Welcome to Debris Defense!

Hello! We are so excited to see you at Algorithms with a Purpose 2024, hosted by ACM@CMU. This document contains the game rules, a reference manual for your code, and a list of the in-game constants.

If you have any questions feel free to ask one of our AWAP staff at Office Hours or on Discord.

Getting Started

>> Important Links

- Dashboard: dashboard.awap.acmatcmu.com
- Discord: https://discord.com/invite/2PvTFS4R
    - If you weren't automatically assigned the @AWAP 2024 role, it can be claimed from #role-select. Scroll down or look in the channel list if you don't see it! We'll post an @everyone when the event starts as well.
    - We also encourage you to drop a cool message in #intro and follow your fellow community members :)
    - For this event, please join #awap-2024 for announcements, #crew-log for public questions and memes, and #support for anything individual!

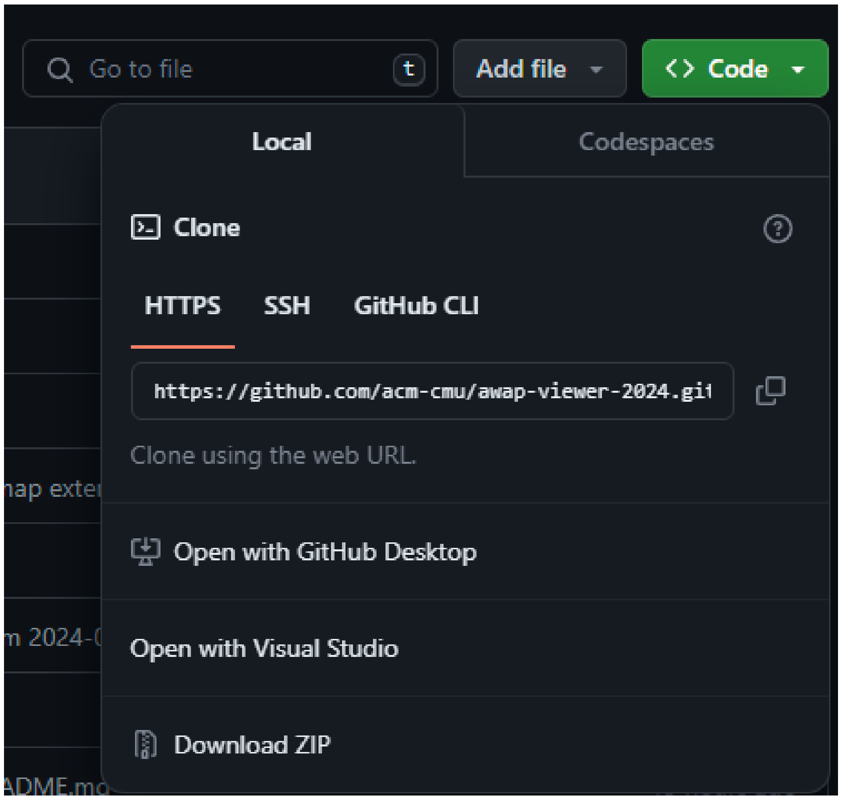
Development & Submission Flow

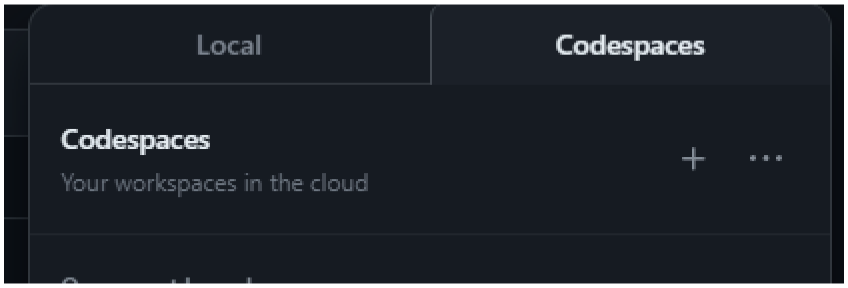
>> Local Development

1. Clone game engine at https://github.com/acm-cmu/awap-engine-2024-public
2. Write your bot and add it to the bots/ directory.
3. Put any maps you'd like to test with in the maps/ directory.
4. Specify players & maps in config.json then run 'python3 run_game.py -c config.json'. You may use the '--render' flag if you would like to see the game play out live.
5. After the replay file archive is created, you can view it again in your terminal by running 'python replay_game.py <file>.awap24r.gz'
    a. A command-line view is available as well by extracting the file and running 'python replay_game_cli.py <file>.awap24r', which may be more compatible with browser-based editors.

>> Using A Codespace

1. Upgrade your GitHub account for free with the Student Developer Pack! This will give your account Pro status and offers many other benefits. Sign up with your andrew.cmu.edu email after attaching it to your profile education.github.com/pack
2. Visit the repo you're interested in, like the game engine or viewer.
3. Click Code, and then switch from 'Local' to 'Codespace'





4. Start a new Codespace! This is a virtual machine, with a VS Code interface. You can customize these with any extensions you're used to, and push commits to your own fork if you'd like. After it starts up, follow the same instructions as local development (usually `npm install` and then `npm start` in the terminal interface), adding extensions like Python if necessary.

5. Just like developing locally, please `git pull` periodically so that we can deploy hotfixes if necessary. We'll let you know if so!
