ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

GAME ENGINE COME LAYER DI MODELLAZIONE DELL'AMBIENTE PER UN SISTEMA MULTI-AGENTE

Relazione per il corso:
Sistemi Autonomi

Studente:
LUCA PASCUCCI
Matricola:
763205

ANNO ACCADEMICO 2018–2019

Indice

Sommario

L'ambiente, o scena secondo la terminologia dei videogiochi, è la parte fondamentale di numerosi giochi dato che permette al giocatore di entrare in sinergia con il tipo, lo scopo e le modalità del gioco. Un esempio lampante è un FPS[1] dove la scena è solitamente vista in prima persona dal videogiocatore e la presenza di elementi con i quali interagire crea interesse nel giocatore ad esplorare l'ambiente attorno a lui.

In questa relazione si intende studiare lo stato di integrazione tra Game Engine (GE) e Sistemi Multi-Agente (MAS) con l'obiettivo di utilizzare la scena, realizzabile con una GE, come layer di modellazione dell'ambiente per il MAS. La prima fase approfondisce le astrazioni e funzionalità messe a disposizione nella GE, in particolare Unity, per realizzare, modellare ed interagire con l'ambiente. La seconda fase studia lo stato dell'arte delle integrazioni e identifica delle linee guida di collegamento delle astrazioni dei due sistemi. Per concludere viene definita la struttura del sistema per effettuare collegamento dei due sistemi, mantendoli separati e quindi senza modificare le loro astrazioni e funzionalità.

[1] First-person shooter = sparatutto in prima persona

Capitolo 1

L'ambiente in Unity

1.1 Game Engine

Le Game Engine (GE) sono framework utilizzati per supportare la progettazione e lo sviluppo di giochi. Il termine "Game Engine" nacque a metà degli anni '90 in riferimento a giochi sparatutto in prima persona (FPS) come il popolare "Doom" progettato con una separazione ragionevolmente ben definita tra i suoi componenti software principali (come il sistema di rendering grafico tridimensionale, il sistema di rilevamento delle collisioni o il sistema audio) e le risorse artistiche, i mondi di gioco e le regole di gioco che comprendevano l'esperienza di gioco del giocatore.

Le GE moderne sono strutture general-purpose multipiattaforma orientate verso ogni aspetto della progettazione e dello sviluppo del gioco, come il rendering 2D/3D delle scene di gioco, i motori fisici per la dinamica ambientale (movimenti, dinamica delle particelle, rilevamento delle collisioni, prevenzione degli ostacoli, ecc.), suoni, script comportamentali, intelligenza artificiale dei personaggi e molto altro.

Come esempio significativo che rappresenta la gamma di piattaforme disponibili, nella sezione successiva verrà esaminata una delle più popolari GE - Unity[17] - con l'obiettivo di:

- rilevare quelle astrazioni e quei meccanismi che hanno più probabilità di avere una controparte nel MAS, o almeno quelli che sembrano fornire un supporto nel riformulare le astrazioni mancanti del MAS;

- evidenziare le opportunità per colmare le lacune concettuali / tecniche che ostacolano l'integrazione dei due mondi.

1.2 Unity

Unity è una Game Engine (GE) cross-platform sviluppata da Unity Technologies, utilizzata per la creazione di videogiochi (sia 2D che 3D) e simulazioni, che supporta la distribuzione su una larga varietà di piattaforme (PC, console, dispositivi mobili, etc.). Fornisce astrazioni che contribuiscono ad estendere il suo utilizzo tra gli sviluppatori e programmatori, rendendola una delle GE più utilizzate per produrre in maniera veloce ed efficace applicazioni e giochi.[17]

Inoltre, questa GE supporta molte funzionalità facili da utilizzare e sfruttabili per creare giochi realistici e simulazioni immersive, come un intuitivo editor real-time, un sistema di fisica integrato, luci dinamiche, la possibilità di creare oggetti 2D e 3D (direttamente dall'IDE o di importarli esternamente), gli shader, un supporto per l'intelligenza artificiale (capacità di evitare gli ostacoli, ricerca del percorso, etc.), e cosi via.

Le funzionalità principali messe a disposizione del designer sono:

- **GameObject**: La classe base per tutte le entità presenti su una scena di Unity- un personaggio controllabile dall'utente, un personaggio non giocabile, un oggetto (2D/3D). Tutto ciò che è presente sulla scena è dunque un GameObject.

- **Script**: Il codice sorgente applicato a un GameObject, grazie al quale è possibile assegnare a quest'ultimo comportamenti e proprietà dinamiche. Gli script vengono eseguiti dal game loop di Unity che, in maniera sequenziale, esegue una volta ogni script durante ogni frame del gioco. Non esiste concorrenza. Il comportamento è il risultato della logica definita nello script attraverso funzioni e routine. Le proprietà equivalgono a variabili che possono essere valorizzare nello script oppure definite dall'IDE grafico.

- **Component**: Elemento, proprietà speciale assegnabile ai GameObject. A seconda del tipo di GameObject che si desidera creare è necessario aggiungere diverse combinazioni di Component. I Component

basilari riguardano la fisica (Transform, Collider,..), l'illuminazione (Light) e la renderizzazione del GameObject (Render). É possibile istanziare runtime Components attraverso gli script.

- **Coroutine**: Una soluzione alla sequenzialità imposta agli script, grazie al quale è possibile partizionare una computazione e distribuirla su più frame, sospendendo e riprendendo l'esecuzione in precisi punti del codice.

- **Prefab**: Rappresentazione di un GameObject complesso, completo di Script e Component, istanziabile più volte runtime. Le modifiche della struttura, le proprietà ed i componenti del Prefab si propagheranno a tutti i GameObject collegati allo stesso presenti nella scena di gioco.

- **Event e Messaging System**: sistema ad eventi utile per far comunicare tra loro diversi GameObject. Questi sistemi sono formati tipicamente da eventi e listener. I listener si sottoscrivono ad eventi di un certo tipo: quando l'evento si verifica, viene notificato a tutti i listener dello stesso tipo che sono in ascolto attraverso l'invio di un messaggio.

1.3 Realizzare un ambiente in Unity

Gli strumenti a disposizione in Unity mettono nelle mani degli sviluppatori la possibilità di realizzare qualunque tipo di oggetto: dai più semplici, come un cubo, ai più articolati, ad esempio un robot.

Per realizzare oggetti complessi è possibile creare una gerarchia di componenti e dotare ognuno di loro delle stesse funzionalità dell'oggetto padre. Ripensando all'esempio del robot, lo sviluppatore può suddividere lo stesso in sotto-componenti più articolate, quali testa, braccia, addome, gambe, fino ad arrivare a realizzare parti basilari quali dita, occhi e così via.

