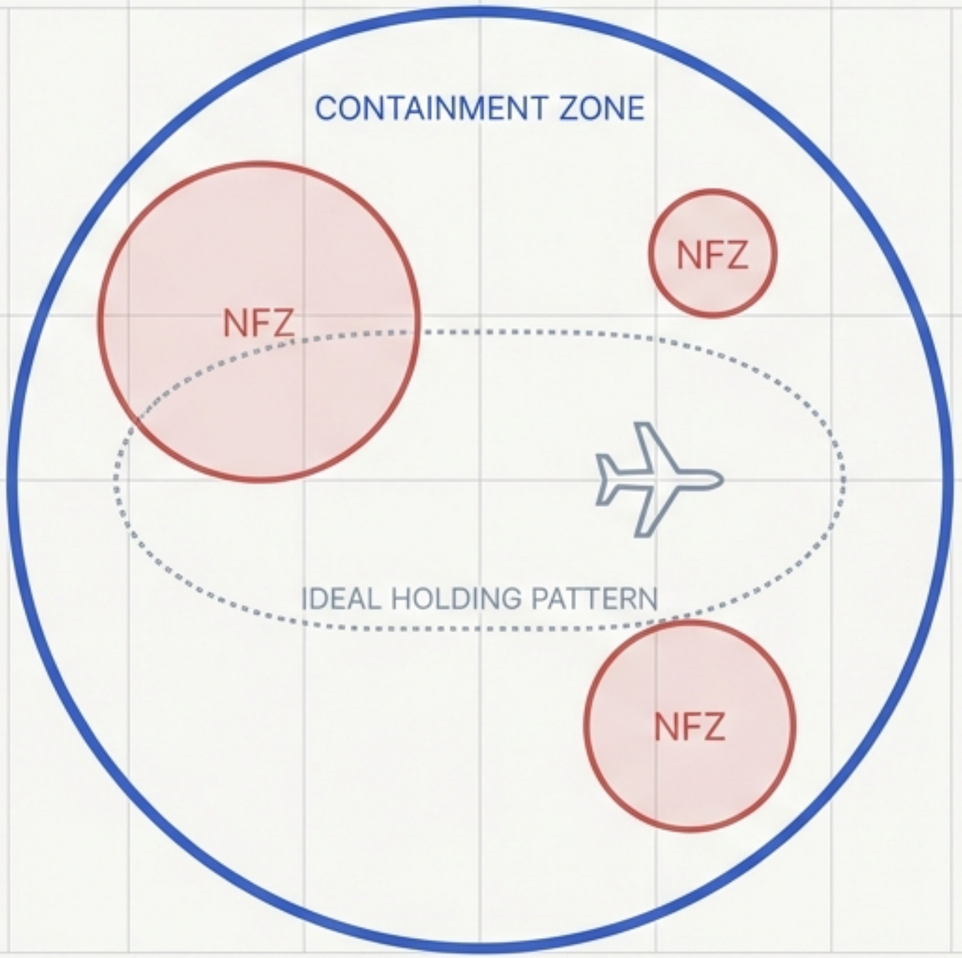
HoldingAgentLidarEnv: A Blueprint for Portable Reinforcement Learning

Designing simulator-agnostic environments with a clean, testable architecture.