6. Similar to ACM@CMU HackCMU this past Fall, we'd recommend trying these out as this way all of our attendees can have the same development environment which makes debugging much easier.

>> Submission to Competition

1. Log into the dashboard at dashboard.awap.acmatcmu.com with the account username and password emailed to you
2. Upload your bot
    a. Request unranked scrimmages with other competitors or raffle bot
3. Review your match results
    a. Check match history
    b. Check leaderboard
    c. Download replay against other competitors

Game Overview

>> Background and Objective

Two competing countries are racing to clean up space debris! In a tower defense style game, build towers to protect your base from debris and send debris to attack your opponent!

Debris travels along a predefined path that is fixed for each map. Each piece of debris has a fixed amount of health and cooldown, which determine the amount of damage it can take and its speed, respectively. Debris can come from either natural sources or be sent by the opponent. Natural debris gets gradually stronger over time based on a predefined schedule.

Both teams have a base at the end of the debris path, with a certain amount of health. When debris reaches the end, it deals damage to the base equal to its maximum health. The win condition is to survive longer than your opponent. If both teams die on the same turn, then whoever has the most energy net worth wins.

>> Tower Types

There are 4 types of towers: Gunships, Bombers, Solar Farms, and Reinforcers. Each tower has a cooldown, which represents the rate at which it can take actions. Right after a tower takes an action, its cooldown is set to the tower type's max cooldown. Every turn, the cooldown is decremented by 1, and the tower can take another action once it reaches 0 cooldown.

A tower's range describes a circular area with the tower at the center. The range value is the circle's radius squared.

Solar Farms and Reinforcers will act automatically once they are built; you should not write code to control them.

Towers may be sold for 80% of the price at which they were purchased.

Gunship

Gunships deal a large amount of damage to a single debris in its range.

- Cost: 1000
- Range: 60
- Cooldown: 20
- Damage: 25

Bomber

Bombers deal a small amount of damage to all debris in a small area surrounding it.

- Cost: 1750
- Range: 10
- Cooldown: 15
- Damage: 6

Solar Farm

Solar Farms generate energy.

- Cost: 2000
- Cooldown: 10
- Energy Production: 20

Reinforcer

Reinforcers accelerate the cooldown decay of nearby towers by a certain percentage. This effect stacks with more reinforcers.

