Sistemi Autonomi

Moba Simulator

Creazione di un Multiplayer Online Battle Arena con Jason

Professore: Andrea Omicini

Studente: Giacomo Zanotti

Matricola: 0000793982

Indice

1 Introduzione

Il progetto ha come obiettivo la simulazione di un gioco di tipo **MOBA** (**M**ultiplayer **O**nline **B**attle **A**rena) in cui i vari personaggi interagiscono tra di loro per vincere la partita. In un contesto del genere è stato studiato come sviluppare entità autonome in grado di replicare le azioni dei giocatori online, e senza sorveglianza umana riescono a trovare soluzioni per combattere nel modo più efficace i nemici per distruggere la base avversaria.

2 Definizione di agente

Yoav Shoham[1] propone nel suo paper *Agent-oriented programming* una definizione di agente cercando di rispondere alla domanda:

What is an agent?

Un agente è un termine molto generale che può avere significati a volte diversi a seconda del contesto, ma in **AOP**(**A**gent **O**riented **P**rogramming) viene compreso al meglio quando accostato al concetto di entità *autonoma*.
Per autonomia[2] si intende qualcosa di indipendente, capace di portare a termine un'attività senza sorveglianza umana.
Gli agenti sono unità complete, che al loro interno contengono tutto il loro flusso di controllo e per questo motivo non ci si può accedere come succede nella programmazione ad oggetti. La differenza in questo caso è che non sono entità passive, ma proattive, cioè "fanno accadere qualcosa".
Un agente è **situato**: le proprie azioni non sono casuali, ma strettamente connesse con l'ambiente in cui opera e interagisce.
È necessario una rappresentazione del mondo circostante in modo da poter costruire il suo flusso di controllo in modo adeguato.
Da questo punto di vista si può osservare anche l'aspetto **reattivo**, ovvero la capacità di reagire ai cambiamenti dell'esterno. Un agente autonomo, situato in un certo ambiente, deve possedere dei meccanismi per poter percepire quello che accade attorno a lui per portare a compimento un' attività. Si può in questo caso fare l'esempio di un robot che si muove nello spazio, che per poter reagire alla presenza di un ostacolo o seguire un certo percorso deve poter essere in grado di

utilizzare i propri sensori, i cui dati dovranno alimentare il modello dell'azione che deve eseguire (come l'aggiramento dell'ostacolo, o seguire una curva).
Un agente puramente reattivo è un'entità che aspetta degli input dall'*enviroment*, senza che abbia un *goal* da dover portare a termine.
In ogni caso, il modello di un azione porta necessariamente ad un cambiamento, sia per quanto riguarda l'ambiente e gli altri agenti.
Per questo motivo un agente è anche **sociale**: un agente porta a termine un'azione, ma in un sistema complesso dove le operazioni sono tante, è necessaria la presenza di altri agenti e deve essere possibile poter creare delle modalità con cui interagiscono. L'ultima proprietà è proprio l'**interattività**, ovvero informazioni e conoscenze vengono scambiate tra gli agenti.

3 Agenti BDI

Un agente può essere costruito utilizzando l'approccio **BDI**: **B**eliefs, **D**esires e **I**ntentions[3].
L'architettura **BDI** ha l'obiettivo di descrivere un agente ad un livello più alto e logico, secondo una serie di concetti:

1. **Beliefs**. I **beliefs** corrispondono allo stato attuale dell'agente, con le informazioni in suo possesso riguardo ad altri agenti e all'abiemente in cui è situtato.

2. **Desires**. I **desires** sono più noti come i *goal*, ovvero gli obiettivi che deve portare a termine.

3. **Intentions**. Per *intentions* si intendono un sotto-insieme di goal a cui sono collegati altrettanti piani di azione per ottenerli.

4. **Eventi**. Gli **eventi** possono essere *esterni* e accadere attorno all'agente, dei cambiamenti nell'ambiente in cui sta operando e con cui interagisce. All'accadere di un evento è collegato un relativo piano di azione per gestire le nuove informazioni acquisite.

   Un altro tipo di evento è quello *interno*, associato a cambiamenti relativi ai **beliefs**, l'adozione o l'adempimento di un *goal*.