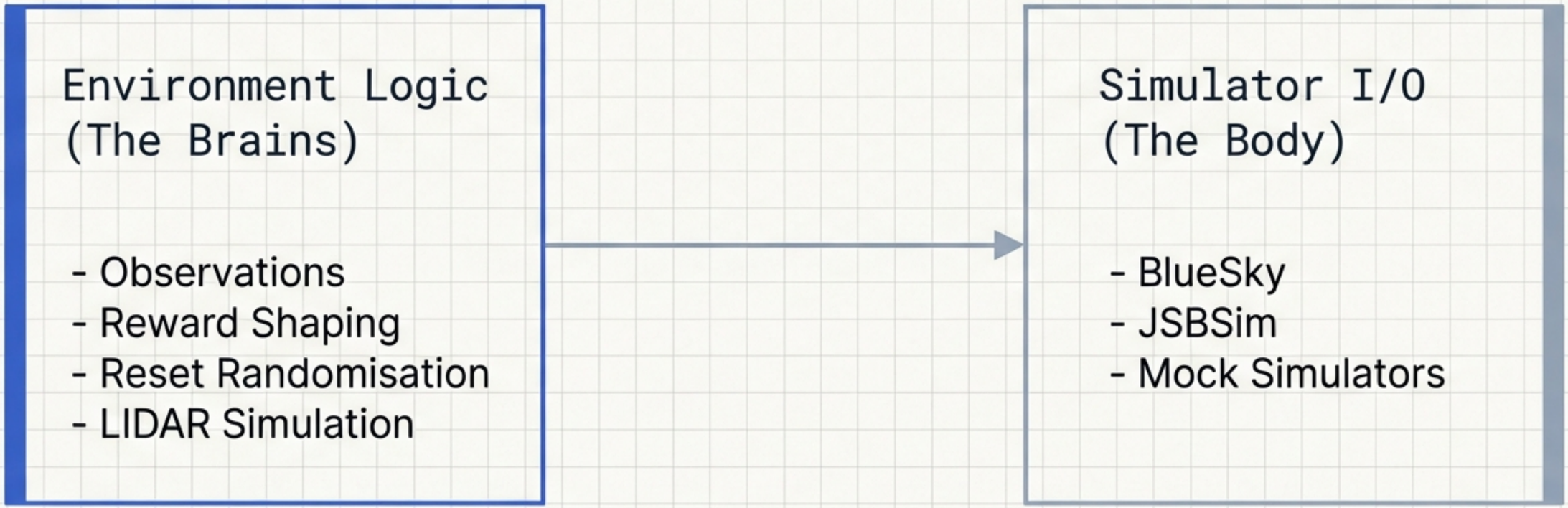
THE MISSION: TRAIN AN AGENT FOR A COMPLEX AEROSPACE TASK

To train an aircraft agent to hold a pattern inside a circular containment zone while avoiding circular No-Fly Zones (NFZs) within a robust, simulator-independent framework.



THE CORE PHILOSOPHY IS A CLEAN SEPARATION OF CONCERNS

The environment's core logic is completely independent of any specific simulator.
The design intentionally separates simulator I/O from the environment's logic.



THIS ARCHITECTURAL DISCIPLINE DELIVERS STRATEGIC ADVANTAGES

TESTABILITY

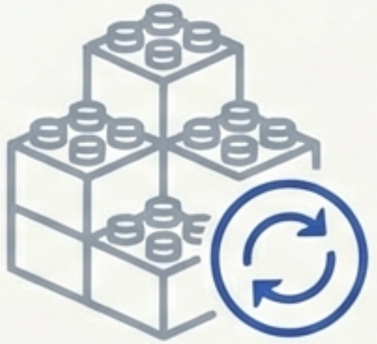
Mock the simulator for unit tests. Test environment logic without needing to run a full, heavy simulator.

PORTABILITY

Swap simulator backends with minimal code changes. Move from BlueSky to JSBSim or another compliant simulator easily.

