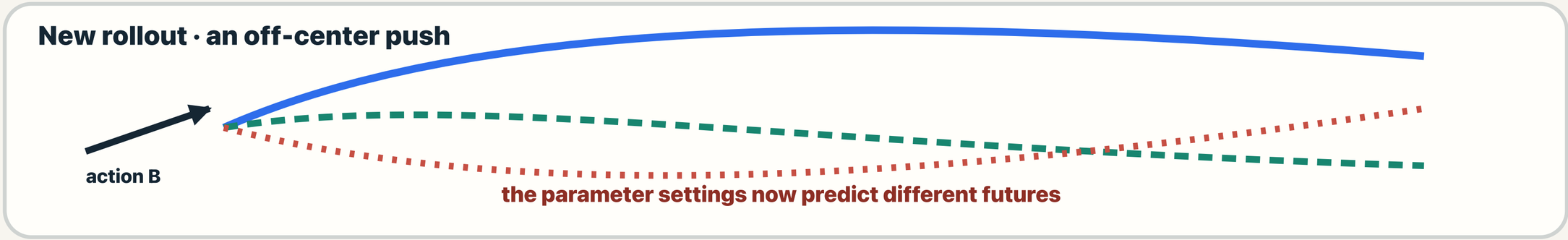
Velocity, contact state, friction, and hidden geometry often have to be inferred over time.

observation history → **estimated predictive / latent state** → **future observations and outcomes**

Lecture 2's predictive-state problem returns here: useful state is constructed from observation history when the target world's internal state is unavailable.

Calibrating Physical Parameters



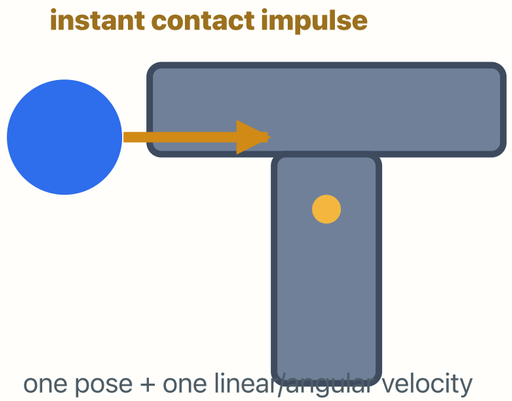


System identification:
choose parameters whose simulated measurements explain informative real experiments.

Matching one rollout does not uniquely identify the world.

Model Form Determines What the Simulator Can Represent

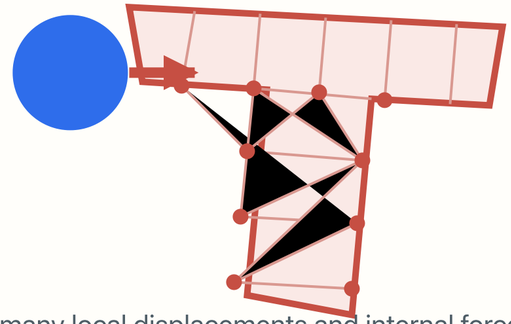
RIGID-BODY PUSHT MODEL



one pose + one linear/angular velocity

Efficient for planar contact and rotation. The T keeps its shape exactly.

COMPLIANT / DEFORMABLE MODEL



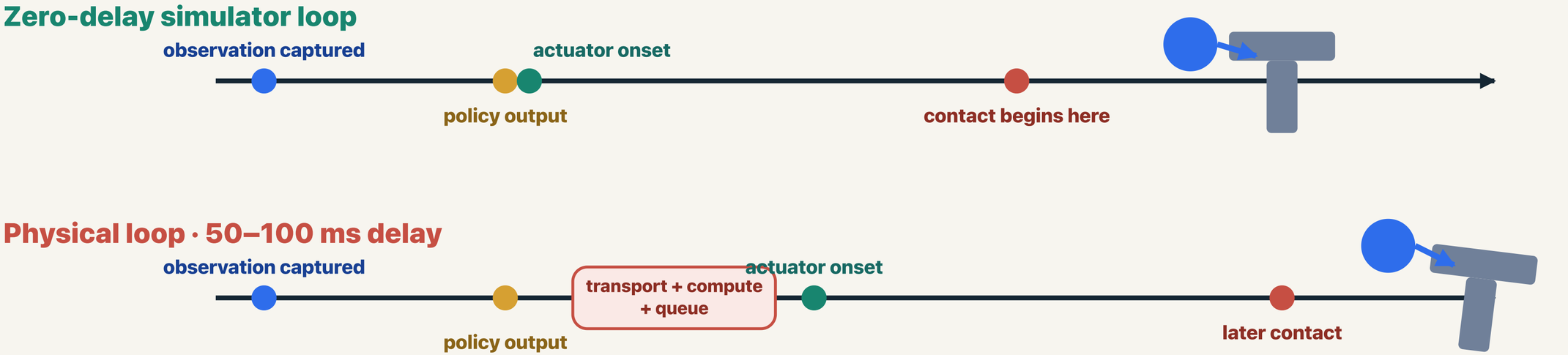
many local displacements and internal forces