4 Agenti del Sistema e Enviroment

Introduzione Utilizzare un paradigma ad agenti (**AOP**) è molto utile nel caso di studio.
Ogni elemento è in stretta connessione con l'ambiente circostante, e non solo reagisce ai cambiamenti delle sue percezioni, indotti dall'azione di altri, ma partecipa in modo attivo, e le sue decisioni influiscono in modo sostanziale al sistema.
I minion devono capire in che modo possono battere i nemici, cercando di scegliere il bersaglio ideale per poter avanzare verso il quartier generale del nemico.
In questo caso è importante osservare quanto sia importante la percezione dell'ambiente e la comunicazione ai compagni di squadra.
Questa situazione è sicuramente adatta per usare il paradigma della programmazione ad agenti, e come scelta del *framework* si è deciso di adottare **Jason**.
Jason implementa la natura descrittiva di *AgentSpeak*, utilizzando un linguaggio specifico per l'architettura dell'agente, ed utilizzando Java per implementare logiche di basso livello.

Impostazione del MAS Un **M**ulti **A**gent **S**ystem è costituito dagli agenti e dall'ambiente in cui sono situati. Il progetto contiene un file, `mobasimulator.mas2j` in cui viene specificata l'infrastruttura del sistema, gli agenti iniziale e la classe che estende l'`Enviroment` di Jason.

```
MAS mobasimulator {

   infrastructure: Centralised
   environment: Arena
   agents:
      blueTeamNexus nexus;
      redTeamNexus nexus;
      blueTeamGaren champion;
      redTeamRiven champion;
      gameMaster master;
      blueTeamMinion minion #6;
      redTeamMinion minion #6;
   aslSourcePath:
      "src/asl";
}
```

Come scelta dell'infrastruttura si è scelto quella *centralizzata*, mentre la classe che estende l'ambiente e implementa logiche di basso livello è l'Arena (il campo di gioco). Dentro il package `src/asl` ci sono tutti i file dei vari tipi di agente, mentre sotto *agents* vengono specificati quelli iniziali: a sinistra il nome che compare sulla console mentre a destra il nome del file in cui si è descritto il funzionamento ad alto livello. Ispirandosi al famoso **MOBA** *League of Legends*, le squadre sono state divise in **RedTeam** e **BlueTeam**, ognuna incomincia con sei minion e il proprio campione.

4.1 GameMaster

Il **GameMaster** è l'agente che coordina l'andamento del gioco, si preoccupa di creare nuovi minion e campioni quando muoiono, informando poi quando la partita finisce al momento della distruzione del nexus (il quartiere generale).
L'agente interagisce con tutti gli altri e rimane neutrale durante l'azione delle squadre, assegnando i vari turni d'azione.

Beliefs I *beliefs* iniziali sono specificati nel file `master.asl`, come riportato:

```
currentChampionKillRedTeam(0).

currentChampionKillBlueTeam(0).

numberOfMinions(1).
```

I *beliefs* iniziali vengono usati per tenere traccia delle uccisioni dei campioni per entrambe le squadre, mentre l'ultimo è un *belief* che viene usato per creare attraverso un loop i minion di una squadra quando muoiono.

Plans L'agente ha come obiettivo quello di creare l'arena di gioco, registrare le morti dei campioni, creare nuovi agenti e assegnare i turni alle varie squadre in cui possono attaccare. Questi vengono attivati come reazione a eventi di tipo esterno, informato dagli altri agenti che usano l'azione

`.send(gameMaster,achieve,***nomedelpiano***)`

I piani che vengono attivati in questa modalità sono:

- `swapTurn(**nomedellasquadra**)`. Questo piano viene attivato quando tutti i giocatori hanno eseguito un'azione, assegnando il turno successivo alla squadra avversaria.
- `spawn(**nomedellasquadra**)`. Il piano viene chiamato al momento della morte di tutti gli elementi della squadra, e attraverso le funzioni fornite da Jason crea tutti i minion necessari con l'operazione `.create_agent(agentName)`.
- `updateChampionKill(**nomedellasquadra**)`. Il gameMaster aggiorna il contatore delle *kill* avvenute nella partita.

4.2 Minion

La definizione migliore per il minion è "carne da macello" e costituisce una prima protezione del campione da parte degli altri elementi. Il minion essenzialmente fa due cose: attaccare il nemico (altri minion, il campione o il nexus nemico) o reagire al danno proveniente dal campione o minion avversari.

Il minion è sostanzialmente di due tipi, il cui comportamento si differenzia a secondo del ruolo con cui è stato assegnato dall'*enviroment*:

1. **Melee**. Un minion che attacca corpo a corpo. Questo tipo di minion deve prima raggiungere il nemico e poi attaccarlo, e deve prima eliminare gli altri minion con lo stesso ruolo per poter passare ad altri bersagli. Per questo motivo possiede molta più vita degli altri minion, ma un attacco minore.

