LaserTag

An Agent-Based Game Simulation for Testing Intelligent and Goal-Oriented Behavior

June 22, 2021
Version 1.0

Contents

1 Introduction

The LaserTag Framework provides an agent-based game simulation that is inspired by the real-world recreational shooting sport known as laser tag. LaserTag is written in MARS C#. A number of methods are provided to serve as an interface between agents and the game world (Battleground) and game mechanisms. These methods should be used to play the game as they allow agents to manage their states, move through the Battleground, and interact with other agents upon encountering them.

This documentation will be updated over the next few weeks. Stay tuned for announcements.

2 Objective

A team is made up of three agents with the same AI. There are three to four teams per match that compete against each other in a team deathmath game mode. The goal of the game is to score points, and the team with the highest cumulative score at the end of the game wins the match. **Note:** The goal is not to be the last man/team standing, but to score the highest cumulative number of points across all team members.

3 Project: Setup and Structure

3.1 Project Setup

The project is available in the GitLab repository MARS Laser Tag Game. The directory **LaserTagBox** contains the game. To use LaserTag, a working installation of JetBrains Rider the MARS C# plug-in is needed. In Rider, open **LaserTagBox** to access the game.

3.2 Project Structure

The following diagram illustrates the project structure along with properties and methods that are relevant for the AI implementation.

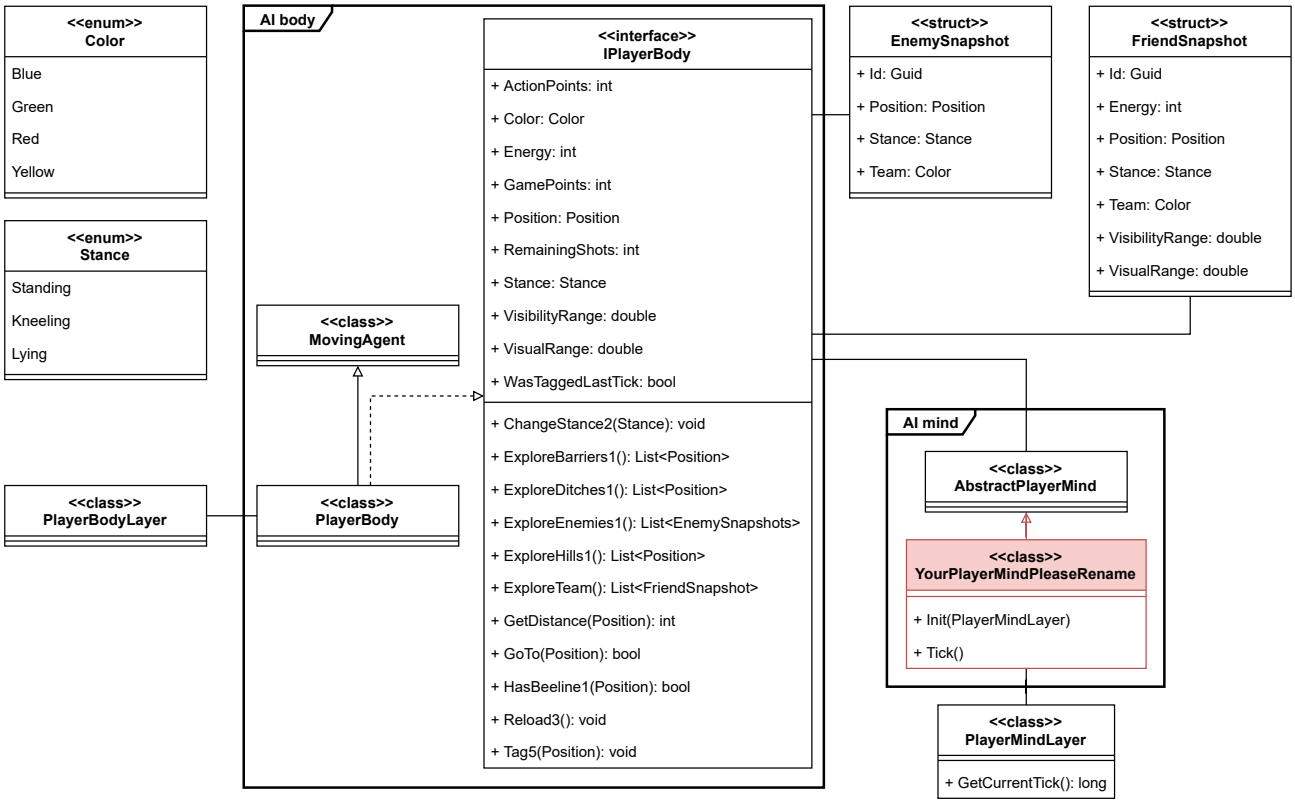

Figure 1: Diagram illustrating structure of LaserTag in MARS C#

The implementation of your AI occurs in a class `YourPlayerMindPleaseRename`, labeled in red in Fig. 1, which inherits from the class `AbstractPlayerMind`. As suggested by the default name, you are welcome to rename the class as you wish. Your AI has a reference to the `PlayerMindLayer`, from which it may obtain the current simulation tick (`GetCurrentTick()`), if desired.