Successivamente alla realizzazione dello scheletro dell'oggetto, attraverso la definizione di "script" e l'associazione di "components" è possibile animarlo, dargli una specifica fisicità e renderlo consapevole dell'ambiente che lo circonda. Verranno ora fatti degli esempi per ognuna delle funzionalità appena descritte.

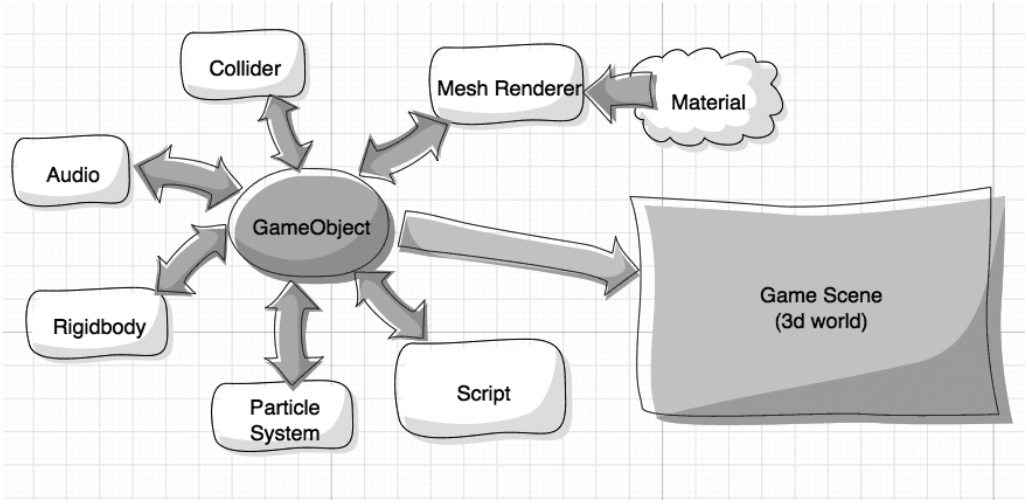

Figura 1.1: GameObject e Components

1.3.1 Muovere un GameObject

Come spiegato precedentemente, ogni oggetto presente su Unity è un GameObject e, per definizione, contiene le seguenti proprietà fondamentali:

- Posizione
- Scala
- Rotazione

Le proprietà appena elencate sono elementi fondamentali del "component" Transform [16], il quale viene automaticamente realizzato per ogni oggetto in scena.

Attraverso l'associazione con uno script, lo sviluppatore può accedere ad ogni proprietà del GameObject. In tal modo è possibile modificare runtime la sua posizione, la scala e la rotazione e, applicando diverse tipologie di trasformazioni, lo si anima. Questa modalità di animazione è basilare, ma sufficiente per l'obiettivo finale posto. Sono poi presenti meccanismi complessi in caso di elaborazioni più articolate e specifiche come, ad esempio, la simulazione della corsa umana.

1.3.2 Fisicità del GameObject

La semplicità nella realizzazione di oggetti all'interno delle Game Engine è affiancata alla presenza di un motore fisico: attraverso quest'ultimo, l'IDE elabora e modifica dinamicamente ogni oggetto in scena in base alle specifiche fisiche ad esso attribuite. Il motore fisico rende possibile la simulazione di forza di gravità, la fisica dei movimenti e le occlusioni della scena.

In Unity, per definire la fisicità di un GameObject, è necessario attribuirgli uno specifico "component" chiamato RigidBody [15]. Aggiungendo questo componente, il movimento del GameObject nella scena è controllato dal motore fisico di Unity. Di questo componente è possibile specificare:

- Massa
- Resistenza
- Velocità di movimento
- Soggezione alla gravità

1.3.3 Percepire l'ambiente

La percezione generalmente è associata all'acquisizione di una realtà interna o esterna attraverso l'elaborazione organica e psichica di stimoli sensoriali [19]. Nelle Game Engine, rendere un oggetto capace di percepire la realtà è spesso collegato a renderlo fisicamente consapevole della propria superficie.

Su Unity è presente il "Collision Detection System", il quale controlla ogni evento di interazione fisica tra due o più GameObject nella scena e, attraverso l'aggiunta del "component" Collider al GameObject, viene specificato che l'oggetto deve essere preso in considerazione dal sistema durante questi eventi.

Il Collider è associabile ad un GameObject, più o meno complesso, ed è capace di creare un'area generica, come ad esempio un cubo/sfera, che circonda l'oggetto, oppure mappare alla perfezione la sua superficie. Gli eventi di interazione creati dal "Collision Detection System" sono utilizzabili, da parte dello sviluppatore, nello script collegato al GameObject, difatti durante una collisione vengono invocati degli specifici metodi all'interno dello script con tutte le informazioni sull'evento. [14]

L'ultimo passaggio è fondamentale, dato che permette di completare il processo di percezione dell'ambiente da parte di un generico GameObject e, quindi, lo rende consapevole della propria presenza nella scena.

Per aumentare la capacità di percezione del GameObject, all'interno di Unity ogni oggetto può essere visto e utilizzato da ogni altro oggetto in scena. In questa maniera oltre alla percezione del proprio corpo fisico, il GameObject è in grado di conoscere anche quali altri elementi compongono la scena, ottenendo quindi la percezione totale dell'ambiente.

Tutte le procedure sopra illustrate sono replicabili per ogni GameObject presente in scena. In questo modo è possibile realizzare scene più o meno complesse.

1.3.4 Modificare l'ambiente

Il GameObject, come illustrato in precedenza, è la classe base di tutti gli oggetti presenti su una scena Unity, di conseguenza, l'ambiente stesso è un GameObject (più o meno complesso). Questo concetto, unito alla possibilità di ogni GameObject di interagire sia fisicamente che logicamente (script) con ogni altro oggetto in scena, dà luogo ad infinite possibilità di modifica della scena. Ad esempio, in caso di collisione tra due oggetti, il motore fisico, unito al motore grafico, calcola il possibile spostamento degli stessi che quindi porta ad un'effettiva modifica dell'ambiente.

1.4 Unity come ambiente in un MAS

L'ambiente all'interno di un MAS è catalogato come "contenitore" in cui gli agenti sono immersi e con il quale questi ultimi possono interagire, modificandolo. La caratteristica degli agenti di essere situati nell'ambiente in cui si trovano permette loro di percepire e produrre cambiamenti su di esso. Con le procedure sopra descritte è dunque possibile modellare l'ambiente da un punto di vista fisico e, quindi, dare un corpo agli agenti e se necessario anche agli artefatti. Di conseguenza, possedendo un corpo fisico (GameObject) ogni agente e/o artefatto è effettivamente in grado di produrre cambiamenti nella scena e quindi modificare l'ambiente.