MAINTAINABILITY

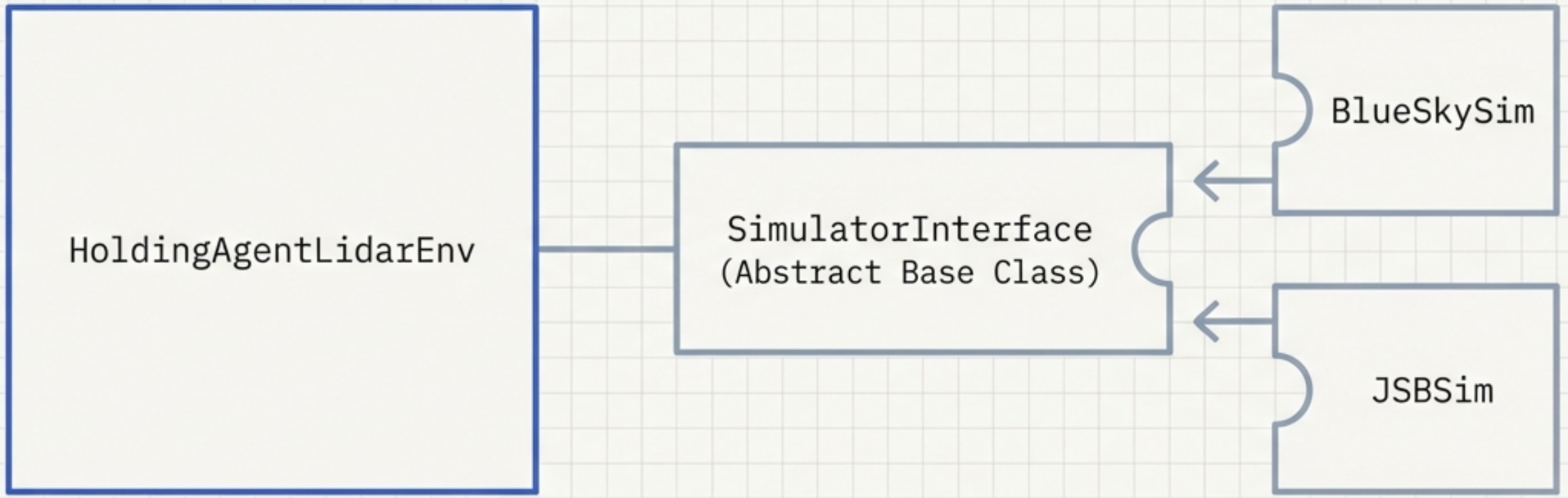
Isolate environment and simulator logic. Changes in one do not break the other, making the system easier to understand and evolve.



REUSABILITY

Simulator interfaces and implementations can be shared across multiple projects and different RL environments.

THE SYSTEM BLUEPRINT RELIES ON A FORMAL INTERFACE CONTRACT



The environment communicates through a `SimulatorInterface` contract, not directly with any simulator. Concrete simulator implementations act as plugins.

THE `SIMULATORINTERFACE` DEFINES THE UNBREAKABLE CONTRACT

Any compliant simulator must implement the following abstract methods, guaranteeing predictable behaviour for the environment.

```
# The SimulatorInterface Contract
class SimulatorInterface(ABC):
    def reset(**kwargs)
    def step(dt)
    def get_aircraft_position(aircraft_id)
    def get_aircraft_heading(aircraft_id)
    def get_aircraft_speed(aircraft_id)
    def set_heading_change(aircraft_id, heading_change_deg)
    def close()
```

SIMULATOR PORTABILITY IS A ONE-LINE CHANGE IN PRACTICE

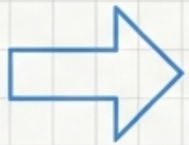
Because the environment is coded against the interface, switching the entire simulation backend is trivial.

BEFORE

```
# Using the BlueSky Simulator
from envs.simulators import BlueSkySim

sim = BlueSkySim()

env = HoldingAgentLidarEnv(simulator=sim)
```