- Cost: 3000
- Range: 5

- Cooldown decay multiplier: 1.2

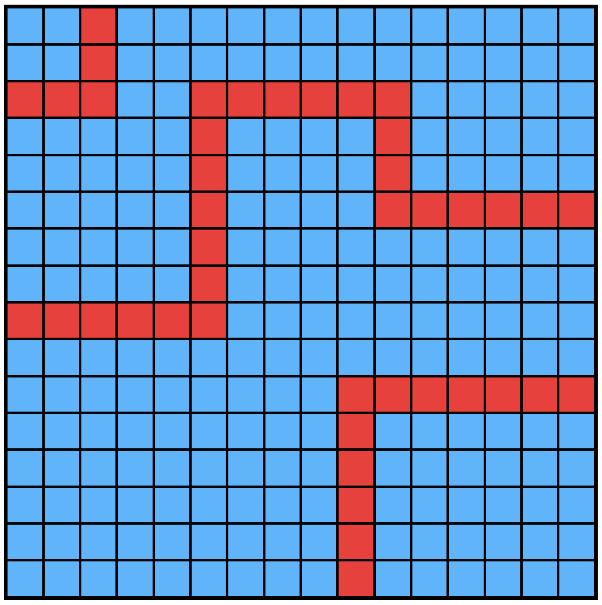
>> Debris

Each debris is parameterized by a cooldown and a health. Cooldown works similarly to towers: after each move, the cooldown gets reset to its maximum value, and will decrease by 1 turn. This decay is not affected by Reinforcers.

Debris can be sent in two ways: naturally by the game, or by your opponent. When sending debris, you can specify any combination of health and cooldown. Here is a table showing the energy costs of various combinations: https://docs.google.com/spreadsheets/d/1jjhh5ejhcRFe78dYi77_F14Xa4fa9bxofifl-o68DSM/edit?usp=sharing

>> Map Overview

The world is a grid with width and height dimensions that may range from 16 to 48. Each map has a single fixed debris path that starts and ends at the edges of the map (debris will not start spawning in the middle). Paths may be discontinuous at the edges of the map, but are continuous otherwise.


An example map with a discontinuous path.

There are three types of tiles: “space”, where you can place towers, “path”, which debris move along, and “asteroid”, where neither towers nor debris can occupy.

>> Gameplay Details

Each team begins with 1500 energy and a base health of 2500. You also get a full view of the debris path and the ability to place ships anywhere that isn't on top of the asteroids or on the debris path itself.

Player code must run within a certain time limit. At the start of each game, players are allocated a 10s time bank. They also get an additional 0.01s every turn. If this time limit is exceeded at any point, the player automatically loses the game.

Each turn, the following happens (in order):

- Debris is spawned (both natural and sent by opponent in previous turn)
- Each player gets passive income of 10 energy
- All debris/tower cooldowns are decayed
- Debris with 0 cooldown moves forward 1 tile
- Check if the game has ended (a player has lost all their health)
- Add time to each player's time bank
- Each player gets income from Solar Farms that have 0 cooldown
- Each player's code is called, allowing them to build towers and take actions

>> Tie Breaks

If one player's base loses all health, they lose. In the case that both players make it to the end, the tie breaking order is as follows:

1. Net worth (energy in bank + sum of the costs of all towers)
2. Coin flip

>> Allowed Packages

- Python standard library
- Numpy
- Scipy
- Anything already imported by the game engine

All packages not listed are not allowed - please let us know if there are packages you want to use and we'll consider them; we're flexible on this.

API Documentation

This is an exhaustive list of all the functions or methods that players are allowed to access. If you have any remaining questions about the functionality, please reach out to us at in person office hours!

Note: Please do not attempt to exploit the game engine by accessing the internal state. Attempting to do so may result in disqualification.

Starting Template

Competitors will create a child class of Player that extends the play_turn function. The child class has to be named ‘BotPlayer’. Below is the starter code you should copy:

```
from src.player import Player
from src.map import Map
from src.robot_controller import RobotController
from src.game_constants import TowerType, Team, Tile, GameConstants,
SnipePriority, get_debris_schedule
from src.debris import Debris
from src.tower import Tower

class BotPlayer(Player):
    def __init__(self, map: Map):
        pass

    def play_turn(self, rc: RobotController):
        return
```

RobotController

These are all the member functions of RobotController. These are used to control your player and obtain information about the game state.

