

# Shower world

## 1. Introduction

ShoverWorld is a custom grid-based reinforcement learning environment using the Gymnasium framework aimed at exploring spatial interaction, resource-constrained planning, and multi-step decision making. There are also moveable box tiles, deadly lava, barriers and growing perfect-squares that allow special transformation moves. Each step combines cell selection with an action type, enabling agents to execute directional pushes, unleash special abilities, and even manipulate massive connected box chains with stamina-based costs.

A special perfect-square-finder when identifies isolated  $2 \times 2$  &  $3 \times 3$ + box areas that could be converted using Barrier Maker/Hellify, with the squares aging themselves to produce ever-mutating architecture. The observation space records the grid layout, agent position, stamina, and action history, while rigorous map-loading validation confirms file integrity via token validation and shape validation. An interactive Pygame renderer visualizes grid states, agent movement, and key metrics, offering a convenient interface for debugging and manual exploration. `test.py` contains a demo loop that takes random actions and prints environment updates step-by-step — letting you play around with the system right away. ShoverWorld also includes a strong test suite confirming illegal moves, map parsing, perfect-square detection, push-chain dynamics, stamina modifiers, and stationary box tracking. Predefined map files facilitate reproducible scenarios that standardize test and training conditions for different reinforcement learning algorithms.

## 2. Environment.py

ShoverWorld is implemented as a custom Gymnasium compatible class that defines a rule-driven grid world in which an agent must interact with spatial elements under deterministic, physics-inspired constraints. The world is illustrated as a 2D grid of empty spaces, sliding box tiles, lava and some immovable walls. Environment initialization allows for configurable options such as grid size, maximum episode length, initial stamina, force costs, perfect-square aging thresholds, and optional loading of custom map files. When loading a map, rigid checking of the row lengths, token values and boundary rules is performed to make sure everything fits the required format prior to the simulation starting.

The observation space consists of the grid state, the agent's position, stamina, and historical interaction meta data such as the last chosen cell and action. The compound action space involves a target row, target column, and an action type, allowing you to take fine-grained spatial control actions. Directional moves trigger the chain-pushing mechanism, in which any adjacent boxes in a direction are found and moved together if possible. Stamina is consumed based on chain length, and invalid operations—like pushing from empty cells or barriers—incur a smaller stamina penalty and leave the map unchanged.

In addition to pushing, the environment features two special architectural functions: Barrier Maker and Hellify. These all rely on a perfect-square detection module that checks for isolated  $2 \times 2$  and  $3 \times 3$ + boxes. Barrier Maker transforms smaller squares into permanent barriers and awards stamina as the square's area. Hellify takes bigger squares and clears their outline and fills the inside with lava, nuking all boxes inside. Perfect squares persist across timesteps via an aging mechanism executed inside `step()` — once a square crosses an age threshold, it dissolves automatically, adding long-term structural evolution to the grid.

The `step()` method controls every environment state change. It increments timestep, applies the action, updates stamina values, squares aging, rebuilds the stationary box structure, processes destruction events, and decides whether to terminate the episode — due to stamina depletion, maximum timestep, or full removal of boxes. The `matching` info dictionary contains diagnostic information, such as destruction statistics, push-chain lengths, validity flags, lava interactions, and the current set of perfect-square candidates. `reset()` resets the environment, restoring stamina, clearing internal trackers, rebuilding perfect-square structures and reloading the map if given a filepath. Returns the first observation dict to have a consistent starting point every episode, making the environment fully compatible with RL training pipelines that utilize deterministic resets and stable initial conditions.

All told, this setting provides a hard, rule-laden grid world for training RL agents on spatial reasoning, resource management, and sequential action planning, backed by an explicit transition model, rigid validation logic, and reproducible reset/stepping functions.