AFTER

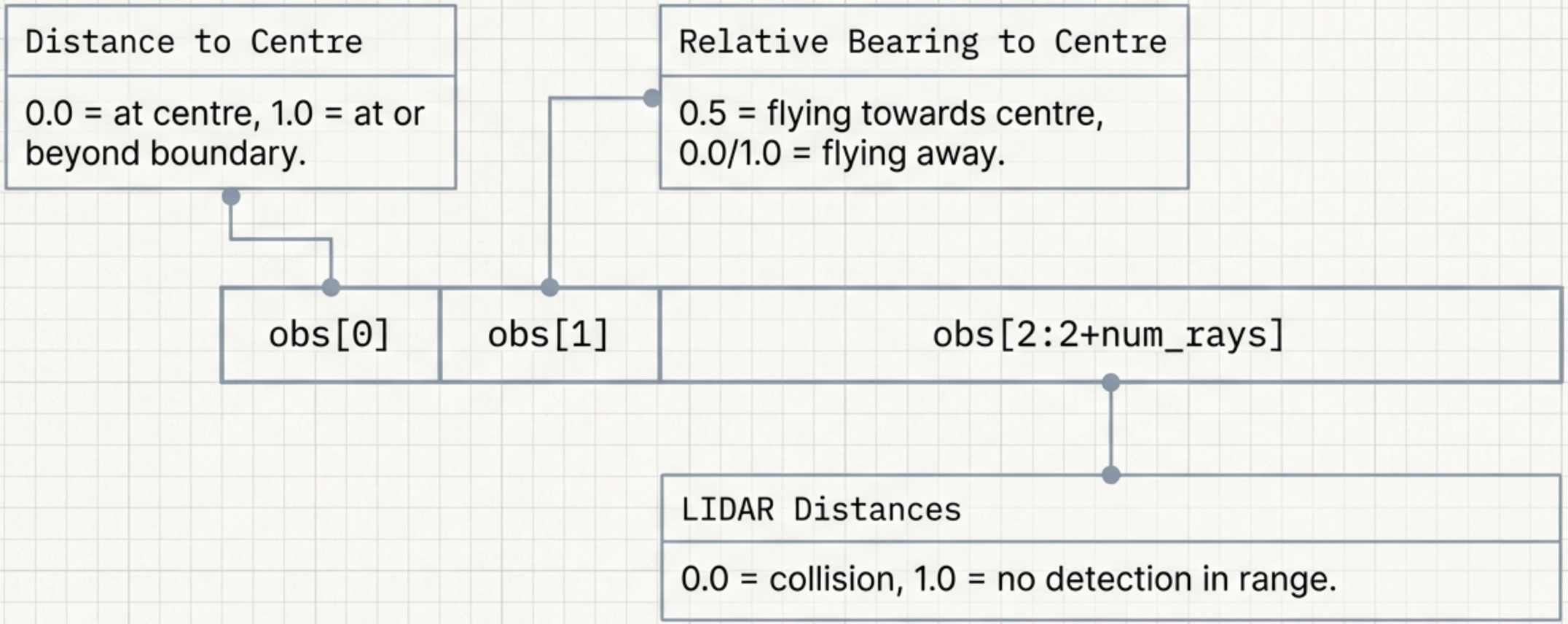
```
# Switching to the JSBSim Simulator
from envs.simulators import JSBSim

sim = JSBSim()

env = HoldingAgentLidarEnv(simulator=sim)
```

THE AGENT PERCEIVES ITS WORLD THROUGH A COMPACT STATE VECTOR

The observation space is a continuous vector where each component is normalised to a [0.0, 1.0] range.



