Curso Isaac Sim + Isaac Lab

Isaac Sim and Isaac Lab

https://developer.nvidia.com/isaac/sim
https://docs.omniverse.nvidia.com/isaacsim/latest/index.html
https://developer.nvidia.com/physx-sdk
https://docs.omniverse.nvidia.com/isaacsim/latest/isaac_lab_tutorials/index.html

Ligar o Issaac Sim e VSCode

1ª Cena: inclusão de objetos

Para utilizar o Isaac Sim, é necessário criar os objetos ANTES de começar a simulação.

Referência do tutorial: link

-> Vamos estudar a API para cada tipo de objeto
-> Entender outras opções de APIs que estão disponíveis

Tutorial

Etapa de geração

Etapas para criar um objeto:
1. Criar a configuração do objeto
2. Gerar ele no simulador

How to use spawners

Gerar o chão

```
import omni.isaac.lab.sim as sim_utils

# Ground-plane
cfg = sim_utils.GroundPlaneCfg()

# Spawn the object:
prim_path = "/World/defaultGroundPlane"
cfg.func(prim_path, cfg)
# OR
sim_utils.spawn_from_usd(prim_path, cfg)
```

API para o GroundPlaneCfg

Gerar iluminação

```
# Lights
cfg = sim_utils.DomeLightCfg(intensity=2000.0, color=(0.8, 0.8, 0.8))
```

```
cfg.func("/World/Light", cfg)
```

API para o DomeLightCfg

Gerar eixos de referência

Para ter vários "ecossistemas" de robôs identicamente configurados, é prático criar eixos de referências para cada um desses ecossistemas. Assim, a distância relativa e outras posições dentro desse ecossistemas permanecem parecidas.

```
import omni.isaac.core.utils.prims as prim_utils
# Create separate groups called "Origin1", "Origin2", "Origin3"
# Each group will have a robot in it
origins = [[0.25, 0.25, 0.],[-.25, 0.25, 0.], [0.25, -.25, 0.], [-.25, -.25,
0.]]
prim_utils.create_prim(f"/World/Origin1", "Xform", translation=[0.25, 0.25,
0.])
prim_utils.create_prim(f"/World/Origin2", "Xform", translation=[-.25, 0.25,
0.])
prim_utils.create_prim(f"/World/Origin3", "Xform", translation=[0.25, -.25,
0.])
prim_utils.create_prim(f"/World/Origin4", "Xform", translation=[-.25, -.25,
0.])
```

API para create_prim

Gerar um objeto rígido

Vamos gerar um Objeto Rígido, que vai ser um Cone, para cada uma das referências

```
from omni.isaac.lab.assets import RigidObject, RigidObjectCfg

# Rigid Object
cone_cfg = RigidObjectCfg(
        prim_path="/World/Origin.*/Cone",
        spawn=sim_utils.ConeCfg(
                radius=0.1,
                height=0.2,
                rigid_props=sim_utils.RigidBodyPropertiesCfg(),
                mass_props=sim_utils.MassPropertiesCfg(mass=1.0),
                collision_props=sim_utils.CollisionPropertiesCfg(),
```

```
                visual_material=sim_utils.PreviewSurfaceCfg(diffuse_color=
(0.0, 1.0, 0.0), metallic=0.2),
        ),
        init_state=RigidObjectCfg.InitialStateCfg(),
)
cone_object = RigidObject(cfg=cone_cfg)
```

API para ConeCfg
API para RigidObjectCfg
API para RigidObject

- Visualizar o atributo `data`

> For a rigid body, there are two frames of reference that are used:
> *Actor frame: The frame of reference of the rigid body prim. This typically corresponds to the Xform prim with the rigid body schema.*
> Center of mass frame: The frame of reference of the center of mass of the rigid body.

Etapa de Simulação

Setup a ser feito após inicializar a simulação

```
# 1. Get the default state for each object
root_state = cone_object.data.default_root_state.clone()

# 2. Change the position of cylinders for matching each origin
root_state[:, :3] += torch.tensor(origins)

# 3. Write root state to simulation
cone_object.write_root_state_to_sim(root_state)

# 4. Reset buffers
cone_object.reset()
```

Para a simulação caminhar, pode executar:

```
# update buffers
cone_object.update(sim_dt)
```

Funções úteis para a simulação

```
math_utils.sample_cylinder(
                radius=0.1, h_range=(0.25, 0.5),
                size=cone_object.num_instances,
                device=cone_object.device
            )
```

API para o sample_cylinder

Desafio 1

Criar um código que gera 4 cones (cada um em uma origem) e os movimenta aleatoriamente.

