Minion Wars

Using Agents to Simulate a Battlefield

Joshua Haskins

TABLE OF CONTENTS

INTRODUCTION

The goal of this simulation is to have agents (referred to as minions) battle one another on different teams to completely wipe out the enemy team(s). There are different types of agents, including a base, healer and attacker. Each unit has its' own advantage and specific parts in the simulation. Agents can attack and heal another unit by being on the same x and y coordinates on the battlefield (20x20 grid).

Each agent is able to see the entire battlefield. Healers only heal their own teammates, if there are no teammates that need to be healed, they will just stay still. Attackers will attack all enemy units; the order of these attacks is determined by exactly what type of attacker this minion is. See Detailed View, for a complete description. The base is the objective of the simulation, destroy the base, and win the simulation.

PROBLEM STATEMENT

WHY USE AGENTS?

Agents are used in this simulation, as each unit needs to be autonomous and figure out the best way to attack or heal their way to a win. There are messages relayed between agents that convey important information regarding position, health, and team.

IS IT SUITABLE TO USE AGENTS?

This simulation is very suitable for the use of agents, as it simulates what an actual battlefield could look like, where each unit has the choice to do what it would like, or to follow orders. Units are able to go rogue, work on one specific mission, or to follow the suggestions of fellow teammates or the base commander.

CURRENT VERSION DESCRIPTION

The current version of the simulator allows for a max of 16 agents to combat each other to determine a winner. With any additional agents, the system sees an extreme slow down and possibly even freeze. Agent positions are randomly generated at the beginning of each simulation, except for the Bases, which are always in one of the four corners. Attacks will attack the first enemy unit, after their required attacks are met. This could be further expanded upon to attack the most damaged unit first.

DETAILED VIEW

This simulation was implemented through a combination of both Java and Jason.

JAVA

There are three main components of the simulator designed in Java. The first being the environment, which is a 20x20 grid and positions of the agents (except the Base) being randomly generated. The second being the GUI that displays the environment and the agents moving around the grid attacking or healing one another. The third component being that any agent calls are handled through the executeAction() method. These actions are ones that not from Jason and are not implemented in the Jason code for that agent.

JASON

There are three different agents coded in Jason. The first being the Base, which is the target of the simulation. Agents of the same team protect their base, while enemies are trying to destroy this Base. Bases only check their health and if it below 100% they request to be healed. If the health is below 30%, the base declares an emergency status and all healers retreat to the base to heal it.

Healers are another type of agent. These agents look at their percepts to look for teammates who need to be healed. Their first priority for healing is their base, if the base is fully healed, the healer will then search for other healers or attackers to heal. If there are no teammates to heal, the unit will wait a specified amount of time for a delay, then search again.

Attackers are the last type of agent. They first check the base area (3x3 grid around the base) for enemy units, if there are any enemy units in this location, this will be their target. If there are no enemies near the base, the will then turn to their first plan. There are four different types of attackers. The first agent will simply just attack the first enemy seen in its percepts. The second type will attack healers first, and then proceed with all other agent types. The third type will attack enemy attackers first, then proceed with other enemy unit types. The last type will first attack enemy bases, then proceed with other enemy unit types. This type of agent is basically suicide as enemy’s protect their base using the 3x3 grid, unless they are all dead, in which the base would be destroyed easily.

IMPLEMENTATION

Please see Appendix 2 for a class diagram represent that Java implementation of this portion of the project.

Each team is represented by their own unique colour. The following identifies each unit:

- a = Attacker

- b = Base
- h = Healer
- x = Multiple minions on one location

See Appendix 1 for a screenshot showing the battlefield. Also see Appendix 3 for a screenshot displaying the simulation over screen.

ISSUES ENCOUNTERED

Some of the issues encountered with the project are as follows:

1. Since I wrote the environment and GUI separately, it took a bit of work to tie these two systems correctly. Also ensuring that the GUI refreshes at a consistent rate and does not refresh too often, causing screen tearing or graphical glitches.
2. Having agents communicate directly with another agent was a problem as they tell command would add the percept, but then whenever the updatePercepts() command was run, it would wipe out this data. To overcome this, I created a Java method called tell() and tell2() to add percept to a linked list of current percepts. Whenever updatePercepts() is called, it now recreates any new percepts that are in the database.
3. Agents sometimes will stop and/or freeze. This problem has largely been solved by removing plans that were inconsistent and cased the agent to freeze.
4. With large amounts of agents running at the same time the entire simulation slows down. This problem has not been solved.
5. Synchronization, initially I did not have synchronization enabled on methods that modified percepts. This caused issues as some percepts would get wiped out right after another agent used the method. By using the keyword Synchronized in Java I was able to ensure that only one agent is able to use the method at a time. This does slow down the system slightly, but it does ensure consistency across all agents.