Capitolo 2

Background

2.1 Integrazione

Esistono già esempi di integrazione tra GE e MAS che concentrano la propria attenzione su obiettivi specifici a livello tecnologico, piuttosto che sulla creazione di un'infrastruttura orientata agli agenti basata sul gioco per scopi generici. Nello specifico:

- QuizMASter [2] si concentra sull'astrazione degli agenti collegando i MAS ai personaggi dei motori di gioco, nel contesto dell'apprendimento educativo;
- CIGA [18] considera sia la modellazione degli agenti che quella dell'ambiente, per agenti virtuali generici in ambienti virtuali;
- GameBots [8] si concentra sull'astrazione dell'agente, ma considera ancora l'ambiente fornendo al contempo sia un framework di sviluppo sia un runtime per i test di sistemi multi-agente in ambienti virtuali;
- UTSAF [12] si focalizza sulla modellistica ambientale nel contesto di simulazioni distribuite in ambito militare[1]

Sebbene rappresentino chiaramente esempi di integrazione (parzialmente) riuscita di MAS in GE, i lavori sopra elencati presentano alcune carenze rispetto all'obiettivo che perseguiamo in questo documento.

[1] Gli agenti vengono considerati, ma solo come mezzo di integrazione tra diverse piattaforme di simulazione, non nel contesto del GE sfruttato per il rendering di simulazione.

Solamente CIGA fa eccezione poichè riconosce il divario concettuale tra MAS e GE e propone soluzioni per affrontarlo (anche se a livello tecnologico). L'unico strato preso in considerazione nel perseguimento dell'integrazione è quello tecnologico - nessun modello, nessuna architettura, nessun linguaggio.
All'interno di QuizMASter, UTSAF e GameBot (in una certa misura) l'integrazione è specificamente realizzata per obiettivo, dove la maggior parte degli approcci fornisce ai programmatori alcune astrazioni per trattare con agenti e ambiente, nessuna attenzione viene data alle astrazioni sociali.[9]

2.2 Situazione iniziale

I lavori precedentemente svolti, che hanno contribuito alla definizione di questo percorso, utilizzano due approcci nettamente separati per l'integrazione MAS e GE:

1. Integrazione delle caratteristiche dei MAS all'interno della GE;
2. Realizzazione di un middleware, come layer software, per collegare l'ambiente MAS con GE.

2.2.1 MAS all'interno di GE

Il primo punto è stato realizzato implementando due modelli tipici dei MAS:

- Il modello Beliefs, Desires, Intentions (BDI) per la programmazione degli agenti [11];
- Un modello di coordinazione degli agenti tramite spazio di tuple e primitive Linda [5][1];

Il cuore pulsante di entrambi i lavori risiede nell'uso intensivo di un interprete Prolog fatto ad hoc per Unity: UnityProlog [7]. Questo interprete dispone di molte funzionalità per estendere l'interoperabilità di Prolog con i GameObject. Dal momento che è stato progettato per essere usato in maniera specifica con Unity, nasce con delle primitive che permettono di accedere e manipolare GameObject e i relativi componenti direttamente da Prolog. UnityProlog introduce tuttavia alcune limitazioni da tenere bene in

considerazione [11], anche se allo stato attuale è l'unica versione di Prolog del quale è stato dimostrato il corretto funzionamento:

- Un interprete per Prolog non sarà mai performante quanto lo può essere un compilatore e questo può rappresentare un problema per simulazioni di MAS piu grandi.

- Utilizza lo stack C# come stack di esecuzione, quindi la tail call optimization non è ancora supportata.

- Non supporta regole con più di 10 subgoal, quindi a fronte di una regola complessa con tanti goal da controllare, è necessario frammentare la regola in questione in sotto regole con non più di 10 subgoal per ognuna.

2.2.2 MAS e GE separati

Il secondo percorso si differenzia dal primo per la scelta di lasciare separati GE da MAS realizzando un canale di comunicazione tra i due ambienti. É stata introdotta una terminologia per contraddistinguere le entità realizzate sul GE (GameObject) e su MAS (agenti), rispettivamente definite "corpi" e "menti" virtuali. [6]

Fondamentalmente un corpo deve eseguire azioni e a seguito di determinati eventi deve trasmettere le proprie percezioni alla mente, mentre la mente deve elaborare le percezioni per decidere quali azioni deve far svolgere al proprio corpo. Per rendere possibile questa comunicazione è stato progettato e implementato un sistema middleware definito secondo il seguente schema.

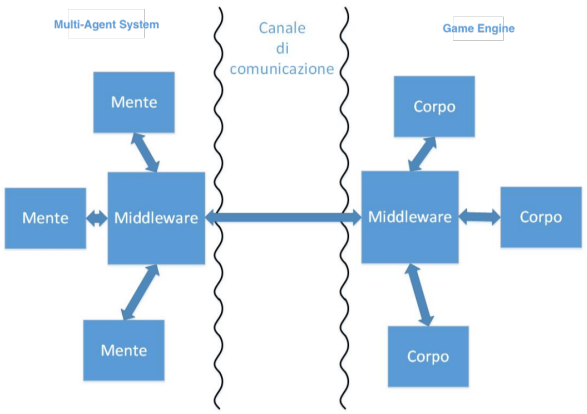

Figura 2.1: Il middleware viene suddiviso in due parti, poste sui due lati del canale di comunicazione. [6]

Dallo schema si può notare la separazione del middleware nei due sistemi, motivato dalle diverse tecnologie utilizzate dai due ambienti. Questa divisione vincola la realizzazione di una nuova parte di middleware in caso di utilizzo di un diversa tipologia di GE e/o MAS.

Il protocollo di comunicazione tra le entità è stato realizzato utilizzando messaggi strutturati. Da una parte, le menti devono definire quale azione deve compiere il relativo corpo (es. "muoviti in avanti", "ruota", "prendi", ecc.), dall'altro i corpi devono far sapere alle relative menti le proprie percezioni dell'ambiente circostante (es. "mi ha toccato un'entità", "sono alle coordinate 23,12,-6", ecc.).[6]

2.3 JaCaMo

JaCaMo è un framework per la programmazione orientata agli agenti che combina tre tecnologie già affermate e sviluppate da diversi anni. Come ogni MAS, JaCamo fornisce agli sviluppatori e ai designer tre astrazioni principali:

- Agenti: Le entità autonome che compongono il sistema. Sono in grado di comunicare e possono essere intelligenti, dinamici, e situati;

- Società: Rappresenta un gruppo di entità il cui comportamento emerge dall'interazione tra i singoli elementi;

- Ambiente: Il "contenitore" in cui gli agenti sono immersi e con il quale questi ultimi possono interagire apportandogli modifiche. La caratteristica degli agenti di essere situati nell'ambiente in cui si trovano permette loro di percepire e produrre cambiamenti su di esso.

