

LaserTag

An Agent-Based Simulation Game

MARS Group

February 15, 2023

Version 1.0

Contents

1	Introduction	2
2	Game Objective	2
3	Project: Setup and Structure	2
3.1	Project Setup	2
3.2	Project Structure	2
4	Simulation: Setup and Execution	3
5	Visualization: Analyzing Game Outcomes	4
6	Rules	4
6.1	Game Logic	4
6.2	Constraints for Developers	5
7	Agent Properties and Methods	5
7.1	Properties	5
7.2	Methods	6
8	Model Description	8
8.1	The Environment	8
8.2	Game Mechanics	9

1 Introduction

LaserTag is an agent-based simulation game that is inspired by the real-world recreational shooting sport known as laser tag. The game is developed with the [Multi-Agent Research and Simulation \(MARS\) Framework](#), which is written in [C#](#) and runs on [.NET](#). Users of LaserTag can implement their own agents with customized behavioral logic by using the provided agent interface. The interface provides properties and methods which enable agent movement, agent state management, agent-agent and agent-environment interactions, and other functions.

2 Game Objective

A game consists of three or four teams. Each team is made up of three agents with the same behavior logic. An agent has an energy level, which is reduced when the agent is tagged by an enemy agent. The team that depletes the energy levels of all enemy agents first wins the game.

3 Project: Setup and Structure

3.1 Project Setup

Follow these steps to set up LaserTag on your machine:

1. Install the [JetBrains Rider](#) IDE.
2. Clone the [GitHub repository](#) that contains LaserTag.
3. In Rider, open the directory **LaserTagBox**.
4. If **Mars.*** dependencies are not resolved, you need to install the [MARS NuGet package](#).
 - a) Click on the NuGet tab in the bottom bar of Rider.
 - b) In the search bar, enter "Mars.Life.Simulations".
 - c) Install the newest version of the NuGet package.

3.2 Project Structure

Figure 1 illustrates the project structure and the properties and methods that are relevant for implementing agent behavior. For more detailed information on properties and methods, see Section 7.1 and Section 7.2, respectively.