Resete a simulação a cada 250 passos e a inicie novamente!

Boilerplate inicial:

```
import argparse
from omni.isaac.lab.app import AppLauncher

# add argparse arguments
parser = argparse.ArgumentParser(description="Tutorial on spawning and
interacting with a rigid object.")

# append AppLauncher cli args
AppLauncher.add_app_launcher_args(parser)

# parse the arguments
args_cli = parser.parse_args()

# launch omniverse app
app_launcher = AppLauncher(args_cli)
simulation_app = app_launcher.app

import torch
import omni.isaac.core.utils.prims as prim_utils
import omni.isaac.lab.sim as sim_utils
import omni.isaac.lab.utils.math as math_utils
from omni.isaac.lab.assets import RigidObject, RigidObjectCfg
from omni.isaac.lab.sim import SimulationContext


def setup_scene():
        """ Include here all code for setup the objects """
        # TODO HERE


def run_simulation(sim: sim_utils.SimulationContext, entities: RigidObject,
origins: torch.Tensor):
        """ Include here all code for running the simulation """

        sim_dt = sim.get_physics_dt()
    sim_time = 0.0
    count = 0
```

```
        while simulation_app.is_running():
                # TODO HERE

        # perform step
        sim.step()
                sim_time += sim_dt
                count += 1

def main():
    """Main function."""
    # Load kit helper
    sim_cfg = sim_utils.SimulationCfg()
    sim = SimulationContext(sim_cfg)

    # Set main camera
    sim.set_camera_view(eye=[1.5, 0.0, 1.0], target=[0.0, 0.0, 0.0])

    # Design scene
    scene_entities, scene_origins = design_scene()
    scene_origins = torch.tensor(scene_origins, device=sim.device)

    # Play the simulator
    sim.reset()

    # Now we are ready!
    print("[INFO]: Setup complete...")

    # Run the simulator
    run_simulator(sim, scene_entities, scene_origins)

if __name__ == "__main__":

    # run the main function
    main()

    # close sim app
    simulation_app.close()
```

2ª Cena: objetos articulados

Introdução

Objetos articulados

Objeto articulado é qualquer objeto que possua uma junta (i.e. robôs).

Por exemplo, para importar o *cartpole*:

```
from omni.isaac.lab_assets import CARTPOLE_CFG  # isort:skip

cartpole_cfg = CARTPOLE_CFG.replace(prim_path="/World/Origin.*/Robot")
cartpole = Articulation(cfg=cartpole_cfg)
```

Podemos controlar as juntas do robô da seguinte maneira:

Para **resetar a simulação** das juntas, inclui-se adicionalmente:

```
# set joint positions with some noise
joint_pos, joint_vel = robot.data.default_joint_pos.clone(),
robot.data.default_joint_vel.clone()
joint_pos += torch.rand_like(joint_pos) * 0.1
robot.write_joint_state_to_sim(joint_pos, joint_vel)
```

Para **dar um passo** na simulação, pode-se fazer:

```
# Apply random action
# 1. generate random joint efforts
efforts = torch.randn_like(robot.data.joint_pos) * 5.0

# 2. apply action to the robot
robot.set_joint_effort_target(efforts)

# 3. write data to sim
robot.write_data_to_sim()
robot.update(sim_dt)
```

As etapas, em outras palavras, são:

1. Setar a velocidade de cada junta
2. Enviar os dados para o simulador

> Notar que o estado do robô é sempre somado das origens, caso contrário todos os robôs iriam para a posição (0, 0, 0) do simulador.

(TODO, incluir no desafio as etapas de #1 resetar para o origin e #3 resetar buffer)

Desafio 2

Criar na cena, ao invés de cones, o cartpole e colocar força aleatória nas juntas para que eles se movam.

3ª Cena: usar cenas interativas

Para facilitar a gestão das cenas, é possível utilizar arquivos de configuração

Há uma diferença importante entre as configurações para o chão/fonte de luz e as configurações para o carrinho de pêndulo (cartpole): o solo e a fonte de luz são *prims* não interativos, enquanto o carrinho de pêndulo é um *prim* interativo.