THE AGENT ACTS BY COMMANDING A CONTINUOUS RATE OF TURN

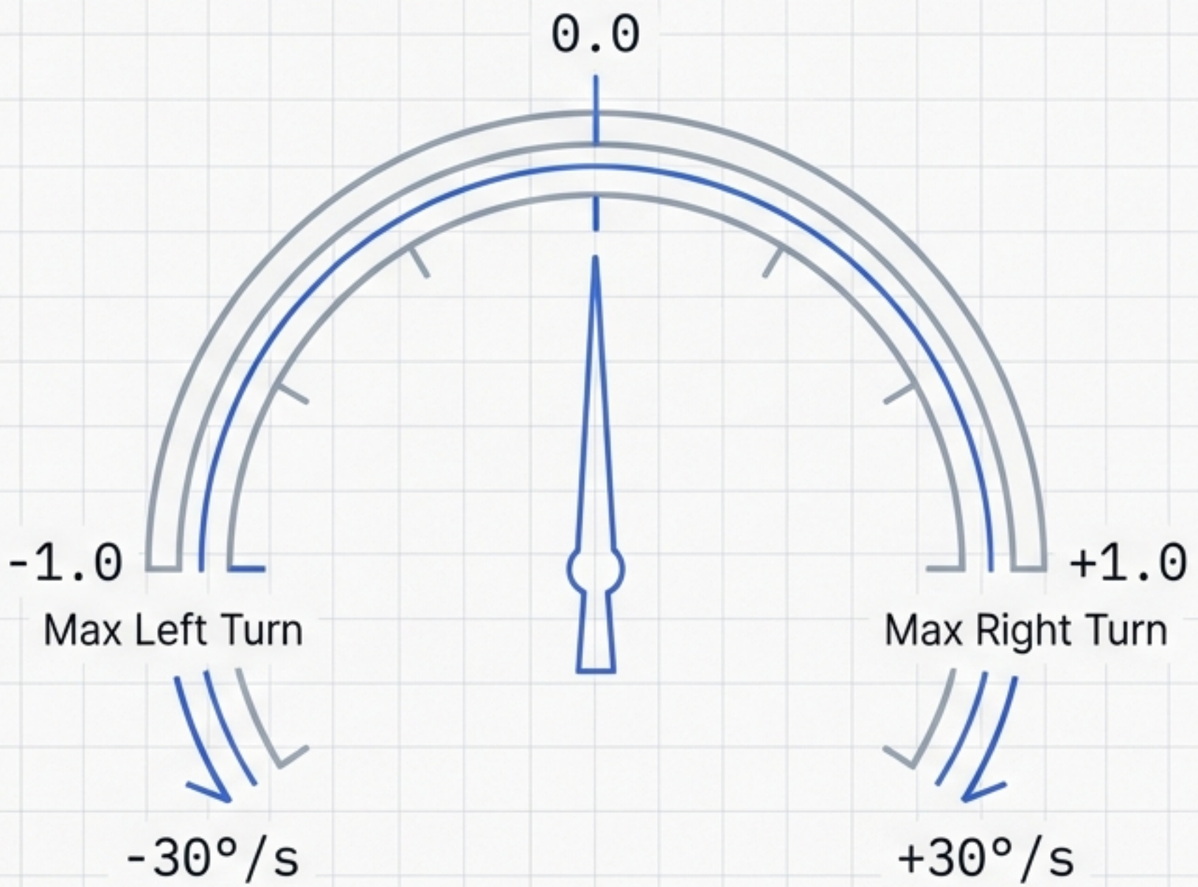
Type
Continuous, `Box(1)`

Range: `[-1.0, 1.0]`

Physical Meaning
The **action** value is scaled to a physical rate of heading change.

Example
With a configured `turn_rate_deg_s` of 30:

- `action = 1.0` → Turn right at 30°/s
- `action = -1.0` → Turn left at 30°/s
- `action = 0.0` → Maintain current heading



THE REWARD FUNCTION MOTIVATES SURVIVAL AND PUNISHES VIOLATIONS

Survival Reward	+0.1	Encourages longer episodes.
Boundary Penalty	-100	For leaving the containment zone.
Collision Penalty	-10	For entering an NFZ.
Proximity Penalty		A smooth negative reward that increases as the agent gets closer to a hazard, providing a useful learning gradient.

Design Rationale

Violations are non-terminating. This is a deliberate choice to allow the agent to learn recovery behaviours, which is critical for robust, on-policy algorithms like PPO.