Can represent compression and bending, with many more degrees of freedom.

many DOFs local points can move differently	**material law** stress depends on deformation	**internal forces** neighboring regions interact	**smaller dt** fast elastic motion needs resolution

Changing parameter values cannot create a mechanism missing from the model form. Richer physics increases both representational power and computational cost.

Delay Changes the PushT Interaction



physics dt
one numerical integration step

control interval
time between policy decisions

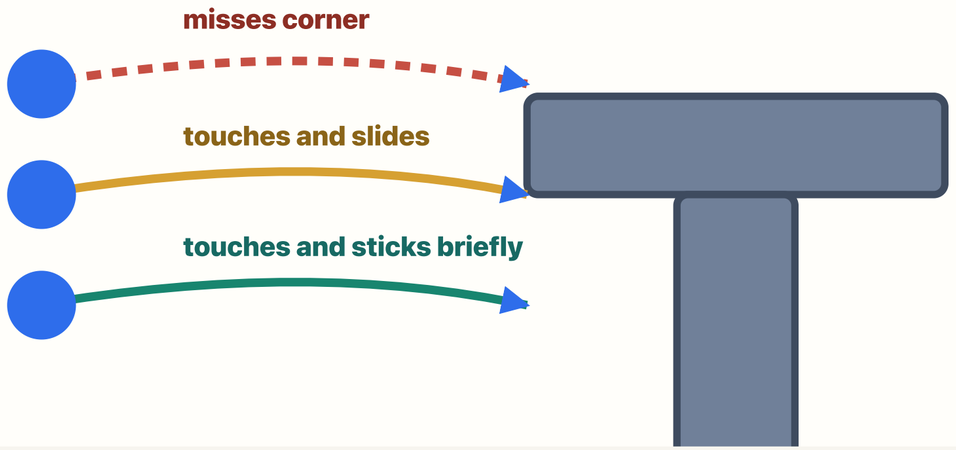
delay
capture-to-effect latency

wall-clock speed
how fast computation finishes

> Delay belongs to the interaction model when it changes which state receives the action.

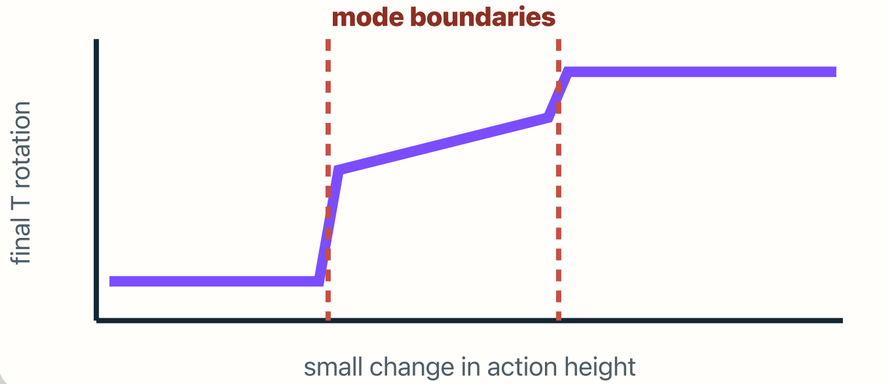
Contact Modes Create Sharp Changes

Vary one action component by a tiny amount



mode = which contacts exist + whether each is separating, sliding, or sticking

the active equations change when the mode changes



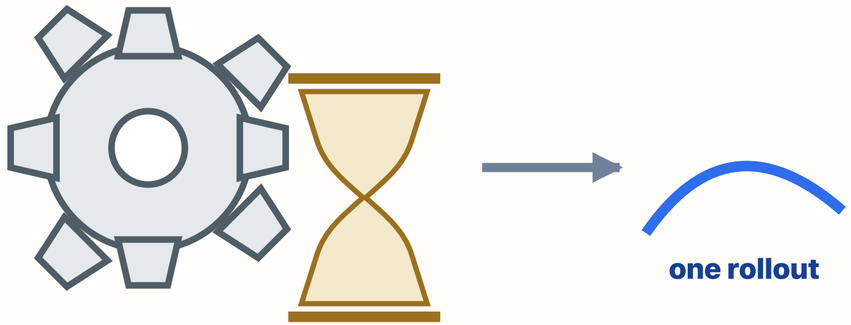
At a mode boundary, a gradient may jump, vanish, or become unstable.