Un sistema multi-agente JaCaMo o, equivalentemente, un sistema software programmato con JaCaMo è definito da un'organizzazione Moise di agenti BDI autonomi basati su concetti come ruoli, gruppi, missione e schemi, implementati tramite Jason, che lavorano in ambienti condivisi distribuiti basati su artefatti, programmati in CArtAgO.

Ognuna delle tre tecnologie indipendenti che compongono il framework ha il proprio set di astrazioni, modelli di programmazione e meta-modelli di riferimento; per questo motivo in JaCaMo è stato realizzato un meta-modello globale avente l'obiettivo di definire dipendenze, connessioni, mapping concettuali e sinergie tra le differenti astrazioni rese disponibili da ogni livello.[3]

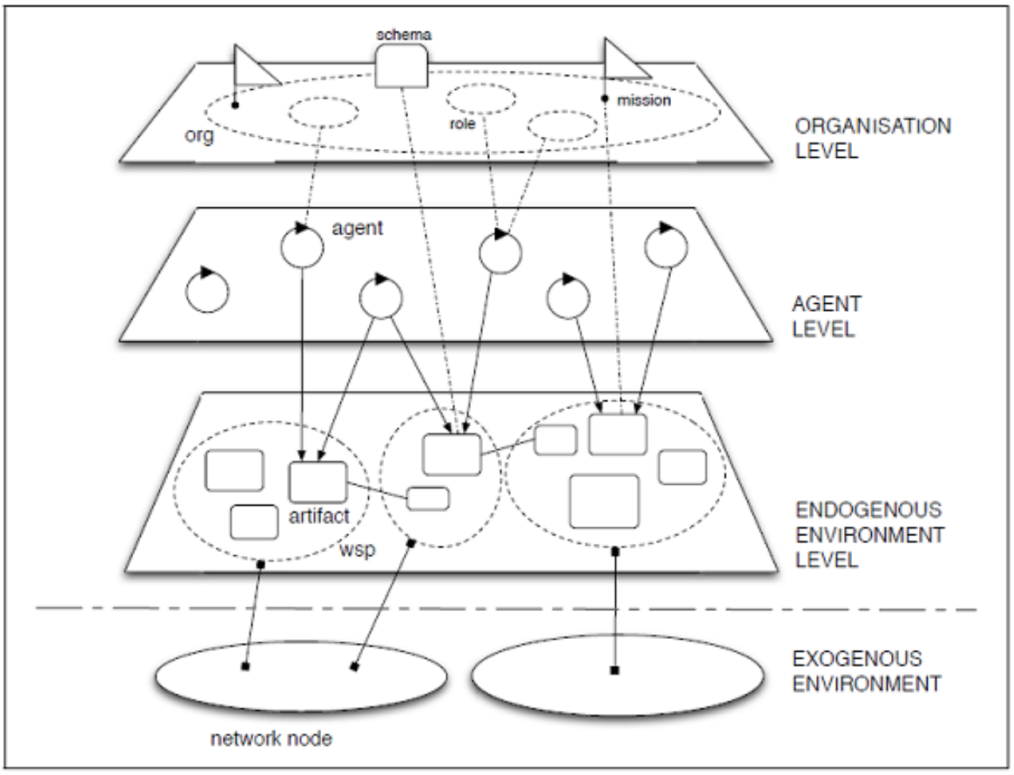

Figura 2.2: Livelli che compongono il framework JaCaMo[3]

2.3.1 Jason

Jason è un interprete di AgentSpeak che implementa la semantica operazionale del linguaggio e fornisce una piattaforma di sviluppo per multi-agent systems con molte funzionalità user-customisable.

Le astrazioni appartenenti agli agenti, correlate al meta-modello Jason, sono principalmente ispirate all'architettura BDI sulla quale Jason è radicato. Quindi un agente è un'entità composta da un insieme di "beliefs" che rappresenta lo stato corrente e la conoscenza dell'agente sull'ambiente in cui si trova; una serie di "goals", che corrispondono a compiti che l'agente deve eseguire / ottenere e una serie di "plans", ossia azioni (external action or internal action) innescate da eventi, che gli agenti possono comporre, istanziare ed eseguire dinamicamente per compiere i "goals".[4]

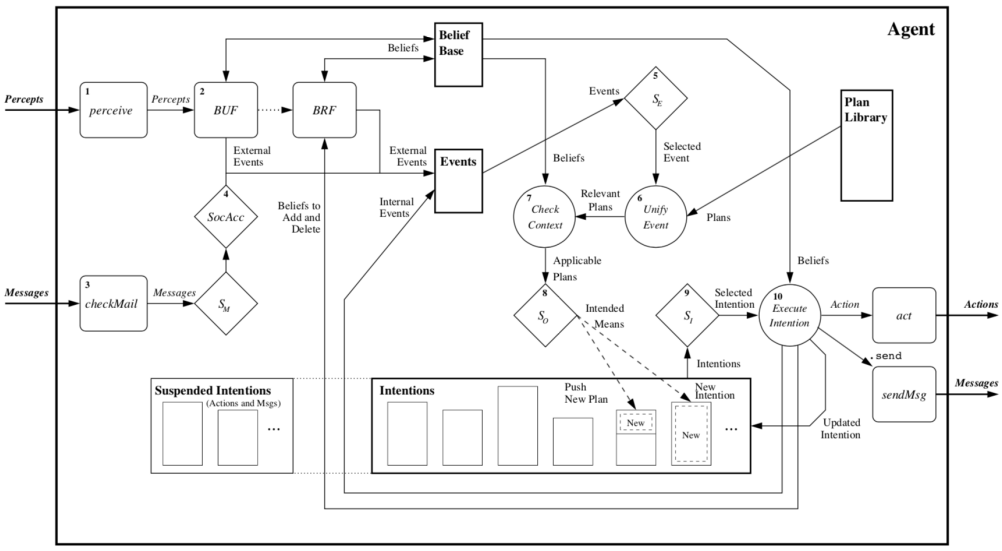

Figura 2.3: Architettura BDI di un agente[4]

Durante l'analisi delle astrazioni disponibili su Jason è risultato particolarmente utile, al fine di integrazione con la Game Engine, il concetto di "beliefs".

La Belief Base, raffigurata nell'immagine soprastante, è il contenitore della conoscenza dell'agente che viene in parte modificata dalle percezioni esterne che l'agente riceve dall'ambiente. Tali percezioni sono facilmente riconducibili alle percezioni / eventi che un GameObject può ricevere durante la sua permanenza nella scena (spiegato nella sezione 1.3), come ad esempio la notifica di una collisione con un secondo GameObject in scena.

Questo concetto ha portato alla decisione di utilizzare il belief come strumento di "notifica" delle percezioni ricevute dal GameObject su Unity, con lo scopo di integrare i due sistemi senza effettuare modifiche su concetti e astrazioni di MAS e Game Engine.