Então, cada um utiliza uma classe diferente:

1. As configurações para o plano de solo e a fonte de luz são especificadas usando uma instância da classe `assets.AssetBaseCfg`,
2. O carrinho de pêndulo é configurado usando uma instância da classe `assets.ArticulationCfg`

Qualquer coisa que não seja um *prim* interativo (ou seja, que não seja nem um ativo nem um sensor) não é *gerenciada* pela cena durante os passos da simulação.

Como gerar vários objetos, um para cada origem/environment desejado?

- Plano de solo: `/World/defaultGroundPlane`
- Fonte de luz: `/World/Light`
- Carrinho de pêndulo: `{ENV_REGEX_NS}/Robot`

Como aprendemos anteriormente, o Omniverse cria um grafo de *prims* no estágio USD.

Os caminhos *prim* são usados para especificar a localização do *prim* no grafo.

O solo e a fonte de luz são especificados usando caminhos absolutos, enquanto o carrinho de pêndulo é especificado usando um caminho relativo. O caminho relativo é especificado usando a variável `ENV_REGEX_NS`, que é uma variável especial substituída pelo nome do ambiente durante a criação da cena. Qualquer entidade que tenha a variável `ENV_REGEX_NS` em seu caminho *prim* será clonada para cada ambiente. Esse caminho é substituído pelo objeto da cena com `/World/envs/env_{i}`, onde `i` é o índice do ambiente.

Tutorial

Classes de configuração

```
@configclass
class CartpoleSceneCfg(InteractiveSceneCfg):
    """Configuration for a cart-pole scene."""
```

```
    # ground plane
    ground = AssetBaseCfg(
            prim_path="/World/defaultGroundPlane",
            spawn=sim_utils.GroundPlaneCfg(size=(100.0, 100.0))
            )

    # lights
    dome_light = AssetBaseCfg(
        prim_path="/World/Light",
        spawn=sim_utils.DomeLightCfg(intensity=3000.0, color=(0.75, 0.75,
0.75))
    )
    distant_light = AssetBaseCfg(
        prim_path="/World/DistantLight",
        spawn=sim_utils.DistantLightCfg(color=(0.9, 0.9, 0.9),
intensity=2500.0),
        init_state=AssetBaseCfg.InitialStateCfg(rot=(0.738, 0.477, 0.477,
0.0)),
    )

        # articulation
    cartpole: ArticulationCfg = CARTPOLE_CFG.replace(prim_path="
{ENV_REGEX_NS}/Robot")
```

Instanciar a Cena

Ao invés de ter uma função como a `setup_scene`, agora a cena como um todo pode ser gerada à partir de sua instanciação:

```
# Design scene
scene_cfg = CartpoleSceneCfg(num_envs=args_cli.num_envs, env_spacing=2.0)
scene = InteractiveScene(scene_cfg)
```

Acessar objetos dentro da cena

Para acessar os objetos que foram configurados, pode-se utilizar a cena como um dicionário:

```
robot = scene["cartpole"]
```

Etapas de simulação

Ao invés de ter que fazer o `write_to_sim` e `update` de cada objeto rígido/articulado, pode-se fazer as chamadas apenas uma vez para toda a cena!

```
# Write data to the simulator
scene.write_data_to_sim()

# Perform step
sim.step()

# Update buffers
scene.update(sim_dt)
```

Desafio 3: cartpole com cena Interativa

Refaça o desafio 2 utilizando a cena interativa.

