LaserTag

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

Daniel Osterholz, Nima Ahmady-Moghaddam, Xavier R Adams-Stewart

June 2, 2020
Version 1.0

Version notes

Version 1.0 provides full documentation of the initial implementation of the LaserTag Framework. The framework's interface methods and game mechanics were fully tested and found to be fully functional.

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 the MARS domain-specific language (DSL) and, as such, offers all the built-in functionality, syntax, and keywords of agent-based modeling in the DSL. In addition, a number of methods are provided to serve as an interface between the AI and the game world (Battleground) and game mechanics. These methods allow agents to manage their states, move through the Battleground, and interact with other agents upon encountering them. The game includes a point system, making the goal for each player to score the highest number of points by the end of the simulation time.

2 Project and Simulation Setup

The project is available in the GitLab repository MARS Laser Tag Game. To use the LaserTag framework, a working installation of the MARS DSL plug-in in Eclipse is needed. Please see the MARS modeling handbook for further details.

The simulation can be run by opening a terminal, navigating to the project's `src-gen` directory, and entering the following command:

```
dotnet run -sm config.json -project lasertag.csproj
```

2.1 Simulation Settings

The file `config.json` includes all the external configuration settings required to run the simulation. The following parameters can be adjusted as needed:

- simulation time
- magnitude and unit of one time step (also known as a `tick` in the DSL)
- input file for the layer to be used by the agents
- names and numbers of agents

Any other parameters and attribute values pertaining to the agents need to be changed directly in the source code.

2.2 Visualization and Analysis

The simulation produces a number of `.csv` output files in the `src-gen` directory. The repository includes a couple of Python scripts that can be used to visually analyze simulation output. This can be useful for studying agent behavior and can help inform decisions on how to adjust agent logic. To run a visualization, navigate to the `Analysis` directory in the repository and run the `csv_creator` script. This generates a `agents.csv` and a `map.csv` file in the `Analysis` directory. (Note: if the directory already contains such files from previous visualizations, they will be overwritten by the new files and should therefore be saved elsewhere if needed). Once the two files are available, the `vis.py` script can be run to start the visualization tool. The tool offers play, pause, and other toggle features to run through the simulation in a time lapse while being able to observe agent movement and positioning.

3 Agent Description

An agent who participates in a LaserTag simulation is described by a set of attributes. Below is a list of the most important attributes for AI developers:

- General attributes:
  - `xSpawn` and `ySpawn`: the x- and y-coordinate of the grid cell on which the agent spawns at the start of the simulation
  - `color`: the agent's color, corresponding to the agent type name
  - `energy`: the agent's energy level 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 (`xSpawn`, `ySpawn`)
  - `actionPoints`: an integer that specifies the number of points the agent has in order to complete actions in the current tick. Each action costs a specific number of `actionPoints` (see 4.3 for more details). At the end of each tick, `actionPoints` is reset to 10.
  - `stage`: an integer attribute that can be used by agent developers to track their agent's current state and guide their decision-making processes and behaviors accordingly.
  - `points`: the agent's score. The score is increased by tagging opponents. The agent with the highest score at the end of the simulation time wins the simulation.
- Movement attributes:
  - `currStance`: a string that specifies the agent's current stance. An agent can assume three stances: "standing", "kneeling", and "lying". Each stance affects the attributes `visualRange`, `visibilityRange`, and `movementDelay` differently.
  - `movementDelay`: an integer that is set based on the value of `currStance` and that specifies the number of ticks that need to pass before the agent can move to another grid cell. The mapping from `currStance` to `movementDelay` is as follows:
    - "standing" $\rightarrow 0$
    - "kneeling" $\rightarrow 1$
    - "lying" $\rightarrow 2$
  - `hasMoved`: a boolean that states if the agent has already made a move during the current tick. An agent can make only one move per tick. At the end of each move, `hasMoved` is reset to `false`.
- Exploration attributes:
  - `visualRange`: an integer that is set based on the value of `currStance` and that specifies the agent's current range of sight. If an agent's `movementDelay === 0`, then the agent can move. The mapping from `currStance` to `movementDelay` is as follows:
    - "standing" $\rightarrow 10$
    - "kneeling" $\rightarrow 8$
    - "lying" $\rightarrow 5$
  - `visibilityRange`: an integer that is set based on the value of `currStance` and that specifies the maximum distance from which the agent can currently be seen by other agents. The mapping from `currStance` to `visibilityRange` is as follows:

    - "standing" $\rightarrow 10$
    - "kneeling" $\rightarrow 8$
    - "lying" $\rightarrow 5$
  - `barriers`: a list of x- and y-coordinates (stored in tuples) of barriers that the agent has so far encountered during exploration. New barrier locations are added to the list as they are found.
  - `hills`: a list of x- and y-coordinates (stored in tuples) of hills that the agent has so far encountered during exploration. New hill locations are added to the list as they are found.
  - `ditches`: a list of x- and y-coordinates (stored in tuples) of ditches that the agent has so far encountered during exploration. New ditch locations are added to the list as they are found.
- Tagging attributes:
  - `magazineCount`: an integer that specifies the agent's currently available opportunities to tag an opponent. If the agent's `magazineCount === 0`, then a reload process needs to be initiated.
  - `wasTagged`: a boolean that specifies if the agent was tagged during the previous tick.
  - `tagged`: a boolean that specifies if the agent successfully tagged an opponent during the previous tick.

These are all the attributes needed by the player to guide his/her agent through the game. For a full list of attributes, feel free to consult the source code.

4 Framework Description

The framework's main components are the Battleground and the methods that serve as an interface for the playing agents to interact with their environment and with each other.

4.1 The Arena: Battleground

The Battleground is a square-shaped grid-layer. In order to effectively simulate the indoor nature of real-world laser tag, the Battleground is "fenced in". This forces agents to remain within the bounds of the grid's dimensions and prevents them from disappearing on one end and reappearing on another end of the grid. To add texture and complexity to the Battleground and to allow agents to interact with it in meaningful ways, it features barriers, rooms, hills, and ditches.

4.1.1 Barrier

A barrier is represented by a `Barrier` agent. A barrier blocks any direct movement or vision through it. For example, an agent X cannot move through a barrier or see through a barrier, and therefore cannot interact with an agent Y who is positioned behind the barrier relative to X's position.

4.1.2 Rooms

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

4.1.3 Hills

A hill is represented by a `Hill` agent. An agent X can climb onto a hill to gain an extended visual field and be able to see other agents from farther distances. However, being exposed on a hilltop also makes X an easier-to-tag target for other agents and having to tag agents across long distances also lowers X's accuracy.

4.1.4 Ditches

A ditch is represented by a `Ditch` agent. An agent X can jump into a ditch to become less easily visible to other agents. However, X's visual range is also affected by being inside the ditch.

4.1.5 Battleground Example

Below is a bird's eye view of an example of the Battleground at the start of the simulation.

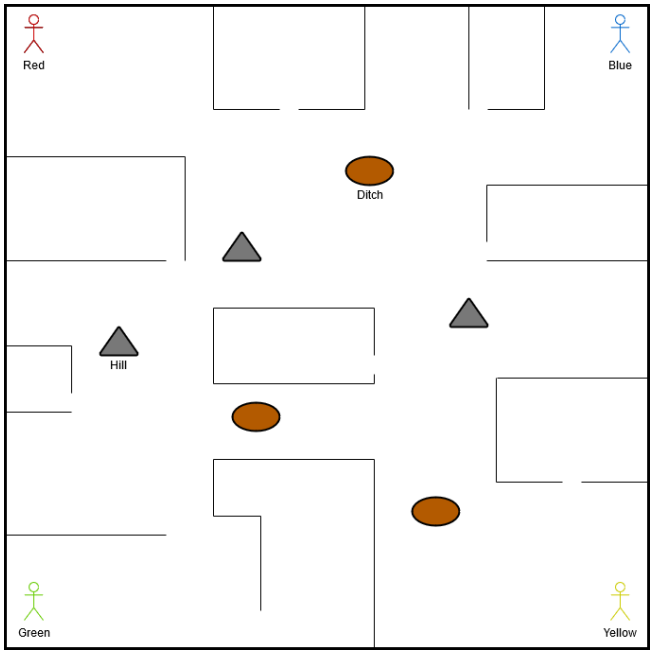

Figure 1: Example of Battleground at simulation start (players and grid cells not drawn to scale)