### 3. `gui.py`

The GUI features an interactive minimalist rendering of the `ShoverWorldEnv` environment in PyGame. It's intended to facilitate human inspection, step-by-step debugging and interactive exploration, and is enabled only when the environment's `render_mode` is set to "human". The main does PyGame initialization on first use, opens a window, and calculates cell sizes with respect to the grid. It then loops waiting for user input — mouse clicks to pick up/move agent and keyboard shortcuts for general moves (directional moves, special interact, reset, quit). Every frame, the window is wiped and the grid is painted based on tile types — lava, empty, moving boxes or barriers. A light grid overlay helps to define the cells and the active cell is highlighted. The HUD reports important environmental information like the timestep, current stamina, last action, selected cell coordinates, and any terminal messages — giving you instant feedback while interacting or debugging. Render updates are throttled to preserve responsiveness and clarity, with PyGame event pumping rendering the interface interactive. The GUI changes the environment either by direct state changes, such as selecting a cell, or by taking actions through the environment's `step` function. It doesn't output anything on its own, all logging or observation output is done by the calling script.

As a separate function, the GUI is decoupled from the underlying environment logic, promoting modularity and making testing and re-use easier. Hardwired selections for colors, fonts, and update pacing prioritize readability and debuggability over speedy animation or visual flair.

### 4. `test.py`

The test script offers a nice interface for executing and exploring the `ShoverWorldEnv` environment. Its primary intention is to showcase environment behavior, prototype validate core mechanics, and offer a reproducible loop for agent-environment interaction observation.

The script initializes an instance of the environment with configurable parameters like grid size, initial agent position, and render mode. When `render_mode='human'`, the PyGame-based GUI is launched, enabling interactive inspection and direct control of the agent. Otherwise, the environment launches in headless mode for automatic scoring. A core loop steps through environment steps, taking actions from a hardcoded sequence, randomly, or from a human via the GUI. Each step calls the environment's `step` function, which returns the next state, reward, and status indicators. These outputs can be printed or logged by the script to view environment responses, such as agent position, stamina, and terminal

condition updates. If rendering is on, each iteration flashes the GUI. There is the current grid, with the selected cell highlighted, and the HUD showing timestep, last action, selected cell and terminal. Frame pacing and PyGame event pumping keep the interface responsive during step-wise interaction. A small pause is inserted with `time.sleep(0.01)` to slow things down for human readability. The loop exits gracefully upon reaching a terminal condition defined by the environment (e.g., clearing all boxes, running out of stamina, or max timestep). Then the episode return is printed and environment closed cleanly. This driver program hence acts as a simple test loop, illustrating how the environment reacts to action and giving both textual and graphical feedback.

In general, the test script serves as a demonstration and validation of. It allows users and developers to probe agent behavior and environment dynamics in a controlled, reproducible setting. Its design preserves a clean separation of environment logic and test routines, consistent with the interactive model used by the GUI and conducive to modularity and reuse.

## 5. test

### 5.1. test\_invalid\_actions.py

This script allows a controlled validation of invalid action handling in `ShoverWorldEnv`. Its main objective however is to make sure the environment reacts strong and consistent when the agent tries moves or interactions which are forbidden under the current state/environment rules. The script starts up a normal instance and runs through some actions hand picked to cause invalid states — like pushing boxes through walls, moving off the grid, or acting when the agent is out of stamina. For each, environment's `step` function is invoked. The returned state, reward, and status information are recorded enabling developers to confirm that invalid actions don't foul the environment state, and that appropriate penalties or restrictions are applied consistently.

By systematically changing the environment state and performing these checks, the test ensures the environment correctly enforces rules, forbids illegal moves, and still updates agent resources. We also created this unit test to automatically check our core environment mechanics, ensuring that `ShoverWorldEnv` remains reliable as development continues or when plugged into reinforcement learning pipelines. It also verifies that edge cases in the step logic don't cause surprising behavior or crashes.

