RBE 550 A1: Flatland Report

Soumaya El Mansouri

Implementation:

For this assignment, we had to create a little chase game where a hero must reach its goal location in a maze while avoiding a group of hostile robots. This implementation is written from scratch in Python and uses PyGame to simulate the environment and chase. Object-oriented design was used to structure the logic, where each agent is an "entity" with its own position and path generation.

Source Code Overview:

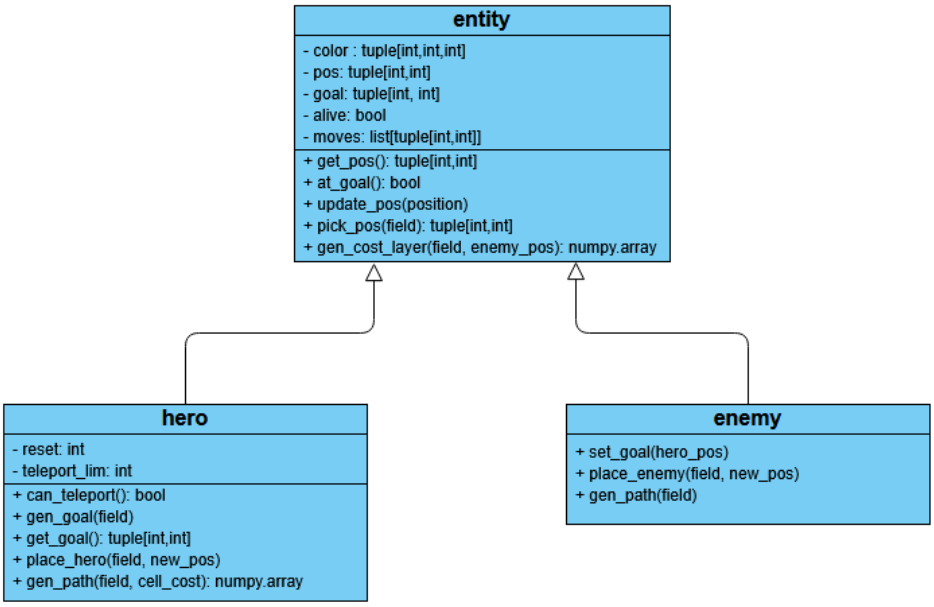
File	Content and Purpose
main.py	Runs the simulation, chase, and all instances of the relevant objects
entity.py	Houses the agent classes and logic, including path planning
env_handler.py	Handles the map class and logic, generates the randomized map

This project is comprised of 3 files: *main.py*, *entity.py*, and *env_handler.py*. Entity has 3 classes: entity, hero, and enemy where hero and enemy are subclasses of entity. Env_handler has map, which just generates the field and makes it available to be referenced. Main has all the instantiated entity, hero, enemy, and map objects as well as the PyGame display and other game logic.

Program Structure and Classes:

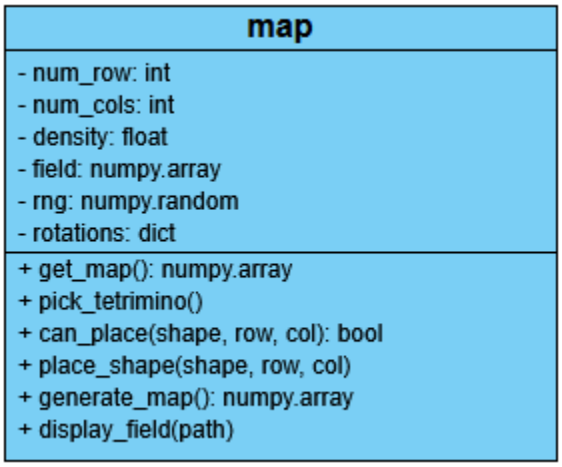
Entity:

Entity is the parent class for the hero and enemy subclasses.



Map:

Map is a basic class that simply generates the field with randomly placed tetriminos at 20% density. All the code is reused from the Turtles assignment, just tweaked and put into a class.