4.2 AI Interface and Methods

A number of methods are available for the agents to interact with the environment and with each other. In the source code, they are labeled with a comment "USER METHOD" above the method signature. These methods' internal logic supplies both the decision-making mechanisms needed for the agent to play the game and the maintenance and management of the agent's attribute values based

on his current state. The following component diagram aims to illustrate the connection points made available to the agents, which are further described in the list below.

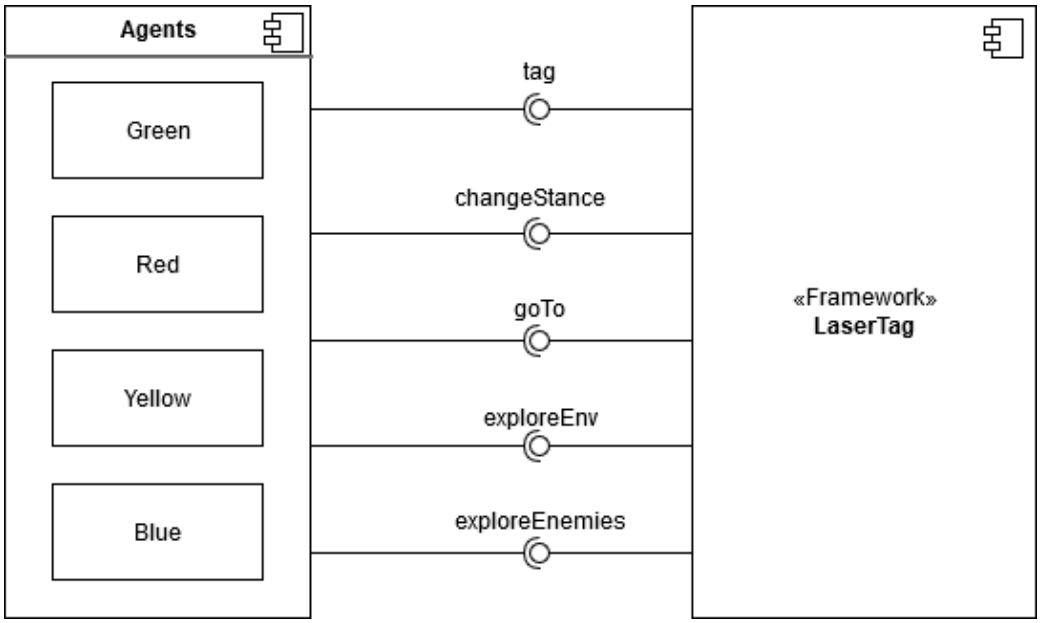

Figure 2: LaserTag Component Diagram showing interfaces made available to agents by framework

Below are the methods that may be called by the agent during his `tick` method:

- `exploreEnv(o : string) : void`: this method takes a string and, based on its value, initiates an exploration of part of the agent's environment. The string may be "barriers", "hills", or "ditches". The method will explore `Barrier`, `Hill`, or `Ditch` agents in the agent's field of vision, and any exploration results are stored in the agent's attributes `barriers`, `hills`, or `ditches`, respectively. Each exploration costs one `actionPoint`.
- `exploreEnemies() : Tuple<Agent[], Agent[], Agent[]>`: this method performs an exploration of opponents in the agent's field of vision. Opponents are stored in the collection that matches their type. Unlike the returns of `exploreEnv` calls, the results of opponent exploration do not persist beyond a simulation tick. Each exploration costs one `actionPoint`.
- `goTo(real, real) : bool`: this is the main method for the framework's path-finding, movement, and path readjustment algorithm. When the agent enters a desired destination (x-coordinate and y-coordinate) and the algorithm will devise a way from the agent's current position to the destination. From then on, each call to the method will move the agent one step along the path (if possible) closer to the destination until the destination is reached. Movement is possible only when the agent's attribute `hasMoved == false` and when `movementDelay == 0`. A recalculation of the path occurs automatically in case of an unforeseen obstacle. The method returns `true` when a move was made successfully and `false` when, for any reason, a move was not made.
- `changeStance(string) : void`: this method takes a string and allows the agent to change between three possible stances: "standing", "kneeling", and "lying". Stance changes affect the values of the agent's attributes `visualRange`, `visibilityRange`, and `movementDelay`. Each change of stance costs two `actionPoints`.
- `tag(Agent) : void`: this method takes an agent (usually from the calling agent's result of `exploreEnemies()`) and attempts to tag that agent. 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, his `energy` is decreased by 20 and the attribute `wasTagged` is set to `true`. The tagging agent's `points` is increased by 10 and the attribute `tagged` is set to `true`. Each tag attempt costs five `actionPoints`.

- `tag(real, real) : void`: this method takes an x- and y-coordinate and checks if an opponent is located at the corresponding grid cell. If yes, then the calling agent will attempt to tag the agent located at that grid cell. In this case in the same manner as outlined in the above `tag` method description, This method, however, allows the calling agent to potentially tag an opponent agent from any distance. Each tag attempt costs five `actionPoints`.
- `reload`: this method is not part of the framework's interface. It is still listed here for reference because it is called automatically when an agent's `magazineCount === 0`. Reloading refills the `magazineCount` to five and costs three `actionPoints`.

The following class diagram aims to give an overview of the agents and objects present in a LaserTag simulation as well as their relationships to each other and to the grid-layer they exist on.