EXPANSION & FUTURE IDEAS

This simulation has many areas where it can be expanded. The first is through additional unit types, having minions that act as rocks, not allowing other minions to move onto that location. Another type is a defender, which was initially planned, was due to time constraints were unable to create. This unit would act like a shield when it moved onto a location; it would have a much higher health than the unit it was protecting. The unit it is protecting could continue doing what it was prior to being shielded, which could be healing or attacking.

Another idea was to add health and communication information to the GUI for all agents. Agents that are dead would be represented with the word “dead” next to their name.

Also with further testing, balancing of units could be achieved. As currently, if a team has two healers and the enemy team has one attacker, the simulation typically ends in a stalemate (this depends how far the secondary healer has to travel in order to heal the injured healer) as the healers will continue to heal each other and cannot attack the attacker.

Another additional feature would be to add more intelligence to the agents. Give them more formulas to choose how to act based on conditions in the environment.

Finally, one more method would be to make the environment more dynamic by doing some of the following: make the battlefield size random, which is similar as to how the current positions of both the Attackers and Healers are determined. Have objects in the environment that obstruct the movement of agents. The obstacles could also move randomly, or in a strategic pattern in order to bring a more complex simulation.

CONCLUSION

This project was very successful in teaching me how to create an agent system from scratch, implement synchronization and get agents to use communication. If more time was available, or was done in a group setting, this project could have been even more epic. Though many problems where encountered, I managed to traverse them and end with a system that does function very similar to how I had initially envisioned it. Even though there are many different ways to expand this system, I am happy with the current status of the system. Agents are a very interesting field and I can see many different areas in which they can be implemented. Overall I did very much enjoy this project and course.

REFERENCES

M. J. Wooldridge, *An Introduction to MultiAgent Systems*, 2nd ed. Chichester, U.K: John Wiley & Sons, 2009.

Jason. 2014.

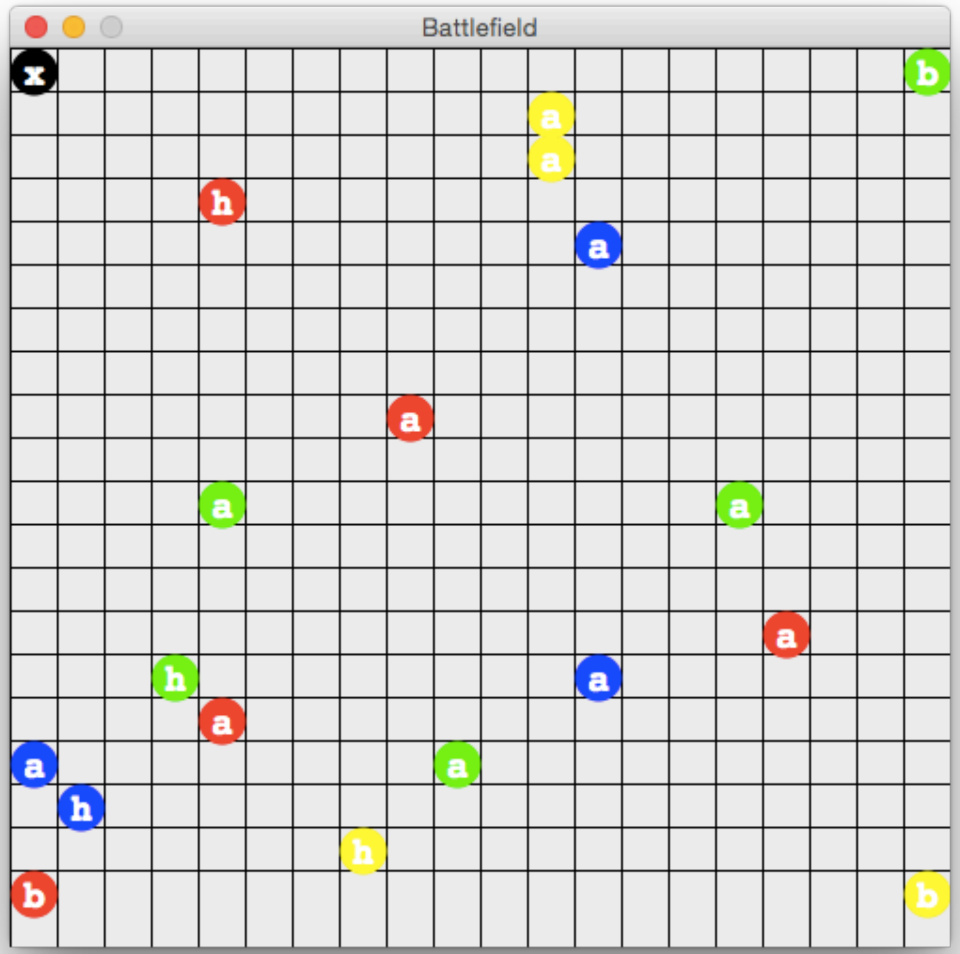
"Overview (Jason - AgentSpeak Java Interpreter)," *Jason*. [Online]. Available: http://jason.sourceforge.net/api/. [Accessed: 22-Nov-2014].

Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge, *Programming Multi-agent Systems in AgentSpeak Using Jason*, 1st ed. Wiley-Blackwell, 2007.

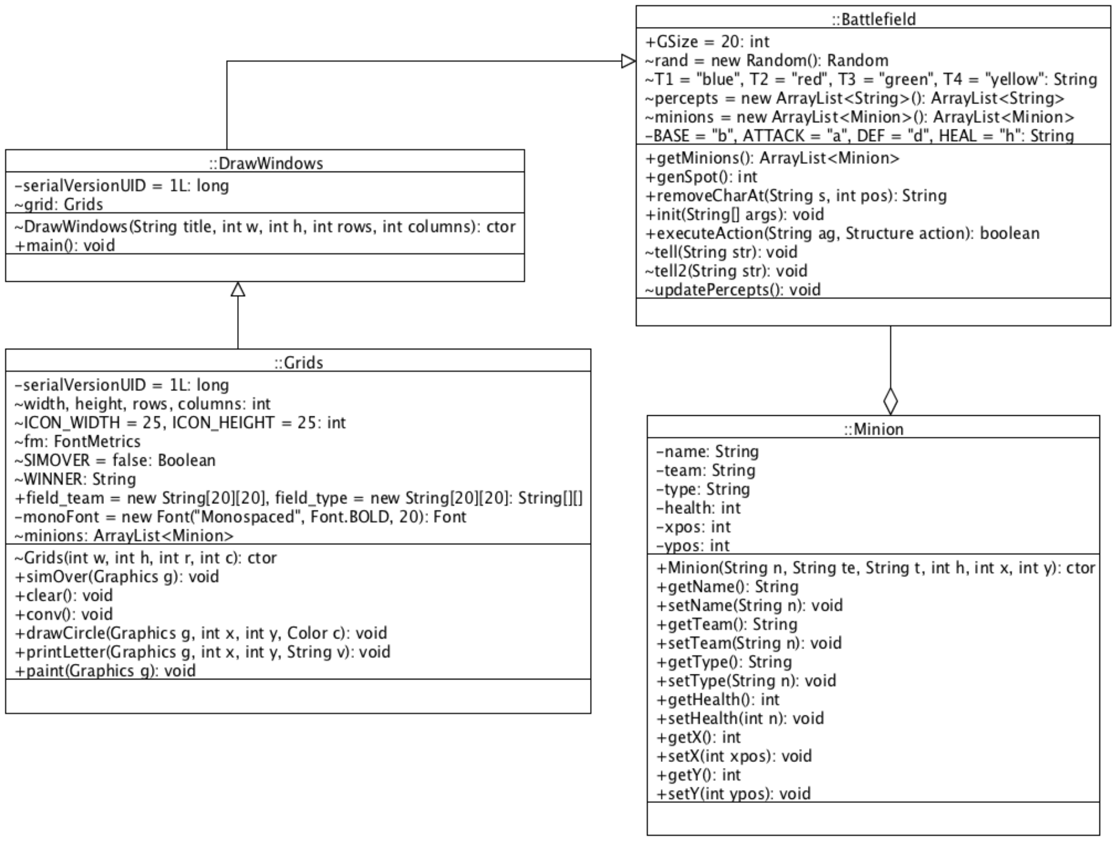
APPENDIX

The following items support the documentation of the project.