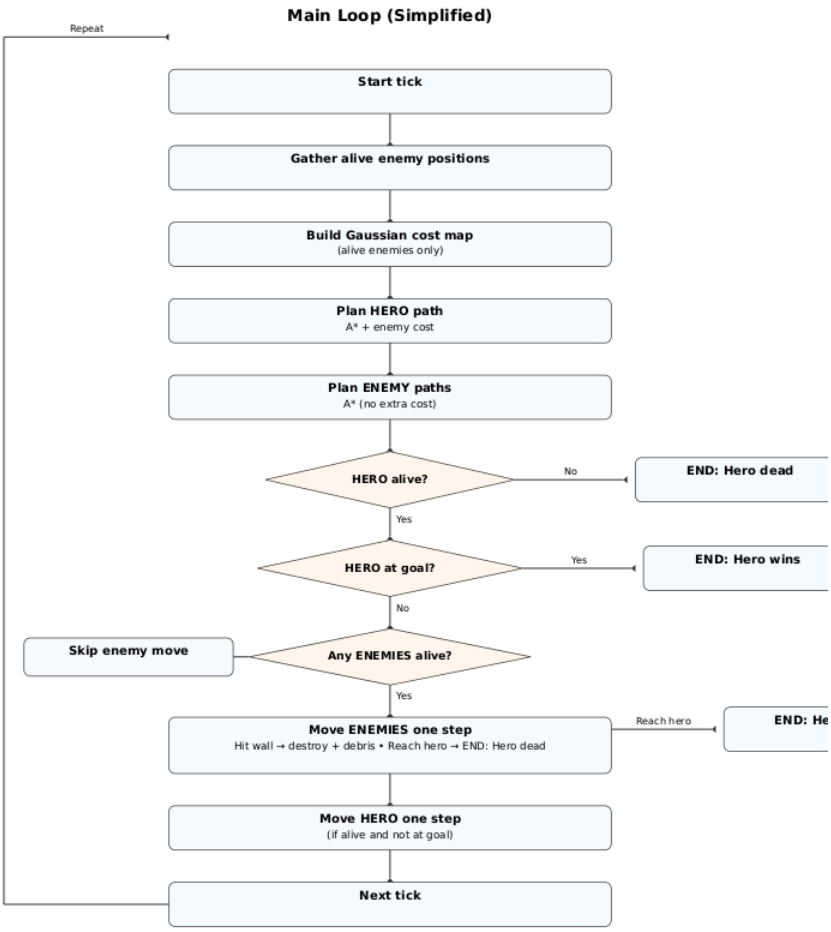


Main:

The main file puts everything together in *run()*. The map, hero, and a list of 10 enemies are instantiated at the top of the program. After being generated, the map object is never called again. The agents are all given random start points, where the hero is also given a random goal and the enemies are given the hero's position as a goal.

Moving onto the loop within *run()*, all the agents regenerate their path every time step, this is due to the dynamic nature of the game. Additionally, every few time steps, the hero gets bored and teleports to a random free spot 3 times. The following is run every time step until the hero is dead or reached its goal:

1. Enemy and hero paths are generated
2. Enemy Gaussian kernel cost map is generated based on valid enemy positions (ie not dead)
3. If the hero is still alive, not at its goal, and the enemy/s are alive, they are moved to their next planned position.
4. If the hero is still alive and not at its goal, it will be placed on its next planned step towards the goal (taking into account the enemy position cost map).
5. If the enemy hits a wall, it is destroyed and its debris becomes an obstacle in the field
6. If an enemy actually reaches the hero, it dies and the game ends
7. If the hero manages to reach the goal, the game ends



As described above, so long as the hero has not been destroyed by the robots (*hero.alive* is True), it will keep avoiding robots until it reaches its goal.

Path Planner:

All the agents use a basic A*algorithm to generate their paths (*hero.gen_path(...)* and *enemy.gen_path(...)*). The key difference is that enemies do not care to run into obstacles and the hero must avoid the enemies so it implements a different cost function.

