LaserTag

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

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

June 14, 2020
Version 1.1

Version notes

Version 1.1 includes:

- bug fixes pertaining to `tag` methods
- removal of `println` statements
- addition of `exploreTeam` method
- addition of `memberID` in agent attributes
- update how standing on a hill or in a ditch affects `exploreEnemies` result
- added deduction of 10 points for tagged agent whose `energy <= 0`
- labeled `reload` as a "USER METHOD"

Documentation updates

- Specified game objective in Section 1.1.
- Added Rules section. See 3.
- Added section on `Spawner` to Section 5.1.
- Added method descriontion for `exploreTeam()` in Section 5.2 and updated Figure 2 and 3.
- Updated how vision works in Section 5.3.1.

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). 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.

1.1 Objective

The game features a point system. Let X and Y be agents on different teams:

- When X tags Y, X receives 10 points.
- When X's tag causes Y's energy to become less than or equal to 0, X receives an additional 10 points and Y loses 10 points.

At the end of the simulation, the points of each team member on a team are added together. The team with the highest sum of points wins the game.

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. Installation instructions can be found in the MARS modeling handbook.

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 pertaining to the simulation. The following parameters can be adjusted as needed:

- simulation time
- length 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
- number of each agent type (default: three per team and one `Spawner` agent)
- spawn coordinates and member IDs of agents (in `in_*.csv` files in the `src-gen` directory)

2.2 Visualization and Analysis

The simulation produces `*.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, helping trace the events that occurred during the simulation flow and allowing players to study their agents' behavior to inform decisions on how to adjust agent logic.

To start 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, run the `vis.py` script to start the visualization tool. The tool offers play, pause, and other toggle features to run through the simulation tick-by-tick or in a time lapse to observe agent behavior.

The visualization tool is a modified version of a tool implemented by Finn-Lasse Joergensen.

3 Rules

In order to play the game as intended, any agent behavior specified in the `tick` method must meet the following requirements:

1. Only methods labeled with "USER METHOD" in a comment above the method signature may be called:
   - `exploreEnv(string)`
   - `exploreTeam() : Agent[ ]`
   - `exploreEnemies() : Tuple<Agent[ ], Agent[ ], Agent[ ]>`
   - `goTo(real, real): bool`
   - `changeStance(string)`
   - `tag(Agent)`
   - `tag(real, real)`
   - `reload()`
2. The following MARS DSL keywords may not be used: `distance`, `explore`, `kill`, `move`, `nearest`, `pos at`, and `spawn`.
3. Agent attributes may be queried, but may **not** be set. The only attribute whose value may be set during the `tick` method is `stage`.
4. No use of setter methods pertaining to the `battleground` layer. `DimensionX`, `DimensionY`, and getter methods pertaining to the `battleground` layer may be used.
5. No loops that are known not to terminate after a reasonable time (example: `while(true)`) may be included.

4 Agent Attributes and Values

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

4.1 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.
- `memberID`: an integer that uniquely identifies the agent within his team
- `color`: the agent's color, corresponding to the agent type name.
- `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 (`xSpawn`, `ySpawn`) with `energy = 100`.
- `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 5.2 for more details). At the end of each tick, `actionPoints` is reset to 10.
- `stage`: an integer attribute that can be used by AI developers to track their agent's current state and guide their decision-making processes and behaviors accordingly. This is the only attribute whose value may be set during the `tick` method.
- `points`: 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, he loses 10 points and an additional 10 points are awarded to the tagger as a bonus. The agent with the highest score at the end of the simulation time wins the simulation.

4.2 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. If an agent's `movementDelay === 0`, then the agent can move. 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 tick, `hasMoved` is reset to `false`.

4.3 Exploration attributes

- `visualRange`: an integer that is set based on the value of `currStance` and that specifies the agent's current range of sight. The mapping from `currStance` 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 `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 tuples of x- and y-coordinates 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 tuples of x- and y-coordinates of hills that the agent has so far encountered during exploration. The locations of newly found hills are added to the list as they are found.
- `ditches`: a list of tuples of x- and y-coordinates of ditches that the agent has so far encountered during exploration. The locations of newly found hills are added to the list as they are found.

4.4 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 tagged an opponent during the previous tick.

These are the attributes needed by AI developers to guide their agent(s) through the game. For a full list of attributes, see the source code.

5 Framework Description

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

5.1 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.

5.1.1 Battleground Example

Below is a bird's eye view of an example of the Battleground at the start of the simulation. The following sections describe each of the objects of interest (OOI). For more information on OOI, see 5.3.1, 5.3.3, and 5.3.4.