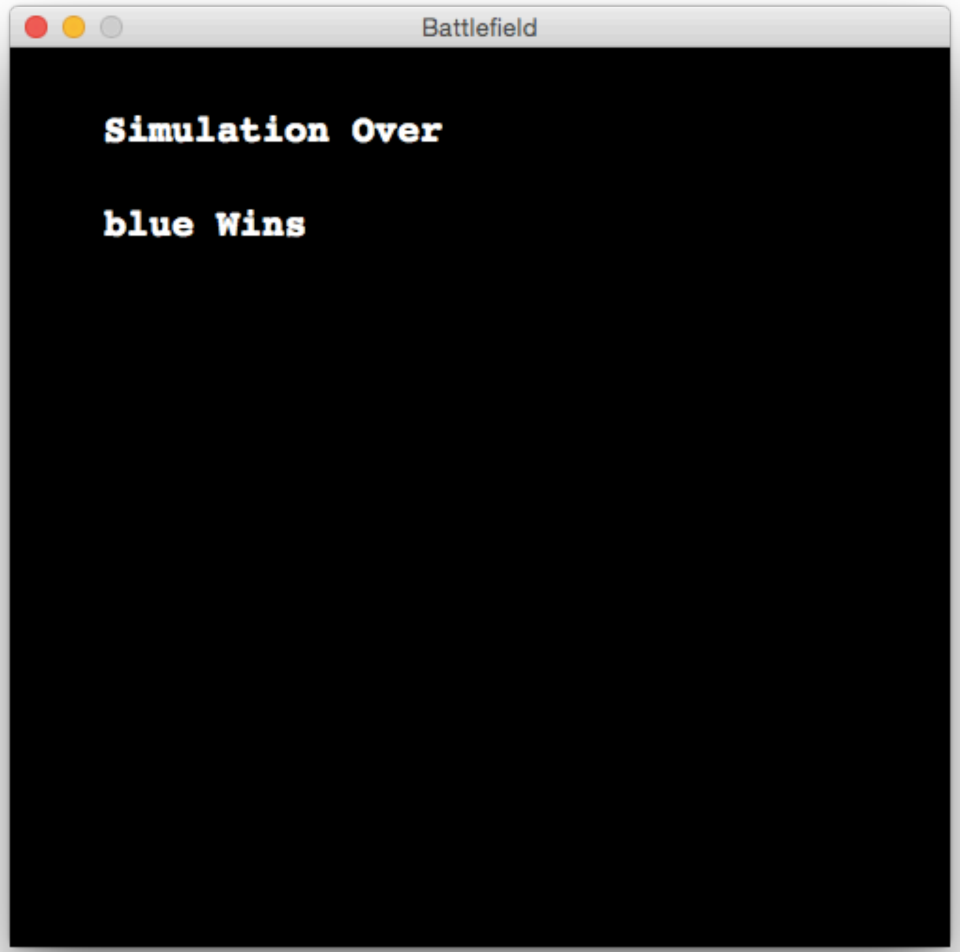
Appendix 1
Main Window



Appendix 2
Class Diagram


::Battlefield
+GSize = 20: int
~rand = new Random(): Random
~T1 = "blue", T2 = "red", T3 = "green", T4 = "yellow": String
~percepts = new ArrayList<String>(): ArrayList<String>
~minions = new ArrayList<Minion>(): ArrayList<Minion>
-BASE = "b", ATTACK = "a", DEF = "d", HEAL = "h": String
+getMinions(): ArrayList<Minion>
+genSpot(): int
+removeCharAt(String s, int pos): String
+init(String[] args): void
+executeAction(String ag, Structure action): boolean
~tell(String str): void
~tell2(String str): void
~updatePercepts(): void
::DrawWindows
-serialVersionUID = 1L: long
~grid: Grids
~DrawWindows(String title, int w, int h, int rows, int columns): ctor
+main(): void
::Grids
-serialVersionUID = 1L: long
~width, height, rows, columns: int
~ICON_WIDTH = 25, ICON_HEIGHT = 25: int
~fm: FontMetrics
~SIMOVER = false: Boolean
~WINNER: String
+field_team = new String[20][20], field_type = new String[20][20]: String[][]
-monoFont = new Font("Monospaced", Font.BOLD, 20): Font
~minions: ArrayList<Minion>
~Grids(int w, int h, int r, int c): ctor
+simOver(Graphics g): void
+clear(): void
+conv(): void
+drawCircle(Graphics g, int x, int y, Color c): void
+printLetter(Graphics g, int x, int y, String v): void
+paint(Graphics g): void
::Minion
-name: String
-team: String
-type: String
-health: int
-xpos: int
-ypos: int
+Minion(String n, String te, String t, int h, int x, int y): ctor
+getName(): String
+setName(String n): void
+getTeam(): String
+setTeam(String n): void
+getType(): String
+setType(String n): void
+getHealth(): int
+setHealth(int n): void
+getX(): int
+setX(int xpos): void
+getY(): int
+setY(int ypos): void

Appendix 3
Simulation Over Screen



```
Simulation Over

blue Wins
```