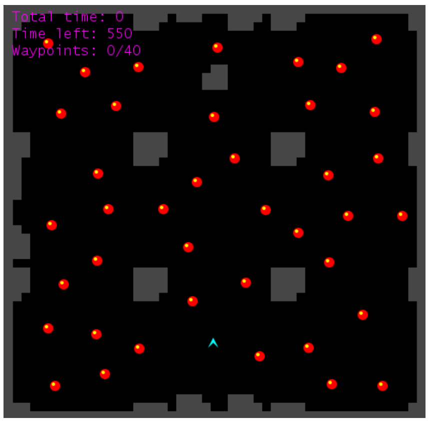
Rules of the CIG PTSP 2012 Competition

Goal of the PTSP

The objective of the PTSP is to visit the maximum number of waypoints of the map in the minimum number of time steps. The map takes the form of a two-dimensional board, where a determind number of waypoints are scattered around and multiple obstacles are present. The following image is an example of a map with obstacles and waypoints.



Evaluation

The interface shows two time values, "total time" and "time left". The first one shows the time spent since the beginning of the execution and it is used to score the controller. The second, however, states the time left to visit another waypoint, and it is decreased by 1 unit every time step. When the next waypoint is visited, this time is set back to its original value. The game can end in two different ways: either all waypoints have been visited, or the time left counter goes down to 0. The score of the controller in a single map will be the number of waypoints visited and the time spent. Several executions will be made in the same map, so the final result will be the average of waypoints visited and time spent.

Multiple competitors will be organized by their results according, first, to the number of waypoints visited and, secondly, to the time spent. Furthermore, as the competition will take place on more than one map, the final winner will be the one that performs best across multiple maps. Hence, each map will have a ranking of controllers, and point will be given to then according to this ranking. The points will follow the following scheme:

- 25 points for the first classified of the map.
- 18 points for the second.
- 15 points for the third.
- 12 points for the fourth.
- 10 points for the fifth.
- 8 points for the sixth.
- 6 points for the seventh.
- 4 points for the eighth.
- 2 points for the ninth.
- 1 point for the tenth.

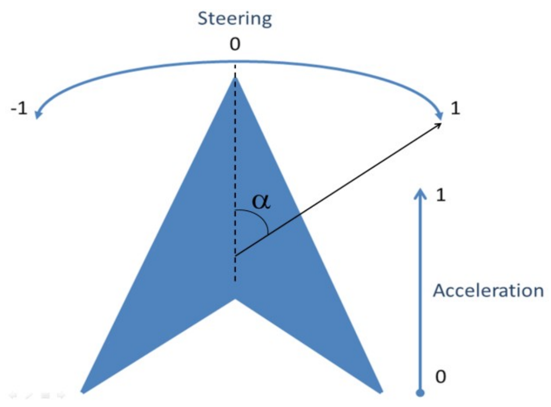
Here is an example of a possible ranking with several controllers in one of the maps:

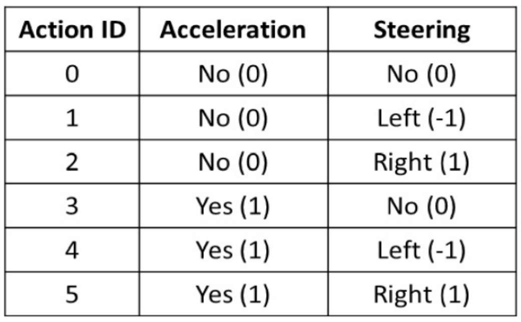
Participant	Waypoints visited	Time spent	Points awarded
BattlestarShip	40	9241	25
Sovereign	37.8	8998	18
UltraShip	35	8100	15
Prometheus	21	8500	12

In the case of a draw in the final score, the best controller will be considered to be the one with more first positions. If the draw persists, the number of second, third, etc. positions will be taken into account.