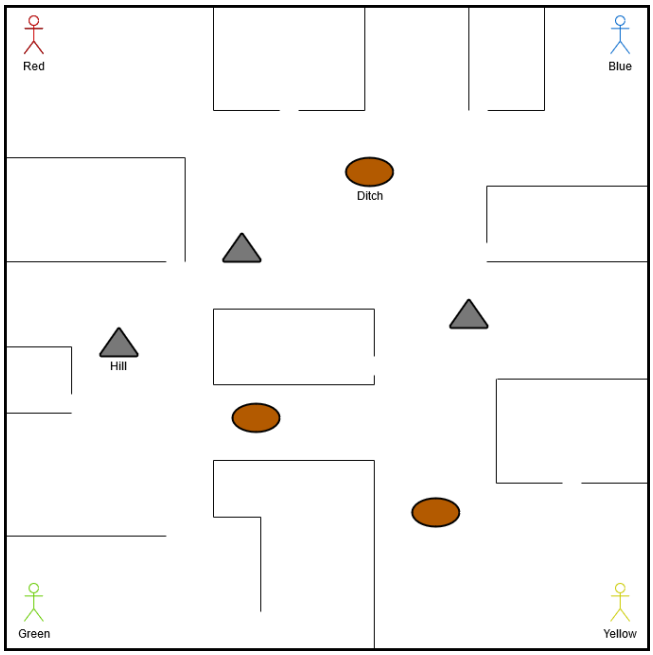

Figure 1: Example of Battleground at simulation start (not drawn to scale)

5.1.2 Barrier

A barrier is represented by a `Barrier` agent. In 1, barriers are drawn as straight lines for simplicity. 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.

5.1.3 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.

5.1.4 Hill

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.

5.1.5 Ditch

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.

5.1.6 Spawner

The `Spawner` is not accessible to the players. This agent is responsible for spawning OOIs at the beginning of the simulation. The spawn locations are specified by the cells values in the `.csv` file that is used as the grid-layer for the agents to move on. The mapping from cell value to OOI is as follows:

- $0 \rightarrow$ nothing
- $1 \rightarrow$ `Barrier`
- $2 \rightarrow$ `Hill`
- $3 \rightarrow$ `Ditch`

5.2 Agent Interfaces 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 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.

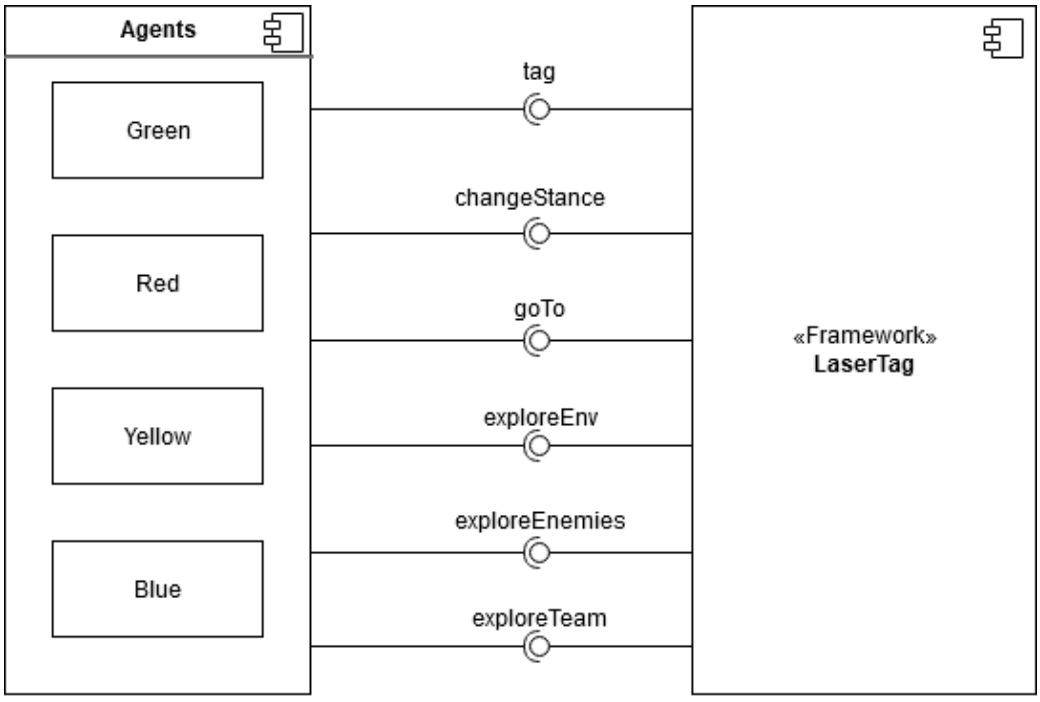

Figure 2: LaserTag Component Diagram showing interfaces made available to agents by framework

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

- `exploreEnv(string) : void`: this method takes a string and, based on his 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`.
- `exploreTeam() : Agent[]`: this method generates an array with references to all of the calling agent's team members, regardless their location relative to the calling agent. Unlike the return values of `exploreEnv(string)` calls, the result of an `exploreTeam()` call does not persist beyond the simulation tick in which the call was made. 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 array that matches their type. Unlike the return values of `exploreEnv(string)` calls, the result of an `exploreEnemies()` call does not persist beyond the simulation tick in which the call was made. 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), the algorithm devises a way from the agent's current position to the destination. From then on, each call to `goTo(real, real)` 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 and `false` when, for any reason, a move was not made. For more information on agent movement in LaserTag, see 5.3.2.
- `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 10 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`. For more information on tagging, see 5.3.3.
- `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, and if the line of sight from the agent to the grid cell is unobstructed, then the calling agent will attempt to tag the agent located at that grid cell in the same manner as outlined in the above method description `tag(Agent)`. 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 part of the `tag` interface. It is called automatically when an agent's `magazineCount === 0`, but may also be called manually by the agent if needed. 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. For readability, several attributes and methods were omitted. For more details, consult the source code.