`get_ally_team() -> Team`

Returns the team you are on.

`get_enemy_team() -> Team`

Returns the enemy team.

`get_map() -> Map`

Returns the map.

`get_towers(team: Team) -> List[Tower]`

Returns the list of towers owned by the specified team.

`get_debris(team: Team) -> List[Debris]`

Returns the list of debris attacking the specified team.

`sense_debris_within_radius_squared(team: Team, x: int, y: int, r2: int) -> List[Debris]`

Returns a list of all debris whose squared distance to (x, y) is at most r2. Internally, this loops through all the debris.

`sense_debris_in_range_of_tower(team: Team, tower_id: int) -> List[Debris]`

Returns a list of all debris in the range of the specified tower. Internally, this loops through all the debris. Throws an exception if the specified tower does not exist.

`sense_towers_within_radius_squared(team: Team, x: int, y: int, r2: int) -> List[Tower]`

Returns a list of all towers whose squared distance to (x, y) is at most r2. Internally, this loops through all the towers.

`sense_towers_in_range_of_tower(team: Team, tower_id: int) -> List[Tower]`

Returns a list of all towers in the range of the specified tower. Internally, this loops through all the towers. Throws an exception if the specified tower does not exist.

`get_balance(team: Team) -> int`

Returns the amount of energy the specified team has.

`get_turn() -> int`

Returns the current turn number.

`get_debris_cost(cooldown: int, health: int) -> int`

Returns the energy cost of sending a debris with the specified cooldown and health values.

`can_send_debris(cooldown: int, health: int) -> bool`

Returns a boolean indicating whether or not you can send debris with this cooldown and health. Checks for the following conditions:

- You have not yet sent debris this turn.
- You have enough energy to pay for this.
- “cooldown” is a positive integer

`send_debris(cooldown: int, health: int)`

Send debris with the specified cooldown and health to the enemy side. Raises an exception if can_send_debris() returns False.

`is_placeable(team: Team, x: int, y: int) -> bool`

Checks if the tile is 1) a SPACE tile and 2) unoccupied by another tower.

`can_build_tower(tower_type: TowerType, x: int, y: int) -> bool`

Returns a boolean indicating whether or not you can build the specified tower at (x, y). Checks for the following conditions:

- You have enough energy to pay for this.
- (x, y) is in bounds and is an unoccupied space tile.

`build_tower(tower_type: TowerType, x: int, y: int)`

Builds the specified tower at the specified location. Raises an exception if can_build_tower() returns False.

`sell_tower(tower_id: int)`

Sells the specified tower, removing it from the map and providing an energy refund. Throws an exception if the tower_id does not exist.

`get_time_remaining_at_start_of_turn(team: Team) -> float`

Returns the time (in seconds) the specified team has in their time pool. **Does not include time spent so far on the current turn.**

`can_snipe(tower_id: int, debris_id: int) -> bool`

Checks whether the specified gunship can snipe the target debris:

- The gunship's cooldown must be 0.
- The debris must be in range.

If the tower_id or debris_id do not exist, or if tower_id is not a gunship, an exception will occur.

`snipe(tower_id: int, debris_id: int)`

Makes the specified gunship snipe the target debris. Raises an exception if can_snipe() returns False.

`auto_snipe(tower_id: int, priority: SnipePriority)`

An all-in-one gunship attack function. Checks if the tower is a gunship, if it's ready to shoot, and if there is any debris in range. If all these conditions are met, the tower picks the debris with the highest "priority" and shoots it. Otherwise, the tower does nothing. This field has several presets:

- SnipePriority.FIRST: targets the debris that has made the most progress.
- SnipePriority.LAST: targets the debris that has made the least progress.
- SnipePriority.CLOSE: targets the closest debris to the tower.
- SnipePriority.WEAK: targets the debris with the least max health.
- SnipePriority.STRONG: targets the debris with the most max health.