2. **Distance**. Questi minion possono essere pensati come gli arcieri medievali: stando sulla distanza non hanno bisogno di protezioni ma infliggono tanti danni a seconda del bersaglio che scelgono.

   Analogamente, con questo ruolo i minion possono colpire l'avversario che preferiscono, concentrandosi prima sui minion e poi sugli altri bersagli, infliggono più danno ma hanno meno vita.

Durante la partita riceve vari beliefs per sapere la sua squadra di appartenenza (`team(X)`), il suo ruolo (`role(X)`) ed è proprio con l'aggiunta di quest'ultimo che aggiorna i propri hitPoints: se è un ruolo di tipo *melee* i propri hitpoints vanno a 150, altrimenti 100. Altri *belief* che verranno descritti nella sottosezione arena attivano i *plans* con cui si scelgono il bersaglio e ricevono danno

Plans I minion percepiscono l'ambiente circostante e a seconda delle presenza o meno dei nemici eseguono un determinato tipo di azione.
Al momento del turno della propria squadra, attraverso l'*enviroment* al minion viene aggiunto un *belief* di tipo attacco. Se sono presenti i nemici, il minion deve attaccare prima quelli attivando il piano `!fight` in cui troviamo l'azione gestita dall'*enviroment*, `selectTarget` per la scelta del bersaglio. Se sia il campione che gli altri minion avversari sono già stati eliminati, il bersaglio è automaticamente il nexus nemico.
Il minion non solo attacca, ma percepisce anche il danno ricevuto. Quando riceve il danno, viene invece attivato il piano `!receiveDamage`, oppure il piano `!receiveDamageFromChampion` quando il danno proviene dal campione nemico. In entrambi i casi il minion deve prima aggiornare i propri *hitPoints*. se gli *hitPoints* sono maggiori del danno che riceve, allora è ancora vivo e può tornare a combattere aggiornandoli con la differenza dei due valori, altrimenti è morto e si uccide con l'azione `.kill_agent(**nomedell'agente**)`.

4.3 Champion

Il campione è l'eroe della squadra. Utilizza i minion per poter sconfiggere l'avversario e può utilizzare le sue abilità per infliggere ancora più danno: il campione possiede gli attacchi normali oppure può usare i suoi poteri che modificano il danno base per una percentuale, causandone di maggiori. Il campione cresce durante il gioco, aumentando il proprio livello a cui corrisponde un danno maggiore delle abilità.

Beliefs Il campione ha come *beliefs* iniziali i propri hitPoints e il livello, che vengono aggiornati durante le sue azioni. Il funzionamento del campione è simile a quello del minion, sia per quanto rigurda l'attacco che la recezione del danno. Come nel caso del minion, è attraverso l'enviroment Arena che vengono aggiunti altri *beliefs* per gestire l'attacco (`commenceAttack`), l'attacco al nexus (`attackNexus`) e la ricezione del danno (`damageFromEnemy`).

Plans I piani vengono attivati quando è aggiunto il *belief* corretto, differenziandosi da quello del minion perchè utilizza il proprio livello per modficare il danno che infligge.
Il bersaglio viene scelto dando prima priorità ai minion e solo successivamente al campione. Quando riceve danno, aggiorna i propri hitpoints e se muore, il campione avversario aumenta di livello. Prima di morire informa il gameMaster dell'accaduto per poi uccidersi sempre utilizzando `.kill_agent`.

4.4 Nexus

Il nexus è un agente puramente reattivo e non compie azioni se non quella di aggiornare i propri hitPoints quando viene colpito e nel caso sono a zero informa il gameMaster, fermando la partita.

4.5 Arena e interazioni con gli Agenti

L'Arena è l'enviroment che implementa le azioni degli agenti che hanno bisogno di conoscere quali agenti sono in vita e identificare quindi i bersagli. L'*Enviroment* è stato chiamato Arena proprio perchè è l'ambiente in cui operano i nostri giocatori,