The agents' AI has a mental and a physical representation – mind and body, respectively. The mind controls the body. Through `AbstractPlayerMind`, the mind obtains a reference to the interface `IPlayerBody` which has several properties and methods that make up the agent's physical representation. The implementation of these properties and methods is found in the class `PlayerBody`. Your AI's mind can interact with the interface to guide your AI's body and shape its physical behavior. The class `MovingAgent` contains auxiliary properties and methods that the body requires to execute some of its functionalities.

More details regarding the enums, structs, and other parts of the model can be found in subsequent section of the documentation.

4 Simulation: Setup and Execution

The game is designed to be played with three or four teams. Use the `config.json` in the root directory to configure the simulation. This configuration file requires resource files that are contained in the directory **Resources** – in particular, a map file and an agent initialization file. The default map is called `Battleground.csv` and encodes a 50×50 grid cell structure on which the agents can move. The default agent initialization file is called `player_positions.csv` and contains the agents' spawn positions on the map.

Simulations can be run via the play button of the IDE.

5 Visualization: Analyzing Game Outcomes

A tool with instructions for visualizing simulation results is available in the repository in the directory **Analysis**. This tool can help you analyze and improve your agents' behavior during the development process. In order for the tool to work, a map file (`map.csv`) is required in the directory **Resources/** and a simulation output file (`PlayerBody.csv`) is required in the directory **bin/Debug/netcoreapp3.1/**. Once those files are in place, simply double-click `vis.py` to start the visualizaton tool.

6 Rules

6.1 Game Logic

Below is a list of some of the most important parts of the game's logic:

- If an agent's `Energy` is equal to or below 0, then the agent is taken off the battleground and does not respawn for the rest of the game. The agent's points, however, are maintained and added to the cumulative score of the team at the end of the match.
- An agent's `Energy` regenerates over time. At the end of each tick, an agent's `Energy` is increased by 1.

6.2 Constraints for AI Developers

In order to play the game as intended, please adhere to the following rules when implementing your AI:

1. Only interact with the interface `IPlayerBody` to access the agent's physical representation.
2. If interacting with the `PlayerMindLayer`, only invoke the `GetCurrentTick()` method.
3. Loops that are known not to terminate after a reasonable time (example: `while(true)`) are not allowed.
4. `PropertyDescription` tags (for loading external information into your agents at runtime) are not allowed.

7 Agent Properties and Methods

The `IPlayerBody` contains a set of properties and methods that described the agent and define its behavioral capabilities.

7.1 Properties

7.1.1 General properties

- `ActionPoints`: an integer that specifies the number of points the agent has in order to complete actions during the current tick. Each action costs a specific number of `ActionPoints` (see 7.2 for more details). At the end of each tick, `ActionPoints` is reset to 10.
- `Color`: the agent's color, indicating the agent's team.
- `Energy`: the agent's maximum energy level is 100 and decreases if the agent gets tagged by an opponent. If the energy level is less than or equal to zero, the agent is positioned at its initial spawn position with `energy = 100`.
- `GamePoints`: the agent's score, which is increased by tagging opponents. Each tag increases the score by 10 points. If a tag causes the opponent's `energy` to be less than or equal to 0, the tagged agent loses 10 points and an additional 10 points are awarded to the tagger as a bonus.

7.1.2 Movement properties

- `Position`: specifies the agent's current position on the map as a pair of xy-coordinates
- `Stance`: an enum that specifies the agent's current stance. An agent can assume three stances: `Standing`, `Kneeling`, and `Lying`. Each stance affects the property `VisualRange`, `VisibilityRange`, and speed at which the agent can move.