`can_bomb(tower_id: int)`

Checks whether the specified bomber can bomb:

- The bomber's cooldown must be 0.

If the tower_id does not exist or is not a bomber, an exception will occur.

`bomb(tower_id: int)`

Make the specified bomber do a bomb attack. Raises an exceptoin if can_bomb() returns False.

`auto_bomb(tower_id: int)`

An all-in-one bomber attack function. Checks if the tower is a bomber, if it's ready to shoot, and if there is any debris in range. If all these conditions are met, the bomber does an attack. Otherwise, the tower does nothing.

Map

`width: int`

The width of the map in tiles.

`height: int`

The height of the map in tiles.

`path: list[tuple]`

A list [(x0, y0), (x1, y1), ...] containing the coordinates of the tiles that debris travels along. The tiles are ordered from first to last.

`tiles: list[list[Tile]]`

A 2d array with dimensions (width, height), containing all the tiles of the map. Each tile is either a path, space, or asteroid. Index by x first, then y.

`is_in_bounds(x: int, y: int) -> bool`

Checks whether the specified coordinates are within the map bounds.

`is_space(x: int, y: int) -> bool`

Checks whether the specified coordinates is 1) in bounds and 2) a space tile.

`is_asteroid(x: int, y: int) -> bool`

Checks whether the specified coordinates is 1) in bounds and 2) an asteroid tile.

`is_path(x: int, y: int) -> bool`

Checks whether the specified coordinates is 1) in bounds and 2) a path tile.

Game Constants

`TowerType`

An enum with the following possible values: SOLAR_FARM, GUNSHIP, BOMBER, or REINFORCER. Each has 4 properties:

- cost: The energy cost to build this tower.
- range: The squared distance representing the tower's range; meaningless for solar farms.
- cooldown: The cooldown the tower is reset to after taking an action; meaningless for reinforcers.
- damage: The amount of damage the tower does to debris; meaningless for solar farms and reinforcers.

`Team`

An enum, either BLUE or RED.

`Tile`

An enum, either PATH, SPACE, or ASTEROID.

`SnipePriority`

An enum, either FIRST, LAST, CLOSE, WEAK, or STRONG.

`GameConstants`

A class containing the following fields:

- STARTING_HEALTH: The starting health of each teams' home bases.
- STARTING_BALANCE: The initial amount of energy each team starts with.
- PASSIVE_INCOME: The amount of energy given to each team per turn.
- FARM_INCOME: The amount of energy a solar farm generates per move.
- REFUND_RATIO: The proportion of tower cost that is refunded when sold.
- REINFORCER_COOLDOWN_MULTIPLIER: The multiplicative factor each reinforcer applies to the cooldown decay of nearby towers.
- INITIAL_TIME_POOL: Initial computation time (in seconds) given to each team at the start of the game.
- ADDITIONAL_TIME_PER_TURN: Additional computation time (in seconds) given to each team every turn.

`get_debris_schedule(turn_num: int) -> tuple`

Returns the (cooldown, health) of the debris that will naturally be sent on the given turn.

Debris

`id: int`

The unique integer identifier for this debris.

`team: Team`

Which team this debris is attacking.

`progress: int`

The index of the map's path that this debris is currently at.

`x: int`

The x-coordinate.

`y: int`

The y-coordinate.

`total_cooldown: int`

Represents the speed. The cooldown value is reset to total_cooldown every time after the debris moves.

`current_cooldown: int`

The number of turns before this debris moves again.

`total_health: int`

The initial health of this debris. When this debris reaches the end of the path, it will deal this amount of damage to the team's home base.

`health: int`

The current health of this debris.

`sent_by_opponent: bool`

Whether or not this debris was sent by the opposing team.

Tower

`id: int`

The unique integer identifier for this tower.

`team: Team`

The team this tower belongs to.

`type: TowerType`

The type of tower this is.

`x: int`

The x-coordinate of this tower.

`y: int`

The y-coordinate of this tower.

`cooldown: float`

The current cooldown of this tower.