Revisão de RL

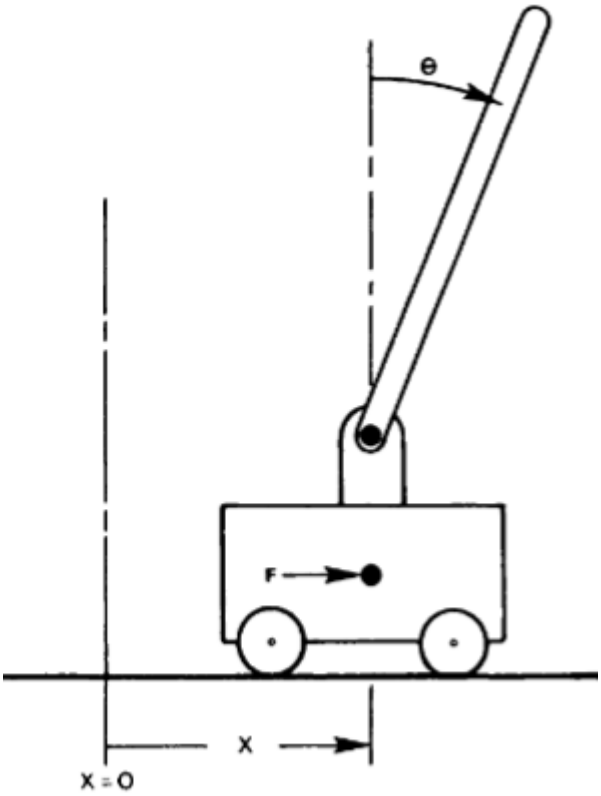
https://www.davidsilver.uk/wp-content/uploads/2020/03/intro_RL.pdf

Slides 16-19

Introdução do cartpole

https://gymnasium.farama.org/environments/classic_control/cart_pole/

Paper original: Neuronlike adaptive elements that can solve difficult learning control problems



Descrição do desafio do cartpole:

Action Space: the action is a array with shape `(1,)` which can take values `{0, 1}` indicating the direction of the fixed force the cart is pushed with.

- 0: Push cart to the left
- 1: Push cart to the right

Observações:

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -0.418 rad (-24°)	~ 0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

Rewards: goal is to keep the pole upright for as long as possible, hence a reward of `+1` for every step taken, including the termination step, is allotted. The threshold for rewards is 500.

Starting State: all observations are assigned a uniformly random value in `(-0.05, 0.05)`

Episode End: the episode ends if any one of the following occurs:

1. Termination: Pole Angle is greater than ±12°
2. Termination: Cart Position is greater than ±2.4
3. Truncation: Episode length is greater than 500

4º Criar um ambiente de Simulação

Referência tutorial

Atuadores

Opções de atuadores:

Função	Descrição
`JointActionCfg`	Configuration for the base joint action term.
`JointPositionActionCfg`	Configuration for the joint position action term.
`RelativeJointPositionActionCfg`	Configuration for the relative joint position action term.
`JointVelocityActionCfg`	Configuration for the joint velocity action term.
`JointEffortActionCfg`	Configuration for the joint effort action term.
`JointPositionToLimitsActionCfg`	Configuration for the bounded joint position action term.
`EMAJointPositionToLimitsActionCfg`	Configuration for the exponential moving average (EMA) joint position action term.
`BinaryJointActionCfg`	Configuration for the base binary joint action term.
`BinaryJointPositionActionCfg`	Configuration for the binary joint position action term.
`BinaryJointVelocityActionCfg`	Configuration for the binary joint velocity action term.
`NonHolonomicActionCfg`	Configuration for the non-holonomic action term with dummy joints at the base.
`DifferentialInverseKinematicsActionCfg`	Configuration for inverse differential kinematics action term.

No caso do cartpole, criar uma junta de força.

```
import omni.isaac.
lab_tasks.manager_based.classic.cartpole.mdp as mdp

## Todo -> posso fazer o importe da função de joint effort padrão?
```

```
@configclass
class ActionsCfg:
    """Action specifications for the environment."""
    joint_efforts = mdp.JointEffortActionCfg(asset_name="robot",
joint_names=["slider_to_cart"], scale=100.0)
```

Observações

Função	Descrição
`base_pos_z` (env[, asset_cfg])	Root height in the simulation world frame.
`base_lin_vel` (env[, asset_cfg])	Root linear velocity in the asset's root frame.
`base_ang_vel` (env[, asset_cfg])	Root angular velocity in the asset's root frame.
`projected_gravity` (env[, asset_cfg])	Gravity projection on the asset's root frame.
`root_pos_w` (env[, asset_cfg])	Asset root position in the environment frame.
`root_quat_w` (env[, make_quat_unique, asset_cfg])	Asset root orientation (w, x, y, z) in the environment frame.
`root_lin_vel_w` (env[, asset_cfg])	Asset root linear velocity in the environment frame.
`root_ang_vel_w` (env[, asset_cfg])	Asset root angular velocity in the environment frame.
`joint_pos` (env[, asset_cfg])	The joint positions of the asset.
`joint_pos_rel` (env[, asset_cfg])	The joint positions of the asset w.r.t.
`joint_pos_limit_normalized` (env[, asset_cfg])	The joint positions of the asset normalized with the asset's joint limits.
`joint_vel` (env[, asset_cfg])	The joint velocities of the asset.
`joint_vel_rel` (env[, asset_cfg])	The joint velocities of the asset w.r.t.
`height_scan` (env, sensor_cfg[, offset])	Height scan from the given sensor w.r.t.
`body_incoming_wrench` (env, asset_cfg)	Incoming spatial wrench on bodies of an articulation in the simulation world frame.
`last_action` (env[, action_name])	The last input action to the environment.

