Task 5: OctoSpace

Gameplay outline:

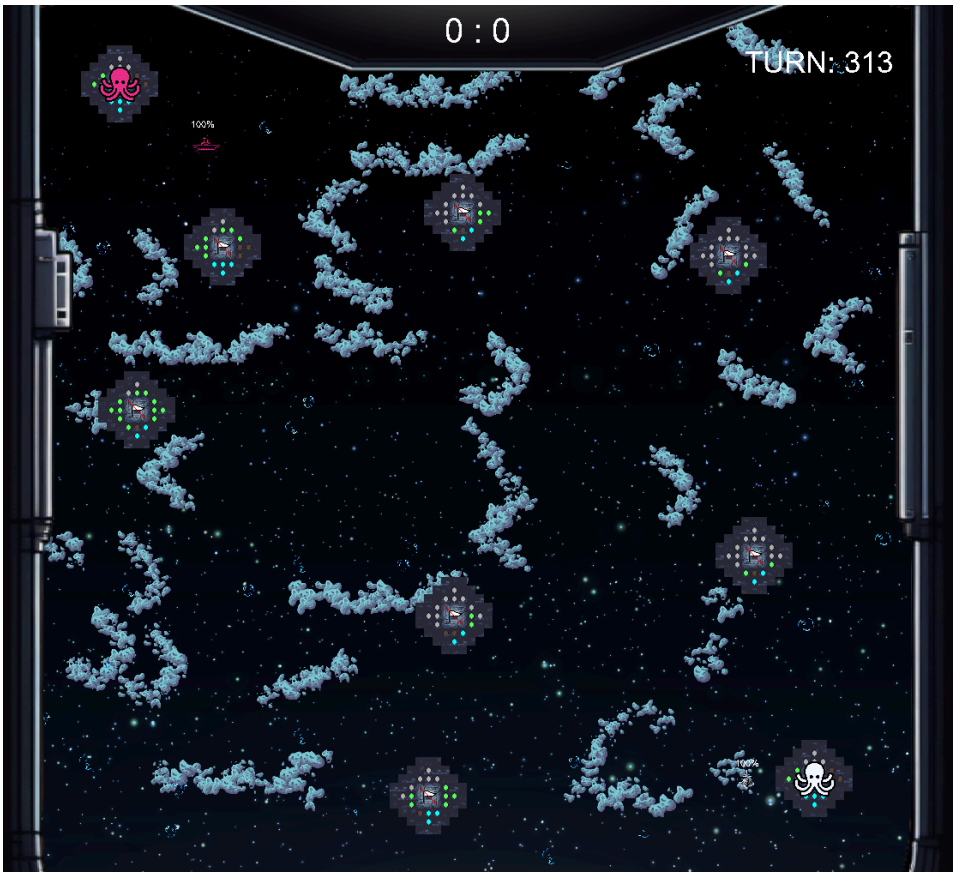
Two space-octopus nations have appeared in the planetary system, competing for power. The player takes on the role of one of the nations and has the task of driving the opposing nation out of the system. To do this, you must write a program that will strategically wage war in space.

Map:

The map is 2D 100 x 100, composed of tiles representing space, obstacles in this space, or planets. We can find on the map:

- Space
- Asteroids
- Planet surface
- 4 types of resource fields
- Ionized matter fields
- A fog (not an actual tile on the map, but indication, that player hasn't discovered this tile yet)

Players start with 1 fully populated planet each.



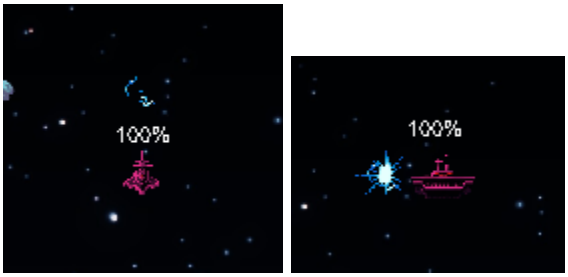
Ships:

Players can move around space in ships. These ships can fight enemy ships and colonize planets. In addition, they will allow players to explore the map (initially, the entire map, except for the starting planet, is seen by the player as fog), because by moving, they will discover part of the surrounding terrain. Each turn, the player can move or shoot with each of his ship.