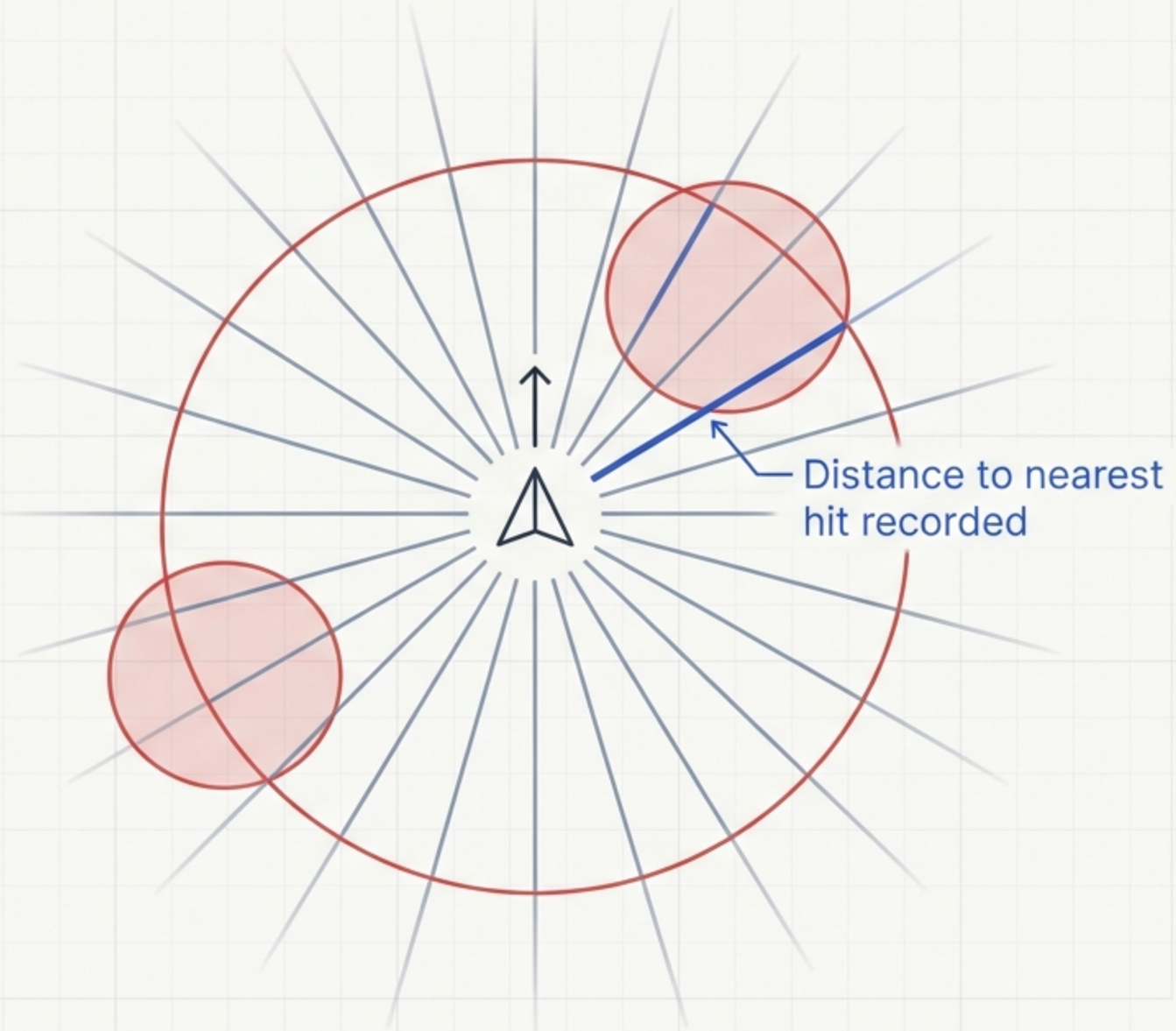
A SIMULATED LIDAR PROVIDES 360° HAZARD PERCEPTION

The system casts a configurable number of rays radially around the aircraft.

Ray angles are relative to the aircraft's current heading (body frame).

Detection is based on efficient ray-circle intersection tests against all NFZs.

The system records only the distance to the *nearest* hit for each individual ray.



COORDINATE SYSTEMS ARE HANDLED WITH A PRECISION-PERFORMANCE TRADE-OFF

The simulator provides **Latitude/Longitude**, but the environment's physics and LIDAR require a **local XY plane** in **metres**.

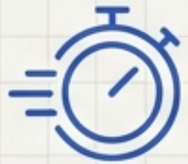
Option 1: Pyproj (High Precision)

Method: Azimuthal Equidistant projection.

Benefit: Preserves true distances and bearings from the centre point. Geodetically accurate for any location or scale.

Use Case: Recommended for large areas (>50-100km radius).