Função	Descrição
`generated_commands` (env, command_name)	The generated command from command term in the command manager with the given name.

```python
@configclass
class ObservationsCfg:
    """Observation specifications for the environment."""

        # PQ a subclasse?
    @configclass
    class PolicyCfg(ObsGroup):
        """Observations for policy group."""
        # observation terms (order preserved)
        joint_pos_rel = ObsTerm(func=mdp.joint_pos_rel)
        joint_vel_rel = ObsTerm(func=mdp.joint_vel_rel)

        def __post_init__(self) -> None:
            self.enable_corruption = False
            self.concatenate_terms = True

    # observation groups
    policy: PolicyCfg = PolicyCfg()
```

Eventos e randomizações

Função	Descrição
`randomize_rigid_body_material`	Randomize the physics materials on all geometries of the asset.
`randomize_rigid_body_mass`	Randomize the mass of the bodies by adding, scaling, or setting random values.
`randomize_physics_scene_gravity`	Randomize gravity by adding, scaling, or setting random values.
`randomize_actuator_gains`	Randomize the actuator gains in an articulation by adding, scaling, or setting random values.
`randomize_joint_parameters`	Randomize the joint parameters of an articulation by adding, scaling, or setting random values.
`randomize_fixed_tendon_parameters`	Randomize the fixed tendon parameters of an articulation by adding, scaling, or

Função	Descrição
	setting random values.
`apply_external_force_torque`	Randomize the external forces and torques applied to the bodies.
`push_by_setting_velocity`	Push the asset by setting the root velocity to a random value within the given ranges.
`reset_root_state_uniform`	Reset the asset root state to a random position and velocity uniformly within the given ranges.
`reset_root_state_with_random_orientation`	Reset the asset root position and velocities sampled randomly within the given ranges and the asset root orientation sampled randomly from the SO(3).
`reset_root_state_from_terrain`	Reset the asset root state by sampling a random valid pose from the terrain.
`reset_joints_by_scale`	Reset the robot joints by scaling the default position and velocity by the given ranges.
`reset_joints_by_offset`	Reset the robot joints with offsets around the default position and velocity by the given ranges.
`reset_scene_to_default`	Reset the scene to the default state specified in the scene configuration.

```
@configclass
class EventCfg:
    """Configuration for events."""

        # Transformar em desafio
    # on startup
    add_pole_mass = EventTerm(
        func=mdp.randomize_rigid_body_mass,
        mode="startup",
        params={
            "asset_cfg": SceneEntityCfg("robot", body_names=["pole"]),
            "mass_distribution_params": (0.1, 0.5),
            "operation": "add",
        },
    )
```

```
    # on reset
    reset_cart_position = EventTerm(
        func=mdp.reset_joints_by_offset,
        mode="reset",
        params={
            "asset_cfg": SceneEntityCfg("robot", joint_names=
["slider_to_cart"]),
            "position_range": (-1.0, 1.0),
            "velocity_range": (-0.5, 0.5),
        },
    )

    reset_pole_position = EventTerm(
        func=mdp.reset_joints_by_offset,
        mode="reset",
        params={
            "asset_cfg": SceneEntityCfg("robot", joint_names=
["cart_to_pole"]),
            "position_range": (-0.25 * math.pi, 0.25 * math.pi),
            "velocity_range": (-0.25 * math.pi, 0.25 * math.pi),
        },
    )
```