Game mechanics

There are two different inputs applicable to move the ship, thrust and rotation, adding up to a total of six different actions that govern the ship. The former can be seen as a boolean input (either the ship accelerates or not), while the latter is an integer value to indicate rotation to the left (-1), to the right (1) or no rotation at all (0).





Action ID	Acceleration	Steering
0	No (0)	No (0)
1	No (0)	Left (-1)
2	No (0)	Right (1)
3	Yes (1)	No (0)
4	Yes (1)	Left (-1)
5	Yes (1)	Right (1)

All actions are as forces applied to the ship that update its position, orientation and velocity at each step. The following equations show how these vectors are updated: orientation (equation 1), velocity (equation 2) and position (equation 3).

$$d_{t+1} = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} d_t \tag{1}$$

$$v_{t+1} = (v_t + (d_{t+1} T_t K))L \tag{2}$$

$$p_{t+1} = p_t + v_{t+1} \tag{3}$$

where d_t, v_t and p_t are the vectors that represent the orientation (direction), velocity and position of the ship at time t; is the rotation angle and L represents the friction factor, used to reduce the speed of the ship at every time step, K is an acceleration constant and the value of T_t depends on the force applied in the step t: 1 if the action involves acceleration, and 0 otherwise. Although there is no damage to the ship, the ship’s velocity (and hence, its position) can be affected if it collides with one of these obstacles. In that case, v_{t+1} is modified accordingly, affecting direction, applying an elastic collision, and speed, multiplying v_{t+1} by a collision factor in order to reduce its speed significantly.

Game maps

The maps that can be used by this game must be contained in ASCII files. They have the following format: the first two lines specify the height and width of the map. After this, the keyword *map* indicates the start of the map in the field. Finally, the following lines compose the map itself. Each position in the map is specified by an ASCII character, and represents a position in the map. Their different values and meaning can be seen next:

- ’@’ and ’T’: walls or obstacles.
- ’C’: Waypoint.
- ’S’: Starting point for the ship.
- ’.’: Empty space.

This format is based on the symbols used by Nathan Sturtevant, from games such as Warcraft, Starcraft or Baldur’s gate. They have been used by many researchers in the literature - http://movingai.com/benchmarks.

The *starter kit* will include 10 maps so the competitors can train their entries. Additionally, this webpage will host a **training submission server**, where the participants will be able to score their controllers. This server, available before the end of the competition, will have 20 unknown maps, which won't be made public to the participants. Furthermore, during the competition and until the submission deadline, the set of 20 maps will be slightly modified (meaning, some of them will be changed for others) periodically, in order to avoid overfitting to the maps provided. The final evaluation of the competition will take place with a different set of 20 maps (10 of them completely new, the other 10 taken randomly from the ones in the training submission server).

The maps contain 30, 40 or 50 waypoints.

Preliminary submission server

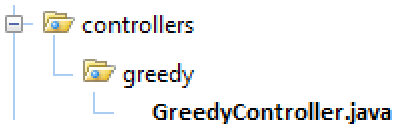
A preliminary submission server will be running at www.ptsp-game.net during the competition allowing participants to submit their controllers to be evaluated. This evaluation will take place in 20 maps that are unknown by the participants, and they will be changed periodically to avoid overfitting. Furthermore, a ranking of these submissions will be kept at the webpage. This way, the participant will be able to know how the controller is performing in unknown maps in relation with other participants.

It is important to notice that this ranking is meant to be for guidance, and it should be taken as an approximation for the final results. It is completely independent from the final rankings of the competition, that will be calculated after the submission deadline, in a different set of maps.

Please, take into account the following aspects before submitting your controller:

- The server will execute your controller in the 20 maps of the preliminary submission server, and also in the 10 maps of the starter kit. Furthermore, each map is executed 5 times, what adds up to 150 executions. This process can take its time, especially if you are making use of much of the 40ms given per action. To quick the execution, please try to keep to a minimum not needed processing, as well as not printing out debug information to console.