Ship movement:

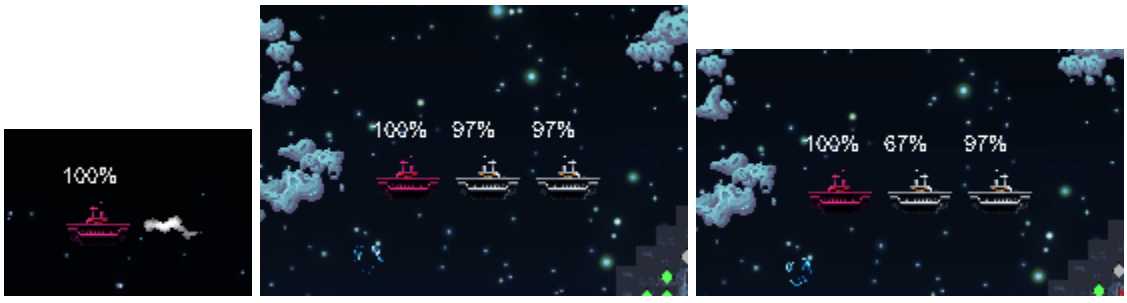
The base movement speed of the ship is set to 1 tile per turn. However, there are 2 modifiers for the movement:

- If the ship enters an asteroid field, it gets a cooldown for movement for 3 turns (basically means, that ships move 3 times slower in the asteroids). Also, ships get damaged.
- If the ship at the start of the turn is staying at the ionized matter field, the maximum speed is raised up to 3 speed. However, the player may decide, that he wants to move the ship only by 1 or 2 fields this turn.



Ship combat:

Ship shots can reach up to 8 squares, after which there is a cooldown for 10 turns, during which the ship cannot shoot. Each hit deals 30 points of damage to the opponent and can only hit 1 ship (even if the animation overlaps 2 ships, only the first one will receive damage). Ships can also heal themselves by entering the tile on a planet, that is already under their government. The healing speed is set to 1 HP/turn.



Observation space:

- game_map: whole grid of board_size, which already has applied visibility on it
- allied_ships: a list of all currency available ships for the player. The ships are represented as a tuple:
    - ship_id: int [0, 1000]
    - position_x: int [0, 100]
    - position_y: int [0, 100]
    - health_points: int [1, 100]
    - firing_cooldown: int [0, 10]
    - move_cooldown: int [0, 3]
- enemy_ships: same, but for the opposing player ships. There are only ships, that are visible to the player.
- planets_occupation: for each visible planet, it shows the occupation progress:
    - Planet_x: int [0, 100]
    - Planet_y: int [0, 100]
    - Occupation: [-1, 100]:
        - (-1) - means unoccupied planet
        - 0 - planet occupied by player 1
        - 100 - planet occupied by player 2
        - Values in between indicate an ongoing conflict for the ownership of the planet
- resources: int [0, 1000] - current resources available for building

Action space:

- ships_actions: player can provide an action to be executed by every of his ships. The command looks as follows:
    - ship_id: int [0, 1000]
    - Action_type: int [0, 1]
        - 0 - movement
        - 1 - firing
    - direction: int [0, 3]:
        - 0 - right
        - 1 - down
        - 2 - left
        - 3 - up
    - speed (not applicable when firing): int [1, 3]
- construction: int [0, 10] - a number of ships to be constructed

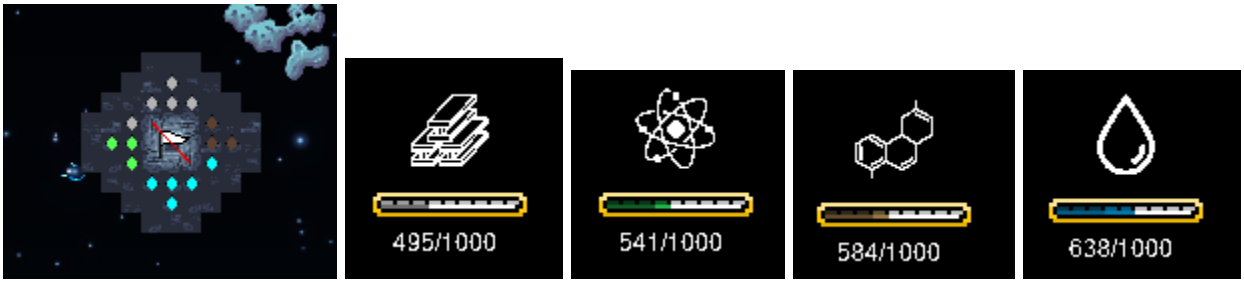
Vision on the map:

At the start, players see only some area around their starting planet. Each time they move a ship, the area around the new ship's position is revealed to the player. Also, when the player captures a planet, the surrounding area automatically gets added to the player's vision field.