Finite **dt**, collision tolerances, and a finite number of solver iterations can shift which mode the numerical simulator selects.

Coverage, Cost, and Simulator Speed

DETAILED PHYSICS SIMULATOR

One rollout may require substantial computation

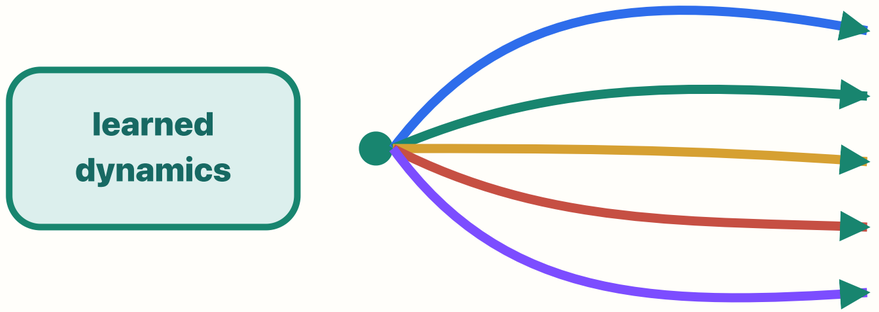


contact solve · many bodies · rendering · sensing

Higher fidelity can reduce throughput. Rare events and long horizons may still receive little coverage.

LEARNED SURROGATE WORLD MODEL

Many approximate futures can be imagined quickly



many action-conditioned imagined branches

Fast batched inference can expand search or training coverage across candidate futures.

The speed comes from approximation. A learned surrogate can omit rare contacts, extrapolate poorly, or confidently imagine a future that neither the physics simulator nor reality would produce.

Reducing the Sim-to-Real Gap

1 · SENSING AND STATE ESTIMATION

Match what the real system can observe

Model cameras, encoders, noise, occlusion, timestamps, and the estimator that converts observation history into control inputs.

2 · CALIBRATION AND VARIATION

Estimate parameters and train across uncertainty

Use system identification for likely values; use domain randomization when the target varies or calibration remains uncertain.

3 · RICHER PHYSICAL MODEL

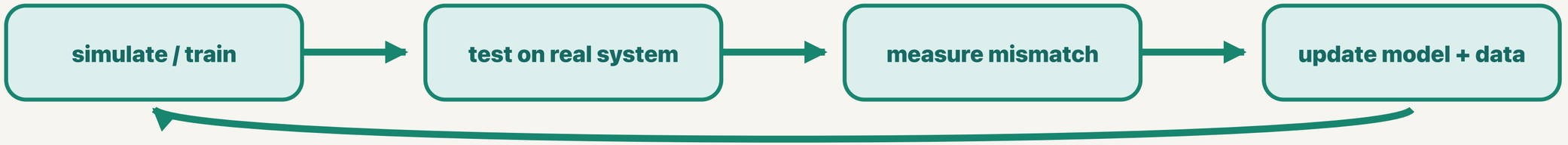
Add mechanisms that matter for the task

Upgrade geometry, compliance, actuator dynamics, contact sensing, or timing when validation shows that the simpler model misses them.

4 · REAL DATA AND HYBRID MODELS

Keep trusted structure and learn the residual

Mix simulated and real rollouts; learn perception, uncertain dynamics, or correction terms around a physics-based backbone.



Planners and policies can exploit errors in both physics simulators and learned world models. Hold out real tests, search for failure cases, and keep the model-improvement loop open.

COURSE SYNTHESIS

ONE EXECUTABLE CHAIN

The Environment Loop Connects State, Control, Physics, Contact, and Learning

1
(z_t, a_t)
current simulator state and action

→

2
$p^\star = p + 100a$
relative pusher target

→

3
PD + dt
controlled velocity update

→

4
`space.step(dt)`
integrate and detect contact

→

5
$J \to (\Delta v, \Delta \omega)$
contact changes T motion

→

6
z_{t+1}
next physical state

→

7
Gymnasium return
observation, reward, episode signals

SIMULATION

Executable world model

Specified physics and a numerical solver answer transition queries.

POLICY LEARNING

Rollouts create experience

If the simulator is the target world, a policy can learn directly from it.

WHY WORLD MODELS?

The target world exceeds the simulator

Use target data, learned components, and reality checks where specification fails.

> Understand the simulator deeply enough to know both what it can generate—and what it cannot.

Assignment 1: TBD · releases Monday, Sep 7