Option 2: Equirectangular (High Performance)



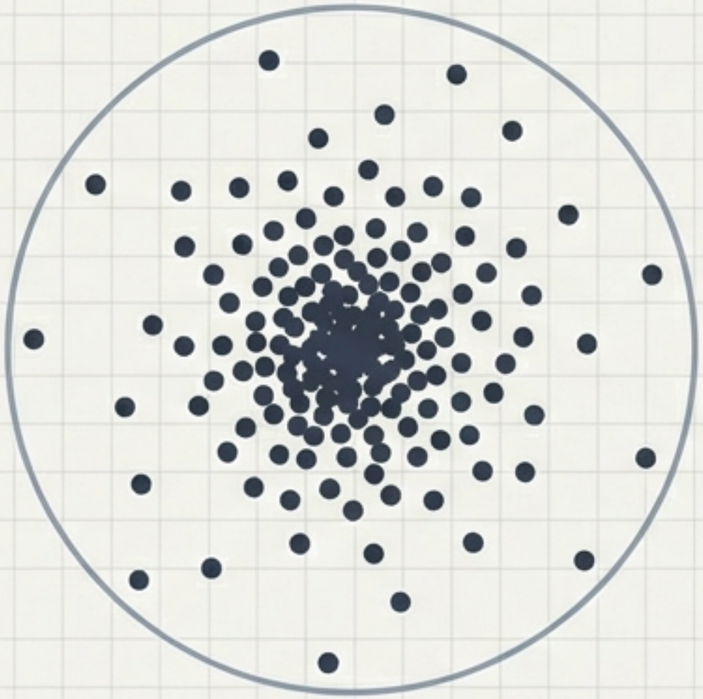
Method: A fast, linear approximation.

Benefit: No external dependencies, significantly faster computation.

Use Case: Excellent accuracy (<1% error) for areas up to ~100km at mid-latitudes (<60°).

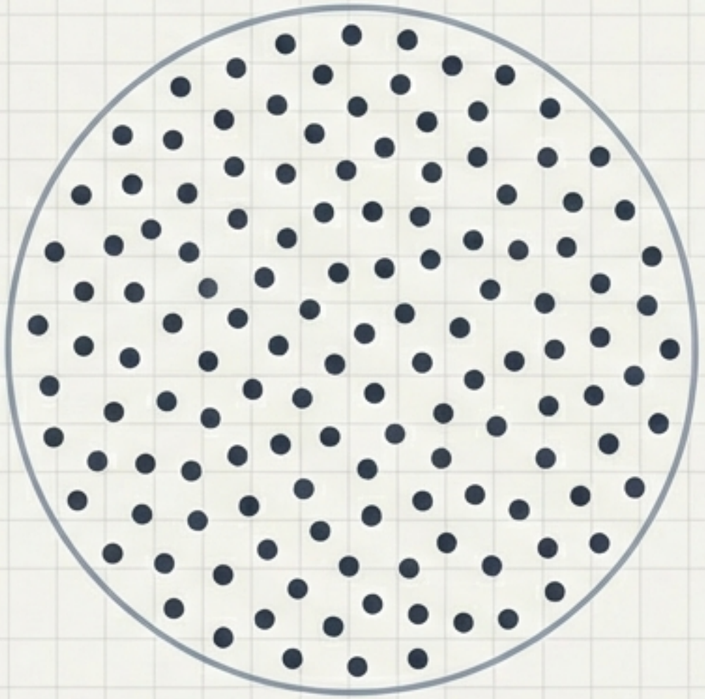
Episode Randomisation Ensures Uniform Spatial Starting Positions