- The file you submit must be a ZIP file containing all folders and Java files needed for your controller. You do not need to include the framework code in you submission. Your code will be compiled and executed on the server. In order to do that, the name of the file must be the name of your controller, including packages.
- You may write and read files from your controller's directory. To avoid problems in the submission server, you must use relative paths starting from "src". For instance, if you want to read a file that is in the directory greedy/, you must specify the file like this: "src/controllers/greedy/myFile.txt"
- The server will execute your controller and it will report you back possible errors or exceptions. It is important that your controller does not print to the error console (System.err) at all, or the process that runs your controller will treat that as an error and hence the execution will not be valid.
- Example: if you want to submit a controller like controllers.greedy.GreedyController, the name of the file should be controllers.greedy.GreedyController.zip. Furthermore, the contents of the file must be like the following:



Getting started

The code

The code provided to create controllers is divided into Java packages. These include:

- *controller*: The participant must create the controller in a sub-package of this package, as the sample controllers random.RandomController, greedy.GreedyController, lineofsight.LineOfSight and WoxController.WoxController are.
- *framework*. This packages contains all the code for the game.
  - *core*: Core code of the game.
  - *graph*: Code for path finding.
  - *utils*: Includes useful classes.
  - *Classes* ExecSync, ExecReplay, ExecFromData: Executes a controller in different execution modes.

Execution

The execution is very simple. There are several main methods provided, located at the following classes with different execution modes. None of them need any arguments.

- **ExecSync.java**: To execute one or several maps, with and without visuals.
  - *Mode 1*: To play the game with the key controller.
  - *Mode 2*: Runs once in a map, supplying frame rate. This is the competition execution mode.
  - *Mode 3*: Executes several games, *N* times each, getting a summary of results at the end.
- **ExecReplay.java**: To execute replays of old games.
- **ExecFromData.java**: This class contains special execution modes, as execution in maps created from data or providing the controller already created. These modes can be used to execute large amounts of consecutive runs, which can be useful for reinforcement learning or evolutionary algorithms.
  - *Mode 1*: Run from data: Create the map from data structures and execute once on it.
  - *Mode 2*: Execute from data, providing also a controller already created.
  - *Mode 3*: Execute from file, providing a controller already created. It can execute in several maps.
  - *Mode 4:* Creates an instance of Game to play on it.

Game flow

In order to create a controller for this game, the participant must write a Java class that extends framework.core.Controller. Two methods need to be implemented in this class:

- A public constructor that receives a game copy (class **Game**) and the time due (**long**).
- A function called **getAction()**, that returns and **int** and receives a game copy (**Game** class, again) and a long variable that indicates where the controller is due to respond with an action. This function will be called every execution cycle to retrieve an action from the controller. This action must be one of the available ones:
    - framework.core.Controller.ACTION_NO_FRONT: No rotation and no acceleration. This is also the action applied if no response is received within the time given.
    - framework.core.Controller.ACTION_NO_LEFT: Left rotation but no acceleration.
    - framework.core.Controller.ACTION_NO_RIGHT: Right rotation but no acceleration.
    - framework.core.Controller.ACTION_THR_FRONT: No rotation, only forward acceleration.
    - framework.core.Controller.ACTION_THR_LEFT: Left rotation and acceleration.
    - framework.core.Controller.ACTION_THR_RIGHT: Right rotation and acceleration.

The main class creates the game and the controller, using the appropriate constructor of this class, expecting a response in a specific time. The main class executes the game loop, which calls the controller's method **getAction()** supplying a copy of the game and the time due to receive the action. The time taken by the controller to reply is monitorised, and three different scenarios can happen:

- If the controller responds in less than PTSPConstants.ACTION_TIME_MS (40ms), the action indicated is executed (this is the normal case).

- If the controller responds spending more than PTSPConstants.ACTION_TIME_MS but less than PTSPConstants.TIME_ACTION_DISQ, then the default action (0: No acceleration and no rotation) is applied.