7.1.3 Exploration properties

- `VisualRange`: an integer that is set based on the value of `Stance` and that specifies the agent's current range of sight. The mapping from `Stance` to `VisualRange` is as follows:
  - `Standing` $\rightarrow 10$
  - `Kneeling` $\rightarrow 8$
  - `Lying` $\rightarrow 5$
- `VisibilityRange`: an integer that is set based on the value of `Stance` and that specifies the maximum distance from which the agent can currently be seen by other agents. The mapping from `Stance` to `VisibilityRange` is as follows:
  - `Standing` $\rightarrow 10$
  - `Kneeling` $\rightarrow 8$
  - `Lying` $\rightarrow 5$

7.1.4 Tagging properties

- `RemainingShots`: an integer that specifies the agent's currently available opportunities to tag an opponent. If the agent's `RemainingShots === 0`, then a reload process needs to be initiated.
- `WasTaggedLastTick`: a boolean that specifies if the agent was tagged during the previous tick.

These are the properties that may be used to guide the agents through the game. For a full list of properties, please see the source code.

7.2 Methods

Below are the methods that an agent's mind may call to guide its body. The digit at the end of the method name indicates the number of `ActionPoints` required to execute the method. Methods with no digit at the end cost zero `ActionPoints`.

7.2.1 Movement Methods

- `ChangeStance2(Stance)`: this method takes a `Stance` and allows the calling agent to change between three possible stances: `Standing`, `Kneeling`, and `Lying`. Stance changes affect the values of the agent's `VisualRange`, `VisibilityRange`, and movement speed.
- `GoTo(Position) : bool`: this is the main method used for path-finding, movement, and path readjustment. When an agent invokes the method, the method devises a path from the agent's current position to the destination specified by `Position`. Each subsequent invocation of `GoTo(Position)` with the same destination will, if possible, move the agent one step closer to the destination until the destination is reached.

  In order for the agent to change his path before having reached his current destination, `GoTo(Position)` must be called with a different destination. For example, if the agent is currently moving towards the `Position` (a, b), this movement process can be interrupted and replaced by a new movement process by calling `goto((c, d))`, where `c != a` or `d != b`.

  The method returns `true` when a move was made and `false` when, for any reason, a move was not made. **Note:** if `GoTo(Position)` is called with a destination that refers to a grid cell that is inaccessible (because there is a `Barrier` on it), then the no path is calculated, no movement is initiated towards the specified destination.

  For more information on `goTo(Position)`, please see Sec. 8.2.2.

7.2.2 Exploration methods

- `ExploreBarriers1() : List<Position>`: returns a list of positions of `Barrier` objects that are in the caller's `VisualRange` and which the agent can see (`HasBeeline == true`).
- `ExploreDitches1() : List<Position>`: returns a list of positions of `Ditch` objects that are in the caller's `VisualRange` and which the agent can see (`HasBeeline == true`).
- `ExploreEnemies1(): List<EnemySnapshot>`: this method performs an exploration of opponents in the agent's field of vision and returns a list of `EnemySnapshot` structs, offering limited information about the identified opponents.

- `ExploreHills1(): List<Position>`: returns a list of positions of `Hill` objects that are in the caller's `VisualRange` and which the agent can see (`HasBeeline == true`).
- `ExploreTeam(): List<IPlayerBody>`: this method returns a list with references to the calling agent's team, regardless of their location relative to the calling agent.
- `GetDistance(Position) : int`: this method returns the shortest distance (i.e., the number of grid cells) from the calling agent's current position to the specified `Position`. In order for the distance to be calculable, the grid cell specified by `Position` must be either visible (based on `HasBeeline1(Position)`). If the distance to the grid cell specified by `Position` is not calculable, the method returns `-1`.
- `HasBeeline1(Position) : bool`: this method may be called by an agent to check if the line of sight between its current position and the grid cell denoted by `Position` is free from vision-blocking obstacles (i.e., barriers or hills). If it is, the method returns `true`; otherwise, it returns `false`.

7.2.3 Tagging methods

- `Tag5(Position)`: this method takes a `Position` and attempts to prompts the calling agent to fire a shot onto the grid cell encoded by the position. If an enemy agent is currently located at that position, that agent is tagged.

  Tagging is implemented as a probability-based process that is influences by both agents' `Stance` and current positions (ground, hill, or ditch). For example, an agent in the `Lying` stance has higher accuracy but shorter `VisualRange`. If an agent is tagged, its `Energy` is decreased by 10 and the property `WasTaggedLastTick` is set to `true`. The tagging agent's `GamePoints` is increased by 10.

  For more information on tagging, see 8.2.3.
- `Reload3()`: this method prompts the calling agent to reload its tag gun. This is necessary when `RemainingShots == 0` so that the agent can continue tagging enemies. A successful call to `Reload3()` refills the calling agent's `RemainingShots` to 5.