Economy:

To create new ships, players need resources. Each planet consists of 16 squares, and each square represents one of these 4 types of resources, and depending on how many of these tiles the player controls, the amount of that resource is produced each turn. The starting planets have 4 squares of each resource, but others may differ in this ratio. Production per turn is given by the simple formula:

$$\frac{number\ of\ resource\ fields}{4}$$



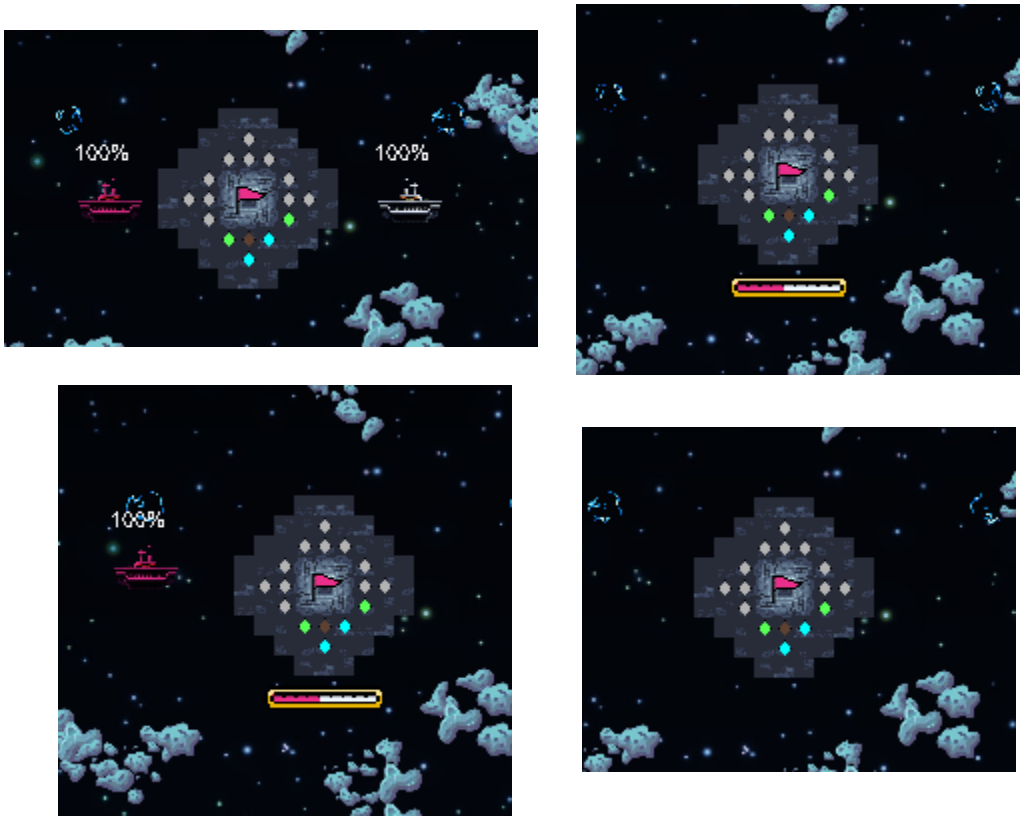
Planet colonization:

When a ship enters a field with a flag in the middle of a planet, several things can happen:

1. If the planet does not belong to anyone, it is immediately taken over by the player

2. If the planet belongs to an opponent, the occupation of the planet begins, which will last several turns.

3. If the planet is currently occupied by an opponent, this will neutralize their occupation (occupation progress will stop at 60%, for example, and the planet will remain under the player's control)
4. If the planet is under the player's control, but occupation progress is stopped (for example, at 60%), a secondary occupation of their own planet begins to 100%



5. If the planet has 100% occupation progression and is under the player's control, the ship begins to regenerate.

In case of situations 1 - 4, the ship is lost

Game Goal:

A round ends when one of the players loses their home planet or after a specified number of turns (1000). Each match is played with the sides swapped, i.e. twice. After winning a round, the winner receives 1 point, in case of a draw both players receive 0.5.