- If the controller responds in more than PTSPConstants.TIME_ACTION_DISQ, the game is ended and the controller is disqualified for this match. The score of the game would be 0 waypoints visited and PTSPConstants.getStepsPerWaypoints() time steps

The next step consists of the update of all entities in the game, including the ship that executes the action retrieved. It is important to notice that the controller has access to a copy of the game, being able to access the state of the game, ship, waypoints, etc., including also the possibility to simulate moves before supplying a final action.

This process is repeated until the game ends, which may happen because all waypoints have been visited or because the time runs out.

Programming specifications

- The controllers must be programmed in Java, as that is the language of the benchmark.
- The ship must reach another waypoint before a time limit runs out. This time depends on the number of waypoints of the map:
    - 30 waypoints: 700 time steps.
    - 40 waypoints: 550 time steps.
    - 50 waypoints: 400 time steps.
- Ship physics:
    - Rotation step is fixed to PI/60 radians, for right and left steering actions resp.
    - The friction factor (L) is fixed to 0.99.
    - The acceleration constant (K) is fixed to 0.025.
    - The collision factor that reduces the speed of the ship, to be used when it collides with an obstacle, is 0.25.
- Multi-threading is not allowed.
- Reading from files is allowed, although writing to them is only allowed if it is done in the controller's own directory. At the moment of the final evaluation, all controllers will be executed with the files included in the Zip file submitted (so, any auxiliary file will be reset to the version received in the submission).
- There are two time limits that must be respected, at the controller initialization and update. The first one is the maximum amount of time allowed for the constructor of the controller, and it is set to a value depending on the number of waypoints of the map:
    - 30 waypoints: 3000ms.
    - 40 waypoints: 4000 ms.
    - 50 waypoints: 5000 ms.

  The second is the time allowed for the controller to provide an action to perform at every execution step, established at 40ms. If the first time limit is not respected, the controller will not be executed on that map. Should the controller spend more time than allowed during the update call, the action performed by the controller in this step will be action 0 (i.e. no thrust, no rotation). However, if the controller spends more than PTSPConstants.TIME_ACTION_DISQ, set to 80ms, it gets disqualified from this match (getting 0 wp and getStepsPerWaypoints() time steps.)
- The process of the controller is also memory limited, being the controller allowed up to 256MB of usage. Not respecting this limitation will produce a result of 0 points in that map.
- The *starter kit* provides code for pathfinding that can be used for any controller. This code creates a grid graph on the navigable parts of the map, using eight-way connectivity between the nodes (i.e, each node is connected to its eight neighbour nodes, if present.). The execution of the code devoted to this end must be triggered by the controller, so that, should the participant decide not to use it, they will not be penalised by the time/storage consumption that this process involves.
- Copies of the game instance are provided to the controllers in order to have full access to the game during execution. Any attempt to modify the real state of the game, other than the

required actions to control the ship, will end with the disqualification of the controller from the whole competition.

Sample controllers

Two sample controllers are included in the *starter kit:* RandomController and GreedyController.

The first one, RandomController, is one of the most basic controllers that can be made. It just returns a random action every cycle:

```
public class RandomController extends Controller
{
    Random m_rnd;

    public RandomController(Game a_gameCopy, long a_timeDue)
    {
        m_rnd = new Random();
    }

    /**
     * This function is called every execution step to get the action to execute.
     * @param a_gameCopy Copy of the current game state.
     * @param a_timeDue The time the next move is due
     * @return the integer identifier of the action to execute (see interface framework.core.Controller for definitions)
     */
    public int getAction(Game a_gameCopy, long a_timeDue)
    {
        return m_rnd.nextInt(Action.NUM_ACTIONS);
    }
}
```

The second sample, GreedyController, finds the action that takes the ship closer to the closest waypoint. It does this by using the path-finding code, in the package *framework.graph.* The code of this controller can be found at controller.greedy.GreedyController.java