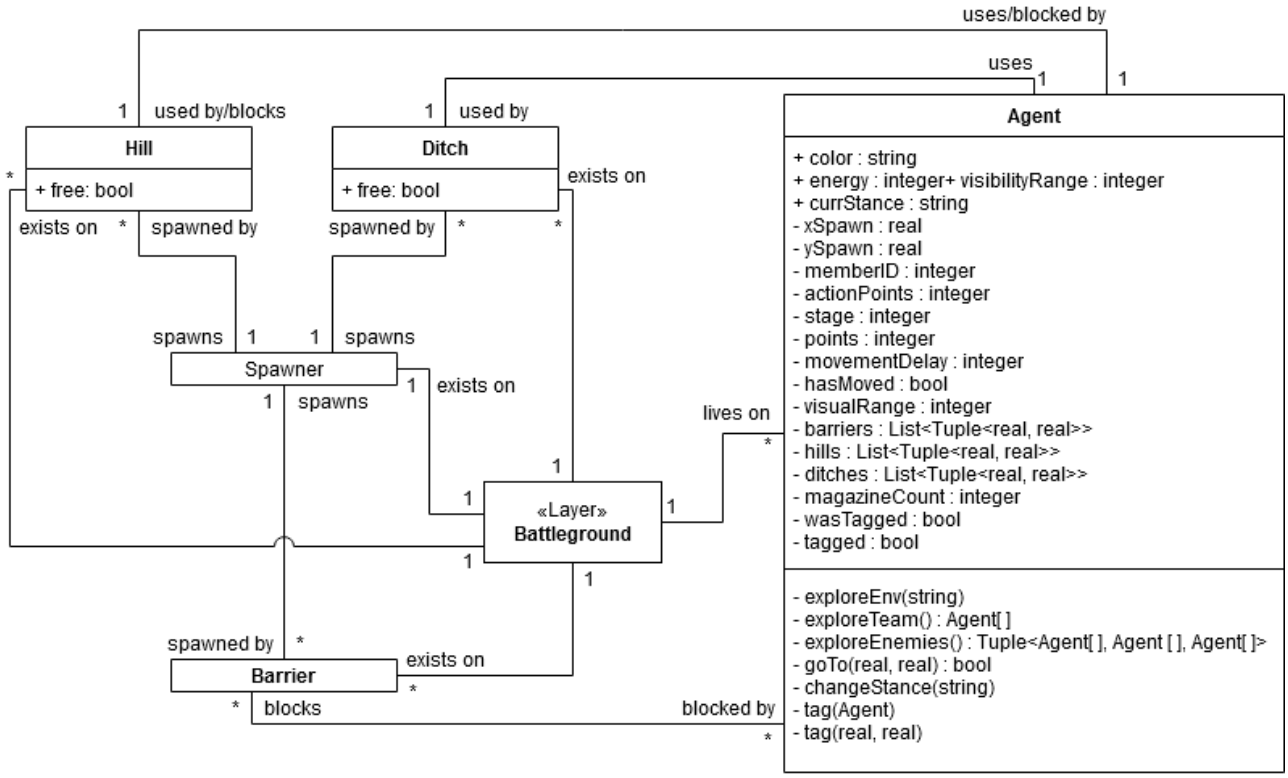

Figure 3: LaserTag Class Diagram showing relationships between agents and layer

5.3 Game Mechanisms

The following section serves to outline some of the built-in logic, mechanisms, and rules of LaserTag to help AI developers devise intelligent and feasible strategies for their agents.

5.3.1 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 calling `exploreEnemies()` and hoping 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` becomes obsolete when X stands either on a hill or in a ditch.
3. 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.
4. If X is located in a ditch, then his `visualRange` is decreased, which requires Y to be closer to X in order for him 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 him to be able to see Y.
5. 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 object (a barrier or hill that causes an obstruction).

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

5.3.2 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 (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 current values of the attributes `movementDelay` and `hasMoved`. 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 common path-finding algorithms such as A*.

5.3.3 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 methods `tag(Agent)` and `tag(real, real)` rely 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 `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 each factors in turn:

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 him.

3. If Y is located on a hill, then his `currStance` 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 `currStance` 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.

5.3.4 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. 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. See 5.3.1 and 5.3.3 as well as 5.1.4 and 5.1.5 for more information.