2.3.2 CArtAgo

Ciascuna istanza dell'ambiente CArtAgO[2] è composta da una o più entità workspace. Ogni workspace è formato da un insieme di artefatti, che

[2]Common ARTifact infrastructure for AGents Open environments

forniscono un insieme di operazioni e proprietà osservabili, definendo anche l'interfaccia di utilizzo degli artefatti. L'esecuzione dell'operazione potrebbe generare aggiornamenti delle proprietà osservabili e degli eventi osservabili specifici. L'ultima entità relativa all'ambiente è il "manual", un'entità utilizzata per descrivere le funzionalità fornite da un artefatto. Cartago è basato sul meta-modello A&A (Agents & Artifacts) gli agenti come delle entità computazionali compiono qualche tipo di attività che mira a uno scopo e, gli artefatti come risorse e strumenti costruiti dinamicamente, sono usati e manipolati dagli agenti per supportare/realizzare le loro attività.[13]

Artefatto L'artefatto è un'entità reattiva, non autonoma, stateful, riutilizzabile, controllabile ed osservabile. Modella strumenti, risorse e porzioni di ambiente agendo da strumento mediatore di azioni e interazioni sociali tra partecipanti individuali e lo stesso ambiente.[10]

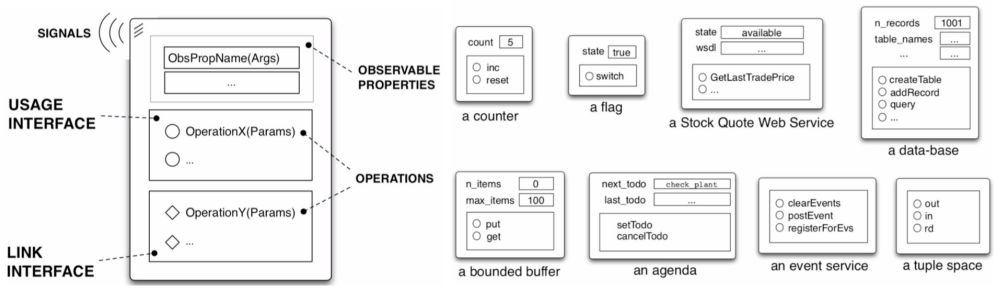

Figura 2.4: Struttura artefatto con relativi esempi

Durante l'analisi delle astrazioni disponibili su CArtAgo sono risultate utili, al fine di integrazione con un Game Engine, i concetti di artefatto, "Observable Properties" e di "Operations". L'immagine sottostante contiene un esempio di interazione e utilizzo tra agente ed artefatto dove viene fatto uso dei concetti precedentemente elencati.

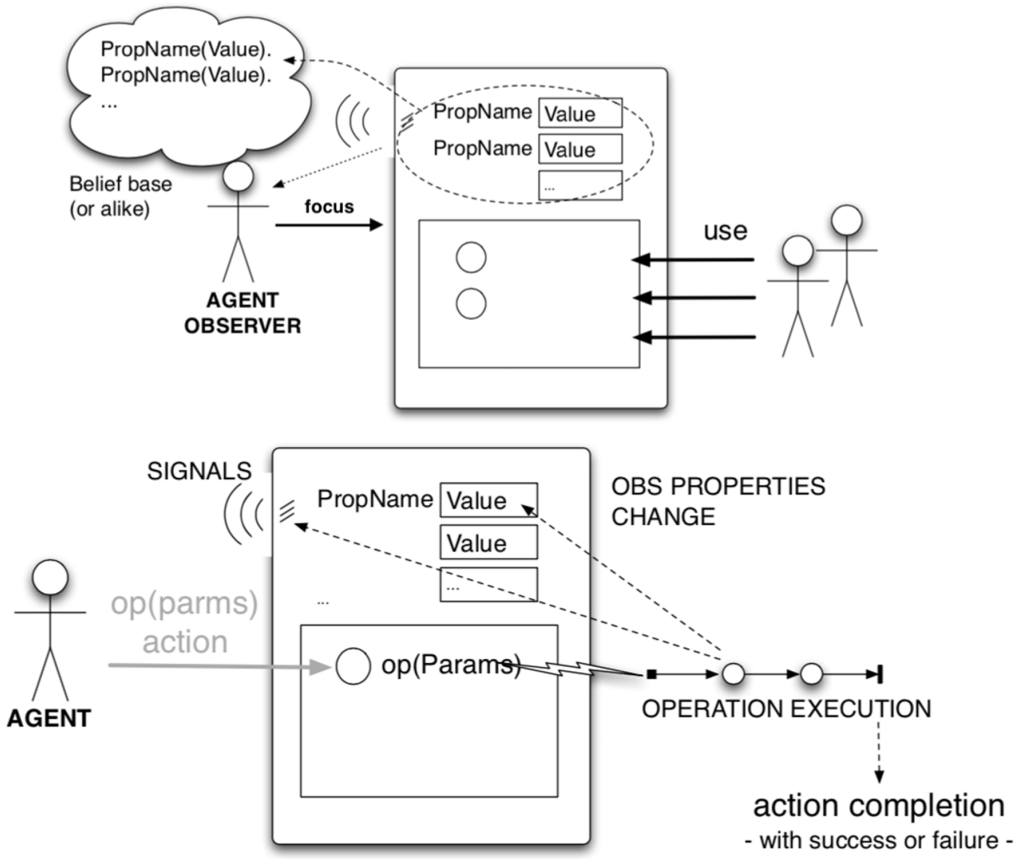

Figura 2.5: Interazione ed utilizzo tra agente ed artefatto

La definizione di artefatto ha contribuito a definire la principale modalità di integrazione tra MAS e Game Engine dato che le sue finalità sono collegabili alle finalità di utilizzo del GameObject su Unity. Entrambi devono rappresentare una porzione di ambiente, risultare uno strumento a disposizione dell'agente per effettuare operazioni/azioni sull'ambiente e "notificare" l'agente in caso di modifiche sulle informazioni contenute. Per questo motivo è stato deciso di effettuare un collegamento univoco tra GameObject e Artefatto con un rapporto di 1:1.

Questa modalità permette quindi ad un agente di utilizzare il proprio artefatto come se fosse il proprio GameObject su Unity e, quindi, di effettuare azioni- utilizzando le "Operations" dell'artefatto- e ricevere informazioni

sullo stato del proprio corpo fisico, attraverso le "Observable Properties", senza modificare concetti e astrazioni presenti nel MAS.

2.3.3 Moise

Moise è un meta-modello organizzativo per MAS basato sulle nozioni di ruoli, gruppi e missioni. Abilita un MAS ad avere specifiche esplicite per la sua organizzazione. Queste specifiche sono usate sia dagli agenti, per ragioni inerenti la loro organizzazione, sia da una piattaforma organizzativa che si assicuri che gli agenti seguano le specifiche.