The script also enforces a well-defined loop over test actions, looping over test actions until the sequence ends or is terminated. And when you're done, your environment is neatly shut down. `test_invalid_actions.py` is a specialized check that allows you to verify your environment constraints and that the environment behaves consistently to invalid operations, rather than being mixed in with the core environment logic.

### 5.2. test\_map\_loading.py

The `test_map_loading.py` script tests the map-loading in `ShoverWorldEnv`. Its main purpose is to translate predefined or custom maps to load into the environment grid and represent internally using the environment's tile conventions.

The script starts the environment with various map configurations - comprising valid prebuilt maps, and optionally, custom layouts. For each map, the environment builds the underlying grid, positions agents and objects, and applies any environment parameters like walls, lava tiles and initial box placements.

During execution, the script verifies that map elements are properly loaded and the resulting environment state corresponds to the expected layout. This means checking obstacle, box, agent start positions, and any special tiles. It also validates that invalid map definitions—like overlapping objects, out-of-bounds tiles, or otherwise malformed layouts—are handled gracefully. The test then attempts to load each malformed map using the environment’s `load_map()` method and asserts that a `ValueError` is thrown in each case. This causes the engine to vigorously test map validity, avoiding sneaky configurations that could break gameplay. By testing different classes of map errors, the script helps preserve the integrity and safety of `ShoverWorld` when consuming external map data — something you need for reproducibility and to avoid runtime exceptions while running RL experiments.

Each map-loading test seizes the environment’s internal state post-initialization, enabling the developer to verify that the grid representation matches the intended design. Terminal states like an empty map or missing agent are verified so the environment throws proper exceptions/warnings instead of runtime errors when stepped through again. It loops through many map scenarios, a reproducible way to check map-loading mechanics. Once all tests are complete, the environment is neatly closed, freeing resources and maintaining a consistent testing context. `test_map_loading.py` is a specific test of the environment’s map-loading capability. It makes sure maps are loaded properly, internal states are consistent, malformed maps are rejected properly, edge cases are handled reliably and allows safe usage of `ShoverWorldEnv` in development and automated RL experiments.

### 5.3. `test_perfect_square.py`

`test_perfect_square.py` ensures that the `ShoverWorldEnv` environment properly addresses grid layouts that must be perfect squares. Its main motivation is that all grid-based computations, indexing and environment logic assume a consistent, square-shaped grid — otherwise, things might break at runtime or behave unintentionally when agents interact. The script starts up environment instances with differently sized grids, both valid and invalid. For every configuration, the environment does internal validation to verify if the grid dimensions satisfy the perfect square constraint. Invalid configurations should throw the right exceptions, like `ValueError`, so that you can only use sensible and safe grid layouts. While running, the test records the environment’s internal state — the grid and how tiles like empty cells, obstacles, and boxes are arranged. It also claims that valid perfect square grids are properly initialized and that agent positions, barriers and other such things are established. For invalid grids, the test ensures that the environment refuses to operate and that no additional operations are allowed until a valid configuration is supplied. The script runs through several grid situations, to methodically test both normal and edge cases. And when finished, the environment is nicely closed, freeing resources and keeping a stable testing context.

`test_perfect_square.py` is a more focused test that validates grid integrity — making sure that `ShoverWorldEnv` properly enforces square-grid assumptions, crashes safely on invalid layouts, and remains consistent in all internal calculations.

### 5.4. `test_push_chain.py`

`test_push_chain.py` aims at verifying chain push handling in `ShoverWorldEnv`. Its main purpose is to make sure when the agent pushes a sequence of adjacent boxes, the world appropriately updates their locations based on the rules — obstacle collisions, barriers, grid boundaries.

The script starts environment instances with preconfigured layouts designed to test different chain push situations. Each configuration has push sequences of boxes, barriers and open space to test a variety of push results – successful multi-box pushes, push blocked by barriers and edge-of-grid restrictions.

For each test case, the agent performs push actions, and the environment’s step function is invoked. The returned state, reward and status are also captured, letting developers check the chain push logic is working as expected. The script verifies that boxes move only when allowed, that chain reactions obey the environment boundaries, and that illegal pushes do not corrupt the environment state.