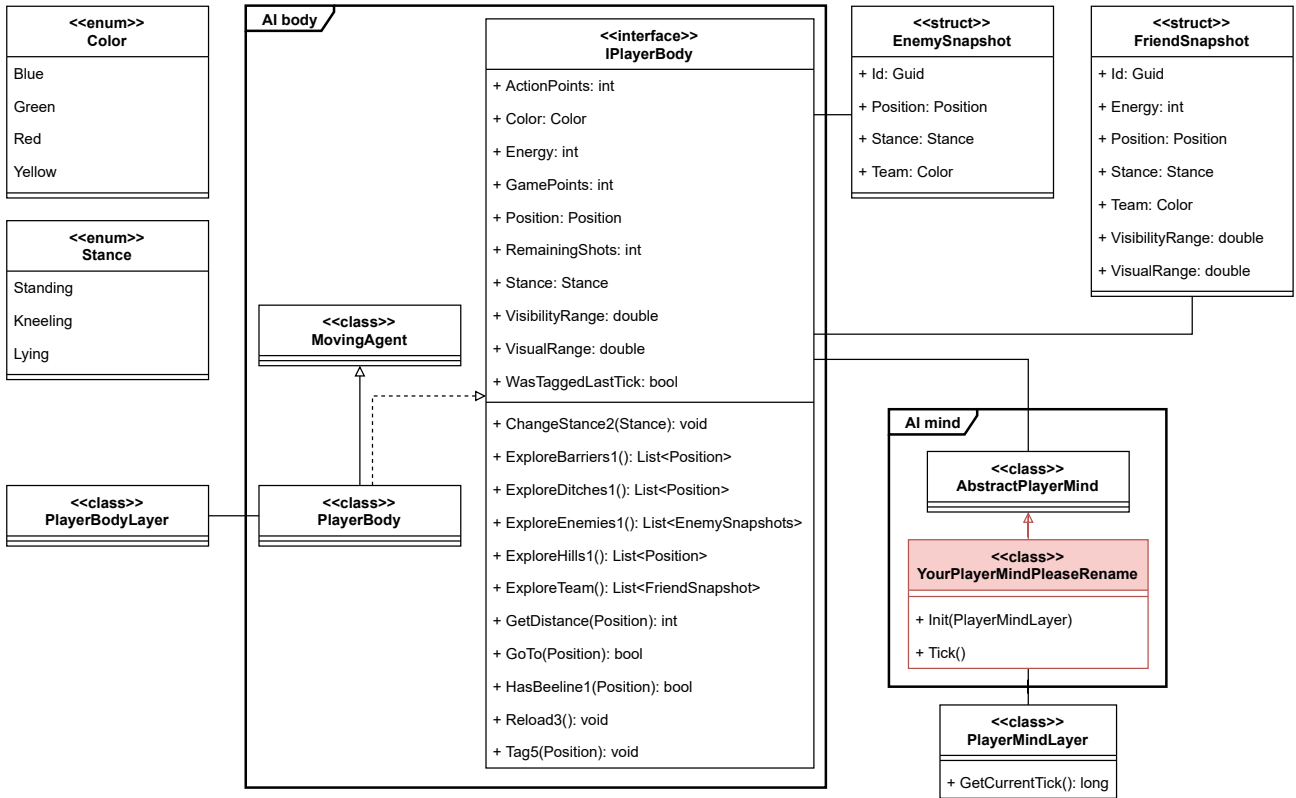


Figure 1: LaserTag class diagram

The implementation of your agents occurs in a class that inherits from **AbstractPlayerMind**. In Figure 1, this is shown by the class **YourPlayerMindPleaseRename** which is labeled in red. You can rename this class and make it your own, or add other classes to the project structure that inherit from **AbstractPlayerMind**. Your agent has a reference to the **PlayerMindLayer**, from which it can obtain the current simulation tick (**GetCurrentTick()**).

In LaserTag, each agent 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**. This interface provides 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 agent’s mind can interact with the interface to guide its body and 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 later sections of the documentation.

4 Simulation: Setup and Execution

The game is designed to be played by three or four teams. Follow these steps to set up and play a game:

1. Open the file **Program.cs**.
2. Add your agents to the **ModelDescription** object using the provided syntax:

```
description.AddAgent<MyAgent, PlayerMindLayer>();
```

In this example, **MyAgent** is the name of the class in which you implemented your agent.

3. Specify the configuration file to be used for the game using the provided syntax:

```
var file = File.ReadAllText("MyConfig.json").
```

In this example, **MyConfig.json** is the name of the JSON file that contains the desired game configuration.

4. Build and run the model in one of the following ways:

- a) In Rider, press the ► Run button
- b) In a terminal, go to the directory **LaserTagBox** and enter the commands **dotnet build** and **dotnet run**.

The JSON file **config_3.json** or **config_4.json** in the directory **LaserTagBox** can be used to configure a game with three teams or four teams, respectively. Each configuration file requires resource files that need to be in the directory **Resources**:

- A CSV file encoding the grid-based environment
- A CSV file with agent parameters

The default environment is called **rec1_battleground.csv**. It encodes an environment consisting of 50×50 grid cells. The default agent initialization file for three-player and four-player games is called **player_positions_3.csv** and **player_positions_4.csv**, respectively. Each file lists the spawn positions of the agents on the map at the beginning of the game.

5 Visualization: Analyzing Game Outcomes

A Python-based tool for post-game visualization is available in the directory **Analysis**. This tool can help you analyze and improve your agents' behavior during the development process. 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/net6.0/**. Once these files are in place, simply double-click **vis.py** to start the visualization. For more detailed information, see the README file in **Analysis**.

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 out of the environment 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 Developers

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

1. Only interact with the interface `IPlayerBody` to access the agent's physical representation.
2. When interacting with the `PlayerMindLayer`, only invoke the `GetCurrentTick()` method. Other calls to the `PlayerMindLayer` are not allowed.
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 via an external configuration file 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. These properties and methods can be accessed by your agents by inheriting from `AbstractAgentMind` (see Figure 1).

7.1 Properties

Below are the properties provided by the `IPlayerBody` interface that an agent's mind can access to gain information about the current state of its body.

7.1.1 General properties

- **ActionPoints**: An integer that specifies the number of points the agent has to perform actions during the current tick. Each action costs a specific number of **ActionPoints** (see Section 7.2 for more details). At the end of each tick, **ActionPoints** is reset to 10.
- **Color**: The agent's color, indicating the team the agent belongs to.
- **Energy**: The agent's maximum energy level is 100. It decreases if the agent gets tagged by an opponent. If the energy level is less than or equal to zero, the agent is removed from the simulation.
- **GamePoints**: the agent's score, which is increased by tagging opponents. Each tag increases the score by 10 points. If a tag causes the tagged agent's **Energy** to be less than or equal to zero, the tagged agent loses 10 **GamePoints** and is removed from the simulation. The tagging agent receives 10 additional **GamePoints** as a bonus.

7.1.2 Movement properties

- **Position**: Specifies the agent's current position on the map as an (x, y) tuple
- **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 the speed at which the agent can move.

7.1.3 Exploration properties

- **VisualRange**: An integer that specifies the number of grid cells that the agent can see from its current position. The value of **VisualRange** is set based on the value of **Stance** using the following mapping:
 - **Standing** → 10
 - **Kneeling** → 8
 - **Lying** → 5
- **VisibilityRange**: An integer that specifies the maximum distance from which the agent can currently be seen by other agents. The value of **VisibilityRange** is set based on the value of **Stance** using the following mapping:
 - **Standing** → 10
 - **Kneeling** → 8
 - **Lying** → 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** is equal to zero, then a **Reload** needs to be initiated.
- **WasTaggedLastTick**: A boolean that specifies if the agent was tagged during the previous tick.