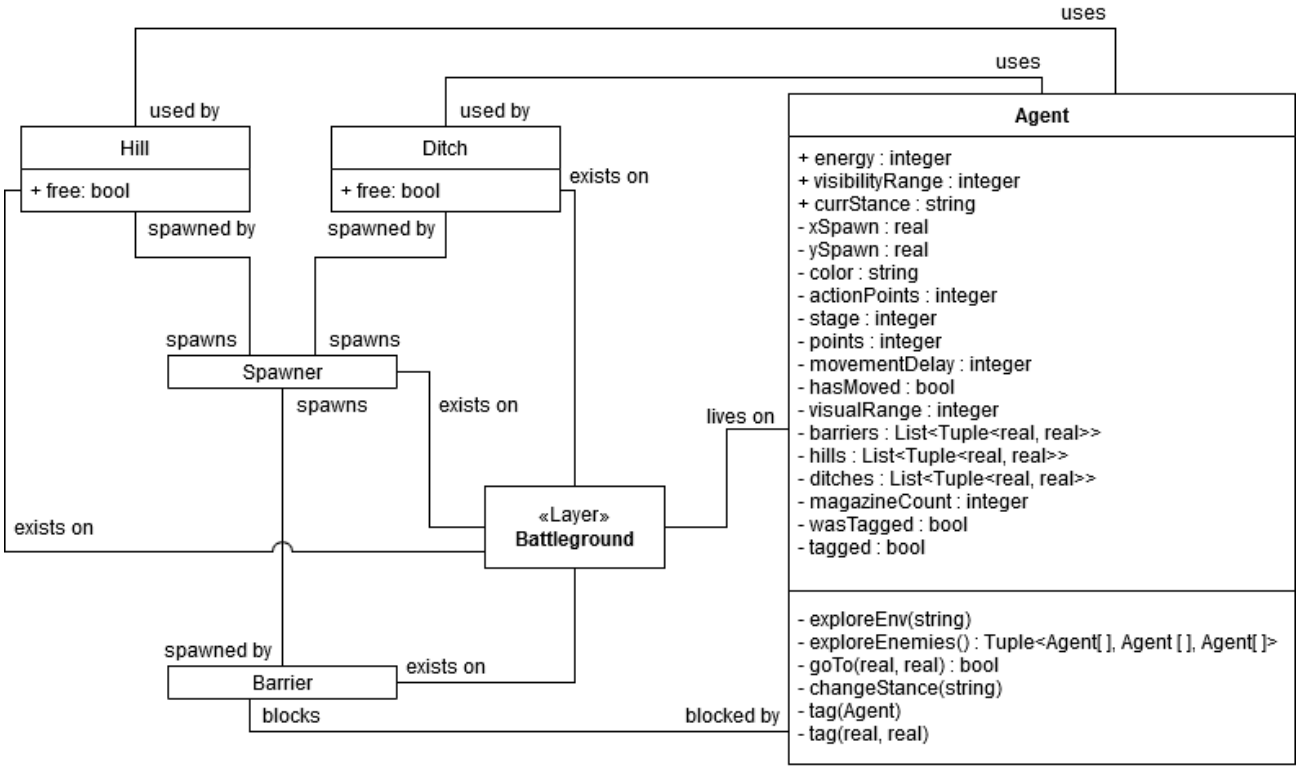

Figure 3: LaserTag Class Diagram showing relationships between agents and layer

4.3 Essential Game Mechanics

The following section serves to outline some of the built-in logic and mechanics that are important to understanding the game and being able to devise intelligent and feasible strategies for an agent to potentially play the game well.

- **Vision**: the framework features a vision and sight system that depends on a number of variables and circumstances. The following example aims to illustrate the process. Agent X is performing

`exploreEnemies` and hoping to see Agent Y. X's ability to see Y may be influenced by each of the following:

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 is currently positioned on a regular grid-cell, a hill, or a ditch
4. whether X can directly see Y with no obstacle obstructing the line of sight

Let us examine these factors one at a time:

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.
3. If X is standing on a hill, then his `visualRange` is increased, making it more difficult for Y to remain unseen. On the other hand, if Y is standing on a hill, then his `visibilityRange` is increased, making it more difficult for him to remain unseen.
4. 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 the LaserTag model to determine if two agents' line of sight is unobstructed).

All of these conditions must be met in order for X to be able to see Y.

- **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 (x- and y-coordinate). Once the route has been calculated, the algorithm guides the agent along the route at a rate dependent on the agent's `movementDelay` and `hasMoved` attribute values. The algorithm performs route readjustments and recalculations only if an obstacle presents itself in the agent's path that was not there during the initial route computation. This makes the algorithm highly efficient and perform at a better runtime complexity than common path-finding algorithms such as A*.
- **Tagging**: Tagging is the core game mechanic that drives LaserTag. In an attempt to simulate real-world shoot-and-hit interactions between laser tag players, the `tag` implementation relies on probability and randomization to create a balanced mix between successful and unsuccessful tag attempts. If agent X attempts to tag agent Y, the outcome depends on the following factors:

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

  Let us examine these factors one at a time:

  1. If X's `currStance === "lying"`, then he is most likely to tag Y. If X's `currStance === "standing"`, then he is least likely to tag Y.
  2. If Y's `currStance === "standing"`, then X is most likely to tag him. If Y's `currStance === "lying"`, then X is least likely to tag his.

3. If Y is positioned on a hill, then his `currStance` cannot lower the likelihood of him being tagged. This is because a hilltop leads to more exposure than being on the ground or in a ditch. If Y is positioned in a ditch, then his `currStance`, then his `currStance` does not increase his likelihood of being hit. This is because a dich provides sufficient cover regardless of the agent's stance.

4. Even if the first three factors are in Y's favor, there is still a chance that X successfully tags Y. Conversely, even if the first three factors are in X's favor, there is still a chance that he might miss Y and not tag him. This is the randomization portion of the tagging method.

- **Points of interest (POI)**: 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. Standing on a hill increases an agent's `visualRange` and `visibilityRange` by five each. Standing in a ditch lowers an agent's `visualRange` and `visibilityRange` by three each. Please see the above sections on **Vision** and **Tagging** for more information.