A* Implementation:

The heuristic is simply based on octile distance, so 1 cost for straight cells and $\sqrt{2}$ for diagonals (which incentivizes diagonal movement). The base cost is the same as the heuristic. The enemies only take into account this base cost, the hero adds a cost layer based off enemy positions.

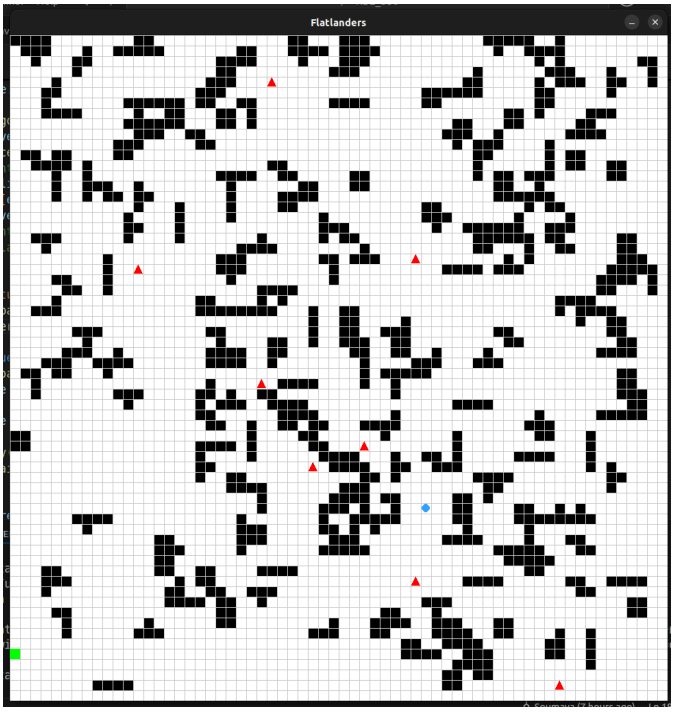
Hero-Enemy Avoidance Cost Layer:

This cost layer is generated using a Gaussian kernel, which is handled by the entity parent class (*entity.gen_cost_layer()*). This approach makes it incredibly easy to scale enemy avoidance zones for increasing enemy agents. Each enemy position in a provided list is added into an impulse map. Then, the *gaussian_filter(..)* function from the SciPy ndimage module is given this impulse map to generate the cost layer that can be used by *hero.gen_path(...)* in its cost function to avoid the enemies. The *hero.gen_path(...)* calculates this enemy cost using the *penalty_at(...)* helper with this cost layer. The hero uses the same base cost as the enemy but it adds this enemy cost to get the final cost. This implementation also allows the hero to still enter enemy space if it has no choice, which leads to a smoother path to the goal.

Enemy Zeal:

The enemy simply uses the base cost and does not take into account any additional cost. However within *enemy.gen_path(...)*, if the current cell occupied is blocked, instead of avoiding that point, it kills the enemy. In this implementation, the enemy always knows where the hero is going, so it is given the goal of moving to the cell the hero means to go to next. This spices up the chase.

Results:



It took some tweaking, but the simulation works smoothly. Currently, the hero and enemies spawn and chase/avoid each other until either the hero dies or reaches it goal. All the enemies successfully turn into debris

A common issue was out of bound indexes and the enemies deciding to sit inside the walls but not die. Originally, the enemies also were aiming to get to the hero's current position, which commonly lead to a trail of enemies behind that could not catch up. This was changed to allow them to sneak in front of the hero to intercept it instead of catching up then moving behind it. It was interesting to see how the hero adapted to enemies surrounding it because the cost map allows it to get close if there is not alternative route. So far, it has not gotten cornered and stuck, it simply tries to move past the enemies. This, of course, still easily leads to its death.

References and Notes:

- Credit to ChatGPT for:
  - Main loop flow chart above
  - Within the source code: graphics and Gaussian kernel cost layer implementation, marked relevant sections in the code
- https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.gaussian_filter.html