Moise consente di definire una gerarchia di ruoli con autorizzazioni e missioni da assegnare agli agenti. Questo permette, ai sistemi con un'organizzazione forte, di guadagnare proprietà di apertura (essenzialmente, la proprietà di lavorare con un numero e una diversità di componenti che non è imposta una volta per tutte) e adattamento.

Capitolo 3

Synapsis

3.1 Terminologia

La sinapsi (o giunzione sinaptica) (dal greco synàptein, vale a dire "connettere") è una struttura altamente specializzata che consente la comunicazione delle cellule del tessuto nervoso tra loro (neuroni) o con altre cellule (cellule muscolari, sensoriali). Nello specifico la sinapsi neuromuscolare rappresenta la giunzione tra neurone motore e muscolo a livello della placca motrice, ove ha luogo la trasmissione dell'impulso con le modalità delle sinapsi chimiche: lo spazio extracellulare della sinapsi neuromuscolare è detto chiave sinaptica.[19] La semplice associazione tra l'obiettivo di questo percorso e la parola sopra definita ha portato a denominare il middleware "Synapsis"[1].

3.1.1 Entità

Successivamente nella trattazione verrà fatto uso del termine "entità" che, generalmente, viene intesa come insieme di elementi dotati di proprietà comuni dal punto di vista dell'applicazione considerata.[19] Concettualmente, in questo dominio, l'entità viene intesa come oggetto divisibile in due partimente e corpo- che, collegate, riescono a trasmettersi informazioni utilizzate dalla mente per raggiungere i propri obiettivi e dal corpo per diventare "attivo" nell'ambiente in cui si trova.

[1] Traduzione in inglese del termine italiano sinapsi.

3.1.2 Mente

La nozione di mente può essere caratterizzata da alcuni punti chiave fondamentali:

- autonomia
- interazione
- obiettivi

In altre parole, una mente può essere pensata come un componente software autonomo che interagisce con l'ambiente per svolgere i propri compiti. I punti sopra elencati rendono facile l'associazione della mente al concetto di Agente, spiegato nella sezione 2.3.1, poiché questa entità del Sistema Multi-Agente (MAS) ingloba astrazioni simili a quelle illustrate nella sezione 2.3.1.

3.1.3 Corpo

Corpo è un termine generico che indica qualsiasi porzione limitata di materia, cui si attribuiscono, in fisica, le proprietà di estensione, divisibilità, impenetrabilità.[19] In questa trattazione è associabile alla nozione di GameObject di Unity, spiegata nella sezione 1.2, utilizzata per avere una rappresentazione fisica dell'entità da realizzare.

3.1.4 Azione

Nel suo significato più generale è intesa come attività od operazione posta in essere da un determinato soggetto.[19] In questo studio, si considera come "azione" un certo gesto richiesto dalla mente che può essere associato ad un'operazione eseguita dal corpo, ad esempio, nel caso di un'azione del tipo *"vai a (posizione)"*, richiesta dalla mente, corrisponde il movimento del corpo nell'ambiente verso la posizione indicata.

3.1.5 Percezione

La percezione è un atto cognitivo mediato dai sensi con cui si avverte la realtà di un determinato oggetto e che implica un processo di organizzazione e interpretazione.[19].

In questo studio, la percezione si collega ad una certa sensazione rilevata dal corpo ed inviata alla mente per portarla a conoscenza di questa nuova informazione, ad esempio, nel caso del raggiungimento della posizione richiesta in precedenza, il corpo trasmette la percezione *"arrivato (posizione)* che informa la mente del completamento dell'operazione.

Esiste inoltre, da parte del corpo, la possibilità di inviare percezioni "libere", ossia non associate a risposta di un'azione inviata dalla mente. Un semplice esempio è il contatto del corpo con una qualsiasi altra entità nell'ambiente che corrisponde all'invio di una percezione del tipo *"toccato (nome_ entità)"*.

3.1.6 Struttura entità

Nella figura sottostante viene rappresentata la struttura di una generica entità, dove:

- Il corpo esegue azioni e, in risposta a queste ultime, oppure, a seguito di determinati eventi esterni, trasmette le proprie percezioni alla mente.

- La mente elabora le percezioni per decidere quali azioni far svolgere al proprio corpo.

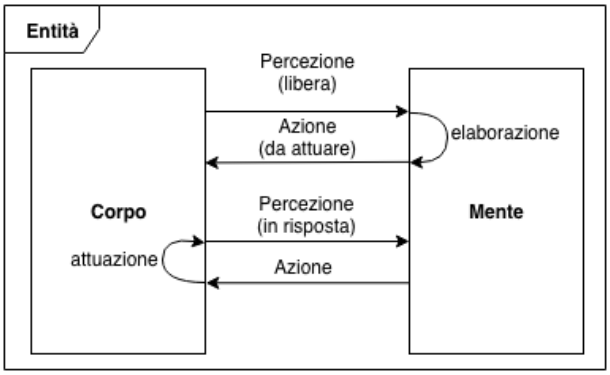

Figura 3.1: Struttura di una generica entità

3.2 Architettura

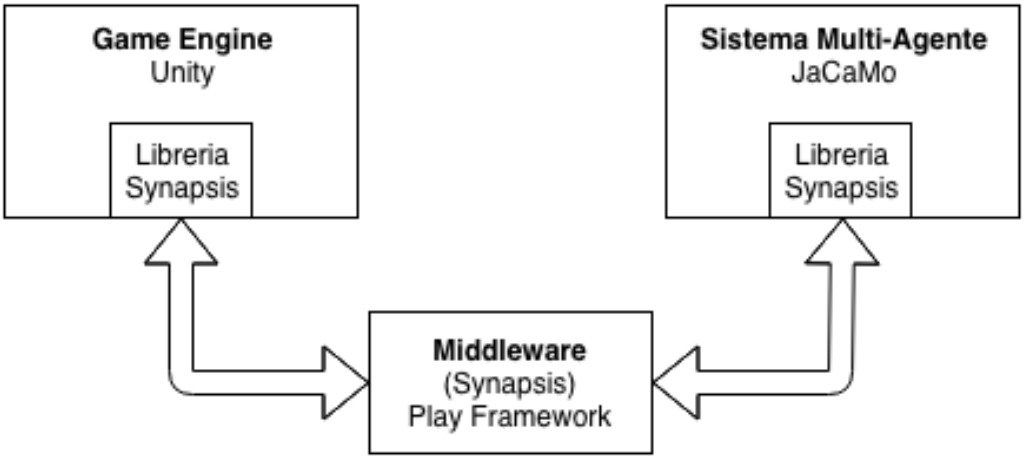

Figura 3.2: Architettura ad alto livello

L'immagine mostra la struttura del sistema realizzato per mettere in comunicazione MAS e GE attraverso l'introduzione di un middleware, realizzato con il framework Play, separato e autonomo rispetto alle differenti tecnologie utilzzate su MAS e GE.