Naive Sampling: Biased to Centre



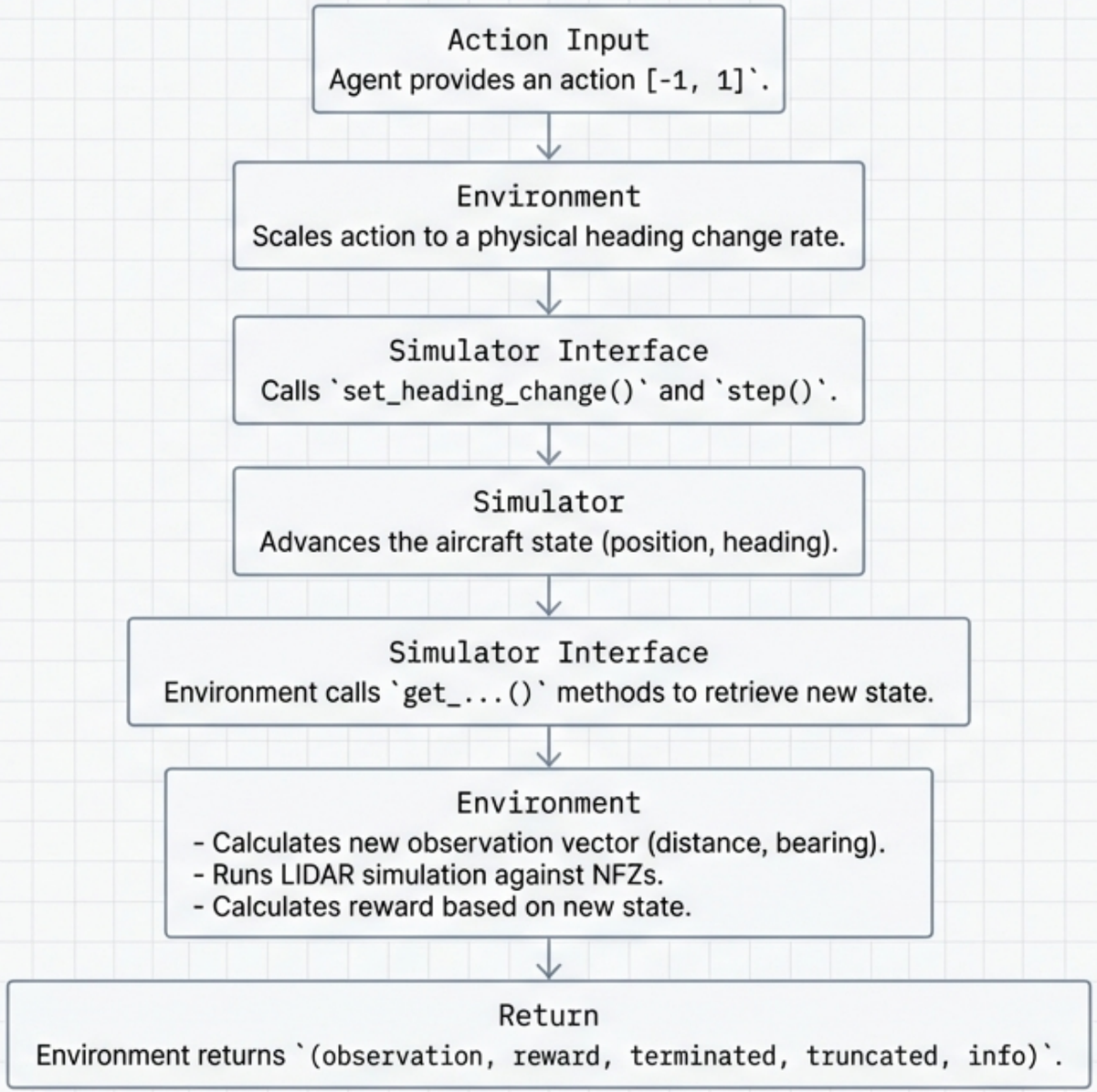
A naive randomisation of starting radius `r = U(0, R)` biases starting positions towards the centre of the holding zone.

Correct Sampling: Uniform Area Distribution



The radius is sampled using `r = R * sqrt(U(0, 1))`. This ensures that the agent's starting position is uniformly distributed over the entire circular *area*.

The Step Execution Flow Orchestrates a Clear Sequence of Events



Getting Started & Extending the Blueprint

1. Key Dependencies

- **Required**: `gymnasium`, `numpy`
- **Optional**: `pyproj` (for high-accuracy geodetics)

2. Actionable Next Steps

To use with BlueSky:

1. Implement the methods in `envs/simulators/bluesky_sim.py`.
2. Connect to your BlueSky server instance.

To add a new Simulator (e.g., JSBSim):

1. Create `envs/simulators/jsbsim_sim.py`.
2. Implement the `SimulatorInterface` contract.
3. Instantiate the environment with your new simulator class.

3.