8 Model Description

The model's main components are the Battleground and the properties and methods that serve as an interface for the agents to shape their behavior and decision-making and interact with their environment and with each other.

8.1 The Arena: Battleground

The default Battleground is a 50×50 grid layer. In order to effectively simulate the indoor nature of real-world laser tag, the Battleground is "fenced in". To add texture and complexity to the Battleground and to allow agents to interact with it in meaningful ways, maps may feature barriers, rooms, hills, and ditches (collectively called objects of interest (OOI)).

Below is a bird's eye view of an example of a square-shaped map at the start of the simulation. The following sections describe each of the OOIs. For more information on OOI, see 8.2.1, 8.2.3, and 8.2.4.

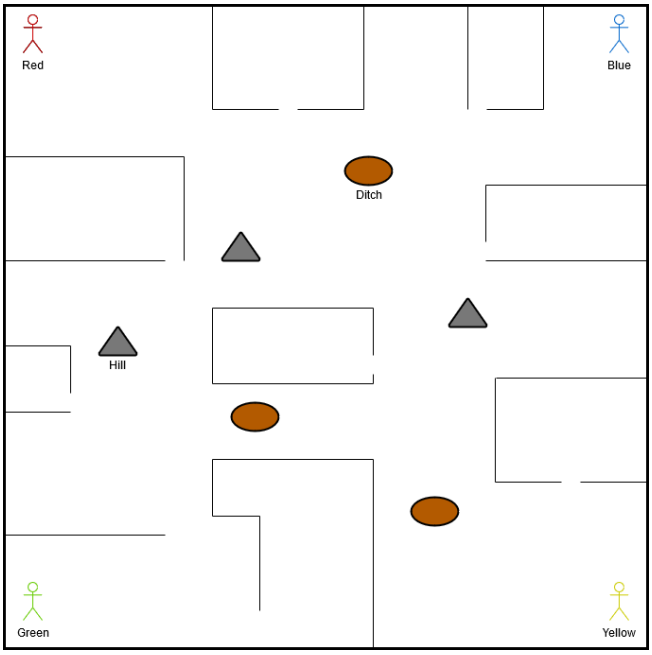

Figure 2: Example of square-shaped map (`Battleground`) at simulation start (not drawn to scale)

8.1.1 Structures

There are a few structural elements in the `Battleground` that agents can interact with by calling one of the `Explore*` methods (except `ExploreTeam` and `ExploreEnemies1`. Below is a diagram showing the inheritance hierarchy of the structures. Each exploration costs one `ActionPoint`.

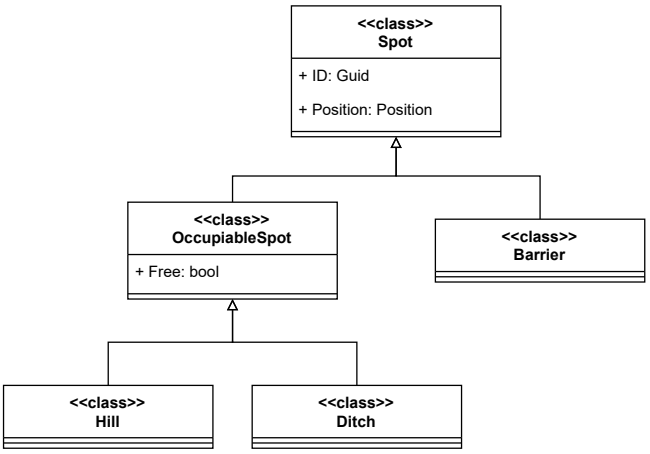

Figure 3: Structures

Barrier A `Barrier` is a structure that cannot be occupied, that acts as an impermeable obstacle to agents, and that disrupts an agent's vision. `Barrier` instances can be explored by calling `ExploreBarrier1`.

Hill A `Hill` is a structure that can be occupied. While an agent occupies a `Hill`, the agent's `VisualRange` and `Visibility` are increased. The probability of the agent getting tagged also increases.

Occpying a `Hill` might be a useful team tactic: An agent on a `Hill` can spot for another agent who is on the ground. `Hill` instances can be explored by calling `ExploreHill1`.

Ditch A `Ditch` is a structure that can be occupied. While an agent occupies a `Ditch`, the agent's `VisualRange` and `Visibility` are decreased. The probability of the agent getting tagged also decreases. Occpying a `Ditch` might be a useful for ambushing enemies. `Ditch` instances can be explored by calling `ExploreDitch1`.

Room A room is a section of the grid layer that is enclosed by barriers, leaving only one or more small gaps to enter and exit the enclosed section.