Per collegare al middleware i due sistemi, sono state realizzate due librerie, definite nell'immagine sovrastante "Libreria Synapsis", contenenti funzionalità di collegamento e comunicazione con Synapsis. Le librerie rispettano astrazioni e modelli computazionali di entrambi i sistemi (MAS e GE) ed utilizzano la terminologia precedentemente elencata.

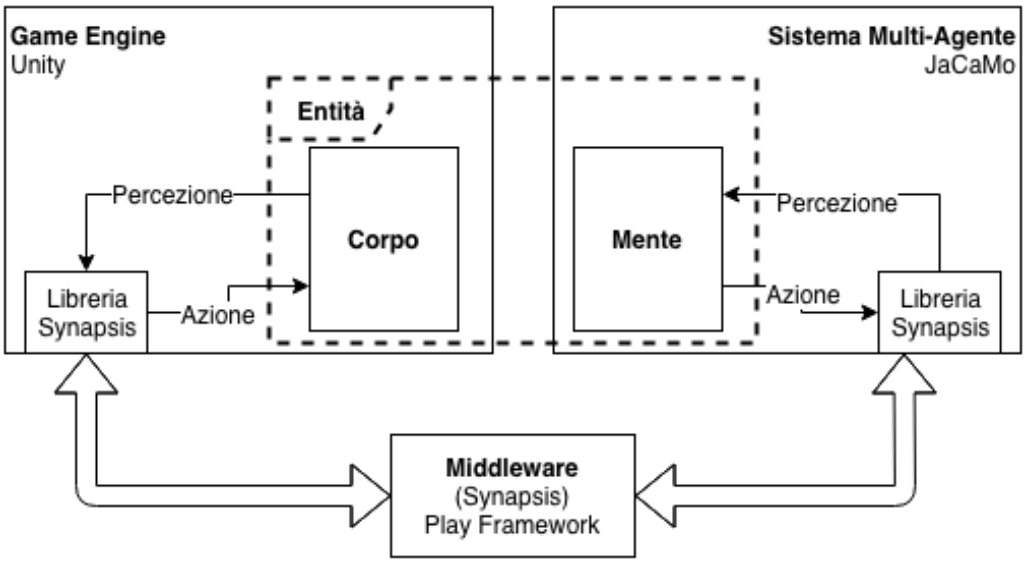

Figura 3.3: Divisione di un'entità nel sistema

Aggiungendo un'entità alla struttura precedente è possibile notare come la stessa risulti suddivisa tra i due sistemi (MAS e GE) ed, attraverso il collegamento al middleware, venga reso possibile lo scambio di informazioni (percezioni, azioni) pur essendo computazionalmente separate.

3.3 Scenario d'esempio

Per meglio comprendere l'architettura di sistema appena descritta e le interazioni tra i suoi componenti costituenti, si prende a riferimento uno scenario d'esempio chiamato "recycling robots". Come si può intuire dal nome, la scena contiene dei robot, i quali hanno il compito di riciclare la spazzatura presente nell'ambiente portandola in un bidone.

Il compito generale di un robot è divisibile in un ciclo di sotto-obiettivi, ad esempio:

1. Cercare la spazzatura;
2. Andare verso la spazzatura trovata;
3. Prendere la spazzatura appena raggiunta;

4. Cercare il bidone

5. Andare verso il bidone trovato

6. Riciclare la spazzatura

Utilizzando il formalismo di Jason, i sotto-obiettivi elencati sono associabili a dei *"plans"* di un agente. Al di fuori del primo plan (Cercare la spazzatura), normalmente attivato dal *"goal"* principale presente nell'agente, i successivi possono essere attivati da una percezione ricevuta dall'agente. Ad esempio, in risposta alla ricerca del bidone è possibile notificare la scoperta di un bidone nelle vicinanze e, di conseguenza, fare in modo che l'agente utilizzi il plan "Andare verso il bidone trovato".

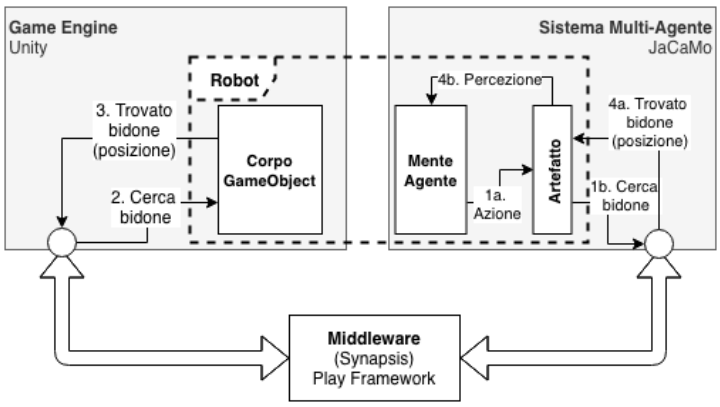

Figura 3.4: Esempio di comunicazione tra mente e corpo

L'immagine mostra il flusso ordinato di interazioni per l'esempio appena descritto. La mente per svolgere il plan *"Cercare il bidone"* vuole inviare al proprio corpo l'azione *"Cerca bidone"*. La richiesta di svolgere l'azione inizia dall'utilizzo dell'operazione presente nell'artefatto personale dell'agente[2], in possesso del canale per comunicare con il middleware.

[2]previa associazione dei due

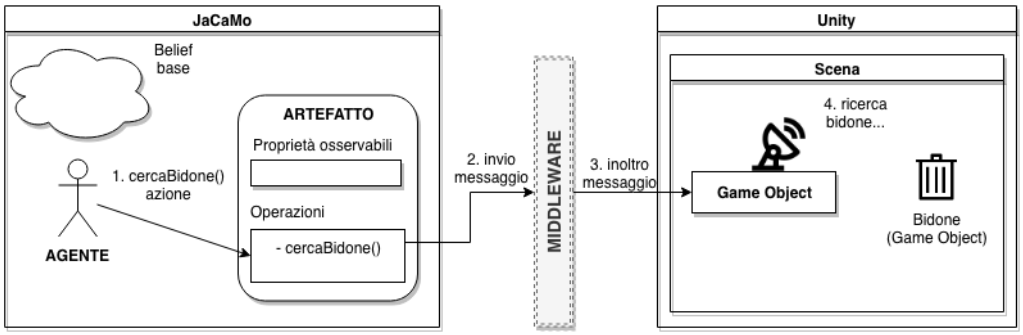

Figura 3.5: Fase in invio azione a GameObject

L'artefatto quindi invia il messaggio al middleware che si occupa di inoltrare le informazioni al Game Object. Alla ricezione delle stesse, l'entità corpo (GameObject) attua l'azione richiesta e risponde alla mente (Agente) inviandogli la percezione generata, ad esempio *"trovato(nomeBidone,posizione)"*.