7.2 Methods

Below are the methods provided by the **IPlayerBody** interface that an agent's mind can 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 sssec: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 (see Section 7.1.3).
- **GoTo(Position) : bool**: This is the main method used for pathfinding, movement, and path readjustment. When an agent invokes the method, the method devises a path from the agent's current position to the given **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 its path before reaching its 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 no path is calculated and no movement is initiated towards the specified destination.

For more information on `GoTo(Position)`, see Section 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 caller 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 caller can see (`HasBeeline == true`).
- `ExploreHills1() : List<Position>`: Returns a list of positions of **Hill** objects that are in the caller's **VisualRange** and which the caller can see (`HasBeeline == true`).
- `ExploreEnemies1() : List<EnemySnapshot>`: Performs an exploration of opponents in the caller's **VisualRange** and which the caller can see (`HasBeeline == true`). Returns a list of **EnemySnapshot** structs, offering limited information about the identified opponents.
- `ExploreTeam() : List<IPlayerBody>`: Returns a list of **FriendSnapshot** structs, offering limited information about the caller's team members.
- `GetDistance(Position) : int`: This method returns the shortest distance (measured in 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 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., **Barrier** or **Hill** objects). If it is, the method returns `true`, otherwise `false`.

7.2.3 Tagging methods

- `Tag5(Position)`: This method takes a **Position** and 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, then that agent is tagged.

Tagging is implemented as a probability-based process that is influenced 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 Section 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 environment 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 Environment

The default environment is a 50×50 grid layer. In order to effectively simulate the indoor nature of real-world laser tag, the environment is fenced in. To add texture and complexity to the environment and to allow agents to interact with it meaningfully, maps may feature structures (barriers, rooms, hills, and ditches).

Figure 2 shows 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 spots. For more information on spots, see [8.2.1](#), [8.2.3](#), and [8.2.4](#).

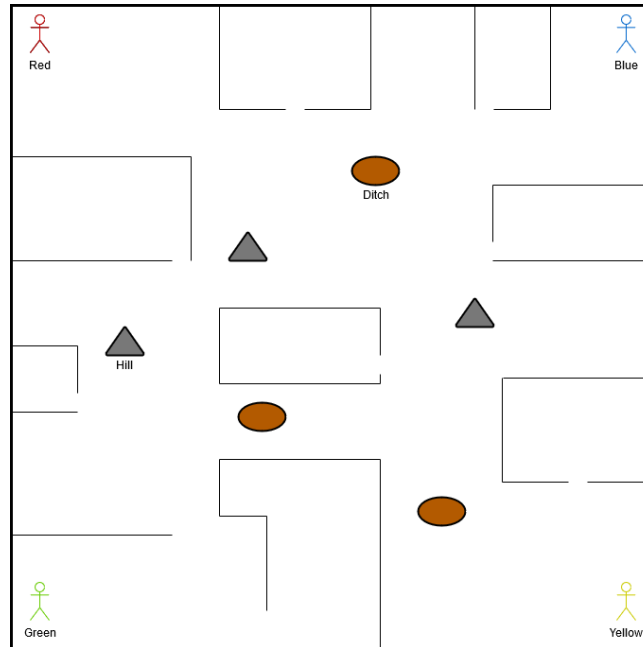


Figure 2: Example of square-shaped map environment at simulation start (not drawn to scale)

8.1.1 Structures

There are a few structural elements in the environment that agents can interact with by calling one of the `Explore*` methods (except `ExploreTeam` and `ExploreEnemies1`). Figure 3 shows the inheritance hierarchy of the structures. Each exploration costs one `ActionPoint`.

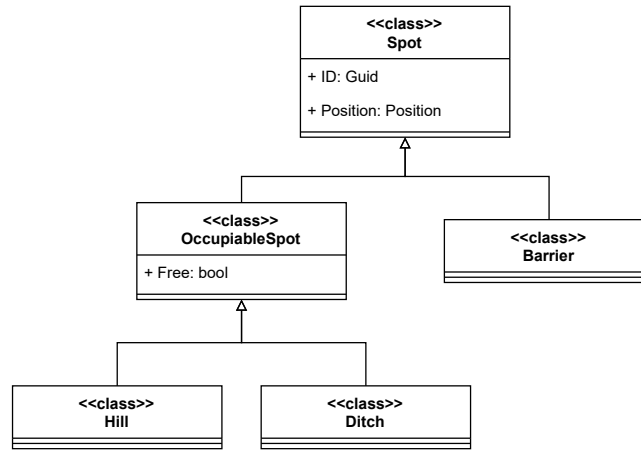


Figure 3: Structures

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. Occupying 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. Occupying 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 a 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 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 Spots

The environment features **Hill**, **Ditch**, and **Barrier** objects for agents to interact with and, under certain circumstances, gain an advantage over their opponents. A **Barrier** object cannot be occupied. 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 Sections [8.2.1](#), [8.2.3](#), [8.1.1](#), and [8.1.1](#) for more information.