Visualization tool:

By passing the *render_mode="human"* parameter to the script playing the matches, a visualization in pygame will be displayed. You can also pass the *turn_on_music=True* parameter so that sound effects and music can be heard during the game.

Teams' task:

Teams receive a class skeleton for the bot. The final implementation can be basically anything, but it must contain implemented classes from the skeleton, because they will be called during the game. Additionally, you can send weights for models, if any are included in your solution. The number of files containing weights must be below 5, and

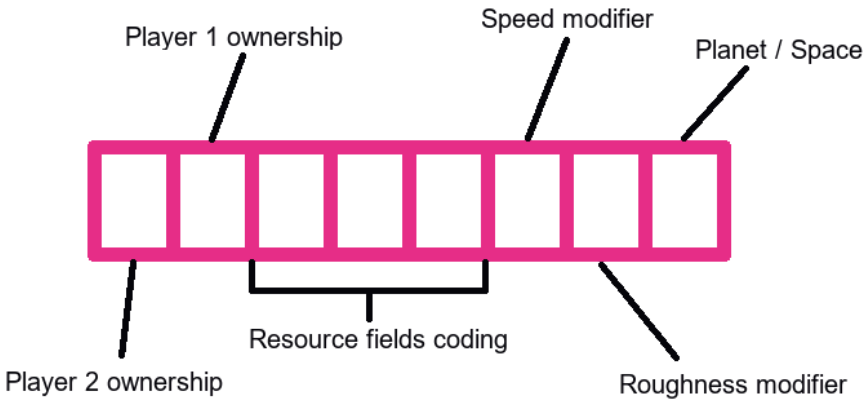
the maximum size of all files must be below 150 MB. You also receive code with an environment for testing your solutions. The libraries that can be used are limited. The solution can use:

- Numpy
- Pandas
- PyTorch
- Scikit-learn

And their dependencies.

Map encoding:

Each field on the map is encoded in 8 bits, as follows:



For example:

- 00000000 - space tile
- 00000001 - land tile
- 00000010 - asteroid field
- 00001001 - resource field 1
- 10010001 - resource field 2, owned by player 2

However, the player receives a map with a vision mask applied, so remember, that fields that the player cannot see **will be filled with the value -1**. Such map encoding allows for efficient extraction of important information from the map using bitwise operations, such as:
Field & 64 == 64 - the field belongs to player 1
Field & 128 == 128 - the field belongs to player 2

Running the test game:

The run_match.py script should be given paths to 2 files that contain the implementations of 2 bots. It can be the same file as well.

Implementation tips:

The implemented bot class must contain functions that are in the class skeleton - these functions will be called by our script playing matches.

You can test your solutions by playing matches through the provided script for testing. The game logic is the same as in the checking script.

Pay attention to the implementation of the .to() function. It allows you to transfer the work of your models to the GPU. This is necessary for the efficient operation of the checker and will allow you to stay below the time limit for the move of your implementation.

During training, you can use the gym environment, that was created “OctoSpace-v0”, but remember to import the octospace.octospace (there’s gym environment registration in the __init__.py()).

Submission:

You are allowed to submit 1 time per 10 minutes. However, notice that the only information you get from your submission is if your submission was correct - it doesn't play any matches with other players right away! The matches between all players will be played every full hour and the leaderboard will be updated afterward. The endpoint for this task is: **http://149.156.182.9:6060/task-5/submit**

What not to do:

- The basic rule is not to include code that could be dangerous for our checker. Implementations, in addition to basic checks, will also be manually checked before the checker is launched. **Attempting to run any malicious code on the checker is equivalent to immediate disqualification of the team and removal from the event.**
- It is important that your implementation works as quickly as possible. If the model exceeds certain time limit, you will receive a notification about it in the feedback after sending the solution, but the model will be accepted. During the game with other players, after exceeding the time limit, the game will consider that the model did not make any move in the given turn.
- If you get response code 429, meaning that you exceeded the limit of submissions for this period of time, don't try to force your way into the server. We don't like DoS attacks :)
- Remember, that the environment we gave you has the same configuration, as the environment during tests, so you shouldn't change anything there (you can, but on your own responsibility)

Scoreboard:

The board with each team's current scores is available here: http://138.68.67.242:3000/