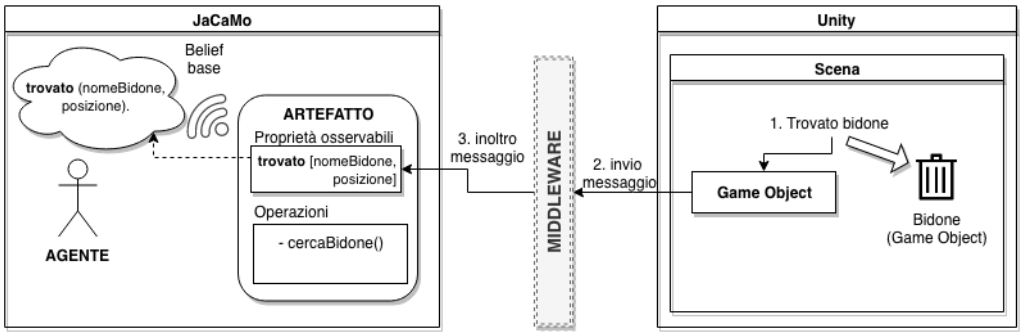

Figura 3.6: Fase di invio percezione ad Agente

A questo punto la percezione viene mandata al middleware che, a sua volta, la inoltrerà all'artefatto collegato. L'artefatto, nel momento in cui riceve la percezione, aggiunge quest'ultima alle sue proprietà osservabili che, automaticamente, aggiorneranno la BeliefBase dell'agente. L'ultimo passaggio rappresenta il punto cruciale per completare il collegamento tra corpo e mente dato che in questa maniera l'agente ha ricevuto la percezione dal proprio corpo.

Si intende inoltre lasciare aperta la possibilità, da parte del corpo, di inviare percezioni non come reazione ad azioni eseguite dalla mente, dato

che il collegamento WebSocket, una volta effettuato, rimane attivo per tutta la durata di vita dell'entità. Ad esempio, in caso di contatto con un oggetto nella scena Unity, il corpo deve essere in grado di mandare una percezione del tipo *"toccata(nome_ oggetto,posizione)"* alla propria mente senza bisogno di stimoli.

Capitolo 4

Conclusioni

Lo studio effettuato ha identificato le linee guida per collegare le astrazioni di Game Engine (GE) con quelle del Sistema Multi-Agente (MAS) con l'obiettivo di utilizzare la scena della GE come layer di modellazione dell'ambiente per il MAS. La definizione di un middleware (Synapsis) di collegamento ha permesso di non effettuare modifiche sostanziali sulle astrazioni dei due sistemi.

I successivi passi comprenderanno la ricerca della tecnologia per realizzare il middleware, della tecnologia per collegare le tre componenti (GE, MAS, Synapsis) del sitema delineato e realizzare le librerie di supporto per GE e MAS, tenendo conto dei loro diversi modelli computazionali.

Elenco delle figure

Bibliografia

[1] A. Bagnoli. *Game Engines e MAS: Spatial Tuples in Unity3D*. PhD thesis, Alma Mater Studiorum - Università di Bologna, 2018.

[2] J. Blair and F. Lin. An approach for integrating 3d virtual worlds with multiagent systems. In *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*, March 2011.

[3] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi. Multi-agent oriented programming with jacamo. *Science of Computer Programming*, 78(6), 2013. Special section: The Programming Languages track at the 26th ACM Symposium on Applied Computing (SAC 2011) & Special section on Agent-oriented Design Methods and Programming Techniques for Distributed Computing in Dynamic and Complex Environments.

[4] R. H. Bordini, H. J. Fred., and M. J. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley, 2007.

[5] M. Cerbara. *Stato dell'arte della progettazione automatica di programmi per robot*. PhD thesis, Alma Mater Studiorum - Università di Bologna, 2017.

[6] M. Fuschini. *Tecnologie ad Agenti per Piattaforme di Gaming: un caso di studio basato su JaCaMo e Unity*. PhD thesis, Alma Mater Studiorum - Università di Bologna, 2017.

[7] I. Horswill. UnityProlog: A mostly ISO-compliant Prolog interpreter for Unity3D. https://github.com/ianhorswill/UnityProlog, Aug. 2015.

[8] G. A. Kaminka, M. M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A. N. Marshall, A. Scholer, and S. Tejada. Gamebots: a flexible test bed for multiagent team research. *Communications of the ACM*, 45(1), 2002.

[9] S. Mariani and A. Omicini. Game engines to model MAS: A research roadmap. In C. Santoro, F. Messina, and M. De Benedetti, editors, *WOA 2016 – 17th Workshop "From Objects to Agents"*, volume 1664 of *CEUR Workshop Proceedings*, pages 106–111. Sun SITE Central Europe, RWTH Aachen University, 29–30 July 2016. Proceedings of the 17th Workshop "From Objects to Agents" co-located with 18th European Agent Systems Summer School (EASSS 2016).

[10] A. Omicini, A. Ricci, and M. Viroli. Artifacts in the a&a meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3), Dec 2008.

[11] N. Poli. *Game Engines and MAS: BDI & Artifacts in Unity.* PhD thesis, Alma Mater Studiorum - Università di Bologna, 2017.

[12] P. Prasithsangaree, J. Manojlovich, S. Hughes, and M. Lewis. Utsaf: A multi-agent-based software bridge for interoperability between distributed military and commercial gaming simulation. *SIMULATION*, 80(12), 2004.

[13] A. Ricci, M. Viroli, and A. Omicini. Cartago: A framework for prototyping artifact-based environments in mas. In D. Weyns, H. V. D. Parunak, and F. Michel, editors, *Environments for Multi-Agent Systems III*, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[14] U. Technologies. Colliders. https://docs.unity3d.com/Manual/CollidersOverview.html, 2019.

[15] U. Technologies. Rigidbody. https://docs.unity3d.com/ScriptReference/Rigidbody.html, 2019.

[16] U. Technologies. Transform. https://docs.unity3d.com/ScriptReference/Transform.html, 2019.

[17] U. Technologies. Unity. https://unity.com/, 2019.

[18] J. van Oijen, L. Vanhée, and F. Dignum. Ciga: A middleware for intelligent agents in virtual environments. In M. Beer, C. Brom, F. Dignum, and V.-W. Soo, editors, *Agents for Educational Games and Simulations*, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[19] Vocabolario Treccani. Treccani. http://www.treccani.it/vocabolario/.