The tests span a variety of cases, such as short and long chains, barrier pushes, and special tiles like lava or no-go zones. This set of evaluations enforces that normal and edge cases chain push operations are processed consistently.

When all test cases have been run, the environment is cleanly closed, freeing resources and ensuring a reproducible testing context.

`test_push_chain.py` provides a targeted check of the environment’s chain push mechanics. It verifies that multi-object interactions are implemented correctly, that environment rules are consistently enforced, and that the environment remains stable under complex manipulation scenarios.

#### 5.5. `test_stamina.py`

`test_stamina.py` verifies stamina is handled within `ShoverWorldEnv`. Its main purpose is to verify that the agent’s stamina gets properly updated from actions, and that the environment imposes restrictions when stamina is exhausted. The script sets environment instances with default parameters and runs stamina-draining action chains. These chains contain regular moves, pushes, specials and tile interactions that may drain extra stamina. For each action, `step` is called on the environment, which returns state, reward and status information. The script records these outputs to confirm that stamina gets decremented properly, that the agent can’t act without stamina, and any penalties or restrictions it applies.

The tests also include edge cases, like depleting stamina to zero, performing invalid actions with insufficient stamina, and verifying that stamina caps are applied at every step. This makes sure that the world acts predictably and no illegal actions can circumvent stamina limitations. Once done, the environment is tidily shut down, freeing resources and preserving a reproducible test context.

`test_stamina.py` is a specific test for the agent’s resource management systems. It verifies that stamina is properly tracked, that stamina-based action restrictions are imposed, and that the world is resilient under fatigue conditions. This test is essential for dependable interaction dynamics and makes sure `ShoverWorldEnv` is safe to utilize in development and RL experiments alike.

#### 5.6. `test_stationary.py`

The `test_stationary.py` script tests the agent when nothing happens. Its main goal is to make sure that the world properly preserves state consistency and passively agent behavior over time.

The script instantiates the environment and sets up the agent in a ‘do nothing’ action. The environment’s step function is called with a no-op or idle action to represent the agent doing nothing. In addition, returned states, rewards and status indicators are recorded to validate that the environment does not unintentionally modify agent positions, object locations, or other state variables. The tests also ensure that environmental mechanics such as stamina, timers, and

terminal conditions act consistently while the agent is idle. Edge cases, such as multiple timesteps passing without action, are tested to make sure nothing unexpected happens and that the environment remains stable. When finished, the environment is cleanly closed, freeing resources and maintaining a reproducible test context.

`test_stationary.py` provides a focused test for validating passive agent behavior. It validates that the world keeps proper state updates, applies consistency, and manages idle scenarios robustly — enabling both safe development and plugging into reinforcement learning experiments.

## 6. map

Here is where the potential maps to use for the game are

## 7. conclusion

ShoverWorldEnv provides a modular and stable testbed for investigating agent interactions in a grid world. Its modular design, which balances robust core logic, clear map structures, and an optional interactive GUI, means it's easy to test and debug — and experiment with different scenarios. Thorough testing makes sure the world acts coherently, managing everything from invalid moves and stamina consumption to map streaming and intricate push chains. These tests provide assurance that the environment is robust, deterministic and secure to operate in, be it for development, showcasing or RL experiments.

By separating visualization, core mechanics, and validation, the environment is easy to extend or modify without breaking functionality. Map validation, tile definitions, and resource tracking are all guardrails that keep things from breaking in surprising ways — everything is reproducible and reliable. In short, ShoverWorldEnv provides a robust, extensible, and well-documented environment for exploring agent-environment interactions in grid-world contexts. It makes probing agent behavior intuitive yet preserves the rigor necessary for reproducible experiments — giving you a strong base for learning, testing, and creative experimentation.

Bahare Motamedi

Maryam Soleimani