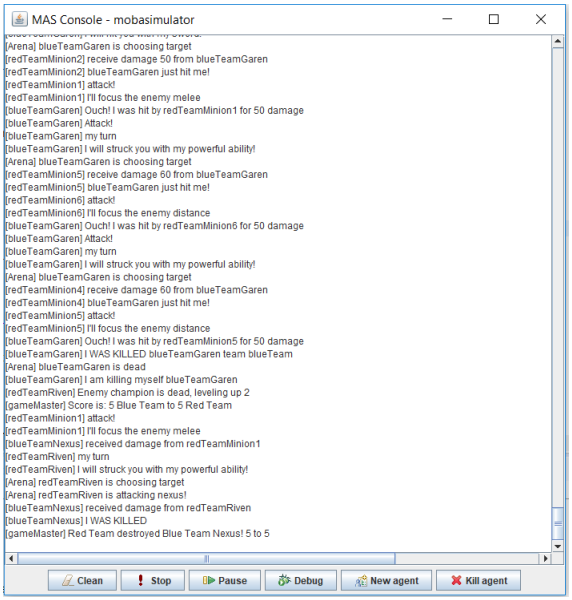

Figura 1: Esempio di una partita

il luogo dove combattono e dove sono situati. L'Arena svolge alcune funzioni molto importanti. Il **gameMaster** all'inizio esegue un piano `+!startGame`, che utilizza l'azione,`prepareArena`. Questa aggiunge vari *beliefs*:

1. `team(**nomedellasquadra**)`. Serve per poter capire di che squadra è l'agente, e li serve per coordinarsi con altri agenti. Una volta che ha eseguito un attacco, manda un messaggio al campione della propria squadra con l'obiettivo di eseguire un attacco, attivando il suo relativo piano

2. `role(**nomedelruolo**)`. Il ruolo del minion.

3. `self(**proprionome**)`. Viene usato per fornire il proprio nome all'azione `.kill_agent`.

Escludendo il *belief* `role`, gli altri vengono aggiunti ai campioni e ai nexus. Una volta preparata l'arena, il **gameMaster** usa l'azione `turn(**nomedellasquadra**)`. L'arena implementa l'azione aggiungendo ad un minion della squadra passata come parametro il *belief* di attacco `attack`. Il minion reagisce al cambiamento attivando il piano di attacco (descritto prima, `!fight`). Il minion interagisce con l'ambiente per ottenere il bersaglie tramite `selectTarget` che ha come parametro il suo ruolo.

L'Arena a seconda del ruolo, seleziona il bersaglio corretto aggiungendo al minion della squadra opposta il *belief* `damage`, che ha come primo parametro il valore del danno e come secondo parametro il nome del minion attaccante.
A questo punto, il campione della squadra che ha attaccato sceglie il proprio bersaglio quando viene aggiunto il *belief* `commenceAttack`, utilizzando una procedura simile a quella del minion: l'azione che implementa l'arena è `selectTart`, che da priorità ai minion, poi al campione nemico e, quando tutti sono morti, al nexus. Il campione può anche utilizzare le abilità, quindi il danno inflitto è variabile e viene passato come parametro.
Ogni volta che avviene un'uccisione, il gameMaster attraverso un'azione modifica l'ambiente con `updateKill`, e l'Arena aggiorna il numero di elementi di una squadra rimuovendo quello eliminato, in modo che le percezioni non siano alterate. Il gameMaster si occupa anche di ricreare gli elementi morti sul campo, e una volta ricreati viene aggiornato anche a livello dell'Arena.

5 Conclusioni

Problemi riscontrati Il linguaggio per descrivere il flusso di controllo degli agenti è di alto livello e molto intuitivo, e l'utilizzo di una classe di Enviroment è stata utile per capire come si comporterebbe un agente in un contesto reale, dove anzichè utilizzare *addPercept* ci sono gli input dei sensori.
Nonostante ciò, è stato sperimentato come essenzialmente unico problema il fatto che aggiungere più *beliefs* dello stesso tipo allo stesso agente può causare il blocco dell'esecuzione del piano corrispondente.
Per risolvere il problema, è stata costruita una funzione `updatePercepts` che rimuove i *belief* ogni volta che inizia un turno nuovo, permettendo un'esecuzione fluida.

Sviluppi futuri Il progetto è stato semplificato immaginando che ci sia una sola "lane", senza torrette. Sarebbe interessante aggiungere l'agente che gestisce le torrette, in grado di percepire gli agenti nemici circostanti e colpendoli come in un MOBA reale.
Altri miglioramenti possono essere l'aggiunta di altre lane, con altre squadre di campioni e minion.

Il campione può essere modificato aggiungendo varie caratteristiche come il *mana*, necessario per attivare le abilità.

Riferimenti bibliografici

[1] Yoav Shoham, "Agent-oriented programming", *Standford University*, 1992

[2] Andrea Omicini, "On Autonomy, Definitions Acceptations", *Alma Mater Studiorum Bologna*, 2018/2019

[3] Andrea Omicini, "Programming intentional agents", *Alma Mater Studiorum Bologna*, 2018/2019