Consolidar tudo

```
@configclass
class CartpoleEnvCfg(ManagerBasedEnvCfg):
    """Configuration for the cartpole environment."""
    # Scene settings
    scene = CartpoleSceneCfg(num_envs=1024, env_spacing=2.5)

        # Basic settings
    observations = ObservationsCfg()
    actions = ActionsCfg()
    events = EventCfg()

    def __post_init__(self):
        """Post initialization."""

        # viewer settings
        self.viewer.eye = [4.5, 0.0, 6.0]
        self.viewer.lookat = [0.0, 0.0, 2.0]

        # step settings
```

```
        self.decimation = 4  # env step every 4 sim steps: 200Hz / 4 = 50Hz

        # simulation settings
        self.sim.dt = 0.005  # sim step every 5ms: 200Hz
```

```
def main():
    """Main function."""

    # parse the arguments
    env_cfg = CartpoleEnvCfg()
    env_cfg.scene.num_envs = args_cli.num_envs

    # setup base environment
    env = ManagerBasedEnv(cfg=env_cfg)

    # simulate physics
    count = 0

    while simulation_app.is_running():
        with torch.inference_mode():

            # reset
            if count % 300 == 0:
                count = 0
                env.reset()
                print("-" * 80)
                print("[INFO]: Resetting environment...")

            # sample random actions
            joint_efforts = torch.randn_like(env.action_manager.action)

            # step the environment
            obs, _ = env.step(joint_efforts)

            # print current orientation of pole
            print("[Env 0]: Pole joint: ", obs["policy"][0][1].item())

            # update counter
            count += 1

    # close the environment
    env.close()
```

5º Criar ambiente para RL

```
@configclass
class CommandsCfg:
    """Command terms for the MDP."""
        # no commands for this MDP
    null = mdp.NullCommandCfg()
```

```
@configclass

class RewardsCfg:
    """Reward terms for the MDP."""
    # (1) Constant running reward
    alive = RewTerm(func=mdp.is_alive, weight=1.0)

    # (2) Failure penalty
    terminating = RewTerm(func=mdp.is_terminated, weight=-2.0)

    # (3) Primary task: keep pole upright
    pole_pos = RewTerm(
        func=mdp.joint_pos_target_l2,
        weight=-1.0,
        params={"asset_cfg": SceneEntityCfg("robot", joint_names=
["cart_to_pole"]), "target": 0.0},
    )

    # (4) Shaping tasks: lower cart velocity
    cart_vel = RewTerm(
        func=mdp.joint_vel_l1,
        weight=-0.01,
        params={"asset_cfg": SceneEntityCfg("robot", joint_names=
["slider_to_cart"])},
    )

    # (5) Shaping tasks: lower pole angular velocity
    pole_vel = RewTerm(
        func=mdp.joint_vel_l1,
        weight=-0.005,
        params={"asset_cfg": SceneEntityCfg("robot", joint_names=
["cart_to_pole"])},
    )
```

```
@configclass
class TerminationsCfg:
    """Termination terms for the MDP."""
    # (1) Time out
    time_out = DoneTerm(func=mdp.time_out, time_out=True)

    # (2) Cart out of bounds
    cart_out_of_bounds = DoneTerm(
        func=mdp.joint_pos_out_of_manual_limit,
        params={"asset_cfg": SceneEntityCfg("robot", joint_names=
["slider_to_cart"]), "bounds": (-3.0, 3.0)},
    )
```

```
@configclass
class CurriculumCfg:
    """Configuration for the curriculum."""
    pass
```

```
@configclass

class CartpoleEnvCfg(ManagerBasedRLEnvCfg):
    """Configuration for the locomotion velocity-tracking environment."""

    # Scene settings
    scene: CartpoleSceneCfg = CartpoleSceneCfg(num_envs=4096,
env_spacing=4.0)
    # Basic settings
    observations: ObservationsCfg = ObservationsCfg()
    actions: ActionsCfg = ActionsCfg()
    events: EventCfg = EventCfg()
    # MDP settings
    curriculum: CurriculumCfg = CurriculumCfg()
    rewards: RewardsCfg = RewardsCfg()
    terminations: TerminationsCfg = TerminationsCfg()
    # No command generator
    commands: CommandsCfg = CommandsCfg()

    # Post initialization
    def __post_init__(self) -> None:
        """Post initialization."""

        # general settings
        self.decimation = 2
```

```
        self.episode_length_s = 5

        # viewer settings
        self.viewer.eye = (8.0, 0.0, 5.0)
        # simulation settings
        self.sim.dt = 1 / 120
        self.sim.render_interval = self.decimation
```