8.2 Game Mechanics

This section outlines some of the built-in logic, mechanisms, and rules of LaserTag to help you devise intelligent and feasible strategies for your agents.

8.2.1 Vision

The game features a vision system that depends on a number of variables and circumstances. The following example aims to illustrate the process. Agent X calls `ExploreEnemies1()` and hopes to see agent Y. X's ability to see Y may be influenced by each of the following conditions:

1. the relation between `distance(X, Y)` and X's `VisualRange`.
2. the relation between `Distance(X, Y)` and Y's `VisibilityRange`.
3. whether X or Y is currently located on a `Hill`.
4. whether X or Y is currently located in a `Ditch`.
5. whether the line of sight between X and Y is obstructed by a barrier or hill.

Let us examine each condition in turn:

1. If the distance between X and Y is less than or equal to X's `VisualRange`, then X may be able to see Y.
2. If the distance between X and Y is less than or equal to Y's `VisibilityRange`, then X may be able to see Y. Y's `VisibilityRange` is irrelevant for when X stands either on a `Hill` or in a `Ditch`.
3. A barrier or `Hill` can block an agent's line of sight. If there is nothing blocking X's line of sight to Y, then X may be able to see Y. (For more information on line-of-sight computation, feel free to check out Bresenham's Line Algorithm which is implemented in LaserTag to determine if any of the grid cells along the line of sight between two agents holds an vision-blocking object (a barrier or `Hill`).
4. If X is located on a `Hill`, then his `VisualRange` is increased, making it possible for him to see Y from a farther distance. Likewise, if Y is located on a hill, then his `VisibilityRange` is increased, making it easier for X to see him.
5. If X is located in a `Ditch`, then his `VisualRange` is decreased, which requires Y to be closer to X in order for X to be able to see Y. Likewise, if Y is located in a `Ditch`, then his `VisibilityRange` is decreased, requiring X to be closer to Y in order for X to be able to see Y.

Conditions 1, 2, and 3 must be met in order for X to be able to see Y. Conditions 4 and 5 merely describe how standing on a `Hill` or in a `Ditch` might affect the vision process.

8.2.2 Movement

Agents move along the grid via a modified version of the D* Lite Algorithm. The algorithm computes an initial (usually close-to-optimal or optimal) route from an agent's current position to the desired destination. Once the route has been calculated, the algorithm guides the agent towards the goal at a rate dependent on the agent's `Stance` and movement capabilities. The algorithm performs route adjustments and recalculations only if an obstacle intersects the agent's path that was not present during the initial route computation. This makes the algorithm highly efficient and perform at a better time complexity than more common path-finding algorithms such as A*.

8.2.3 Tagging

Tagging is the core game mechanic that drives LaserTag. In an attempt to simulate real-world tag-and-get-tagged interactions between laser tag players, the method `Tag5(Position)` relies on probability and randomization to create a balance between successful and unsuccessful tag attempts. If agent X attempts to tag agent Y, the outcome depends on the following factors:

1. X's `Stance`
2. Y's `Stance`
3. whether Y is currently positioned on a regular grid cell, a hill, or a ditch
4. a dose of luck

Let us examine each factor in turn:

1. If X's `Stance == Lying`, then he is most likely to tag Y. If X's `Stance == Standing`, then he is least likely to tag Y.
2. If Y's `Stance == Standing`, then X is most likely to tag him. If Y's `Stance == Lying`, then X is least likely to tag him.
3. If Y is located on a `Hill`, then his `Stance` cannot lower the likelihood of him being tagged. This is because being on a `Hill` leads to more exposure than being on the ground or in a `Ditch`. Conversely, if Y is located in a `Ditch`, then his `Stance` does not increase his likelihood of being tagged. This is because a being in a `Ditch` provides increased cover regardless of the agent's `Stance`.
4. Even if factors 1-3 are in Y's favor, there is still a chance that X tags Y. On the other hand, even if factors 1-3 are in X's favor, there is still a chance that he might miss Y and not tag him. This is due to the element of randomization added to the tagging mechanism.

8.2.4 OOI

The Battleground features `Hills` and `Ditches` as objects for agents to interact with and, under certain circumstances, gain an advantage over their opponents. A `Hill` or `Ditch` can be occupied by only one agent at a time. Being on a `Hill` increases an agent's `VisualRange` and `VisibilityRange` by five each. Being in a `Ditch` lowers an agent's `VisualRange` and `VisibilityRange` by three each. See 8.2.1 and 8.2.3 as well as 8.1.1 and 8.1.1 for more information.