Figure 1: Code World Model of the Ant-v4 environment generated by GIF-MCTS with Llama 3. For reference, the official implementation of the environment can be found at the official Gymnasium GitHub repository of the Farama Foundation (at `gymnasium.envs.mujoco.Ant_v4`).

```python
import numpy as np
class Environment:
    def __init__(self,
                 healthy_reward=1.0,
                 forward_reward_weight=1.0,
                 ctrl_cost_weight=0.5,
                 contact_cost_weight=5e-4,
                 contact_force_range=10.0,
                 healthy_z_range=(0.2, 1.0),
                 terminate_when_unhealthy=True,
                 reset_noise_scale=1e-2,
                 exclude_current_positions_from_observation=True):
        [...]
    def _reset_state(self):
        [...]
    def set_state(self, state):
        self.state = state

    def step(self, action):
        # implement dynamics model or physics engine to predict next state
        next_state = self._predict_next_state(self.state, action)
        observation = next_state
        reward = self._calculate_reward(action, next_state)
        done = self._check_episode_end(next_state)
        return observation, reward, done

    def _predict_next_state(self, state, action):
        # implement dynamics model or physics engine to predict next state
        # for example, using a simple integrator
        # assume velocities are the last 14 components of the state
        velocities = state[13:]
        new_velocities = velocities.copy()
        # update front left leg, front right leg, back left leg, and back right leg
        new_velocities[:4] += 0.01 * action[:4]
        # update corresponding body parts
        new_velocities[4:8] += 0.01 * action[4:]
        # Fix: slice new_velocities to match shape of state[:13]
        new_positions = state[:13] + 0.01 * new_velocities[:13]
        next_state = np.concatenate((new_positions, new_velocities))
        return next_state

    def _calculate_reward(self, action, next_state):
        # implement reward calculation logic here
        healthy_reward = self.healthy_reward
        forward_reward = self.forward_reward_weight * \
            (next_state[13] - self.state[13])  # x-coordinate velocity
        ctrl_cost = self.ctrl_cost_weight * np.sum(np.square(action))
        contact_cost = 0  # implement contact cost calculation logic here
        reward = healthy_reward + forward_reward - ctrl_cost - contact_cost
        return reward

    def _check_episode_end(self, state):
        # implement episode end check logic here
        done = not (self.healthy_z_range[0] <=state[0] <= self.healthy_z_range[1])
        return done
```