Game Engines and Multi-Agent Systems: a Marriage?

Nicola Poli, Mattia Cerbara

Alma Mater Studiorum – Università di Bologna
Scuola di Ingegneria e Architettura - Cesena
via Sacchi, 3, 47023 Cesena, Italy

nicola.poli2@studio.unibo.it
mattia.cerbara@studio.unibo.it

Indice

1 Introduzione

La necessità di entità indipendenti, autonome, nei videogiochi moderni è diventata ormai basilare. È difficile pensare a un videogioco che non implementi qualche forma di intelligenza gestita interamente dal computer.
Un Agente come astrazione nei *MAS*, rappresenta un'entità autonoma e proattiva e nei videogiochi moderni, modellare entità autonome prendendo esempio da questo tipo di astrazione può risultare in un design molto più pulito ed efficiente. Quello che vogliamo cercare di capire con questo progetto è quanto sia facile implementare un sistema dove più di queste entità autonome possano interagire con l'ambiente, e tra loro, e quanto i Game Engines moderni supportino il programmatore in questo; ricercheremo quindi le astrazioni che più si avvicinino a quelle tipiche dei Sistemi Multi Agente e cercheremo di colmare il gap per quelle mancanti.

2 Multi Agent System

Un sistema multi agente è composto da più entità proattive dette agenti, le quali incapsulano uno stato e un comportamento e agiscono in autonomia per raggiungere un certo goal: tali entità sono parte di una società, quindi vari agenti saranno in grado di scambiare informazioni tra di loro mediante un'opportuna organizzazione e interazione formando una società, inoltre sono presenti all'interno di un ambiente con il quale poter interagire. Le astrazioni proprie di un Sistema Multi Agente sono quindi tre e sono descritte in seguito.

2.1 Astrazioni

- `Agenti`: Le entità individuali che compongono il sistema, sono computazionalmente autonomi ed incapsulano il loro comportamento e il loro stato, autogovernando la modalità con cui cercano di raggiungere un certo obiettivo: sono entità dinamiche, possono esercitare azioni in maniera attiva e proattiva, si basano quindi su ciò che la società e l'ambiente sono in grado di comunicargli in un certo istante. Per poter interagire in maniera efficace con

l'ambiente circostante gli agenti devono avere una propria rappresentazione di esso, quindi avranno sia lo stato corrente dell'ambiente sia le regole da seguire per poter efficacemente muoversi all'interno della società e dell'ambiente stesso. Quindi reattività, cioè agire a seconda dei cambiamenti derivanti dall'ambiente circostante, proattività nella loro esecuzione (quindi niente passività nella risposta a stimoli) e approccio situato, cioè consapevole di ciò che lo circonda.

- `Società`: Enfatizza la molteplicità di agenti che fanno parte di un *MAS*, nel quale è presente organizzazione, interazione tra le entità e scambio di informazioni: la società è quindi la composizione di varie entità autonome nella quale il concetto stesso di autonomia trova un senso, in quanto a livello individuale non sarebbe possibile definirlo; quindi, l'interazione tra i vari agenti all'interno dell'ambiente globale è un'astrazione fondamentale per i *MAS*.

- `Ambiente`: In quanto componenti di un *MAS* gli agenti sono in grado anche di interagire con l'ambiente che circonda la società stessa in cui sono presenti; l'interazione con l'ambiente porta gli agenti a scambiare informazioni e conoscenza in maniera pragmatica, con la possibilità di cambiare il proprio comportamento a seconda delle informazioni stesse; inoltre, la caratteristica di essere situati, cioè immersi, nell'ambiente in cui si trovano permette agli agenti di percepire e produrre cambiamenti sulle proprie azioni.

2.2 Modelli di Coordinazione

Importanza determinante in un *MAS* è l'approccio alla coordinazione delle entità che ne fanno parte, utilizzando un modello di coordinazione che permetta di esprimere le interazioni tra gli agenti, la comunicazione e la sincronizzazione. Il paradigma utilizzato per esprimere la coordinazione tra agenti in un *MAS* è il message passing, come modalità di scambio di informazioni : vengono disaccoppiate le entità comunicanti e le strutture di controllo facenti parte del model stesso. Lo stesso modello di coordinazione deve poi esse-

re in grado di elencare in maniera rigorosa tutto l'insieme di regole che esprimano come i vari agenti saranno in grado di coordinarsi, sia attraverso il medium di comunicazione sia attraverso l'utilizzo di primitive di coordinazione: il nostro lavoro si è concentrato sull'utilizzo di message passing come modello data-oriented di comunicazione, in particolare utilizzando come astrazioni lo spazio di tuple (media di coordinazione) e, come primitive, le principali di Linda (linguaggio di coordinazione).

2.2.1 Spazio di Tuple Meta-modello di coordinazione, sintassi utilizzata per esprimere e scambiare strutture dati, è un modello di coordinazione basato sulla rappresentazione dello spazio di comunicazione (communication media) come spazio di tuple, nel quale i coordinables (cioè gli agenti) si sincronizzano, cooperano e competono utilizzando un linguaggio di comunicazione basato su strutture dati dette tuple; nello spazio di tuple risiederanno quindi i messaggi mandati dai vari agenti presenti nel *MAS* e verranno perciò resi disponibili a qualunque entità decida di interagire con lo spazio di tuple stesso (accedendo, consumando e producendo messaggi); i vari messaggi avranno una certa struttura, descritta successivamente.

2.2.2 Primitive Linda Linguaggio di coordinazione su cui ci basiamo per creare/consumare/ottenere le tuple presenti nello spazio di tuple, in particolare, nel nostro progetto, sono state implementate le 3 primitive principali di Linda, ovvero:

- `out(TT)`: inserisce il messaggio all'interno dello spazio di tuple
- `in(TT)`: acquisice un messaggio dallo spazio di tuple che faccia match con il pattern TT
  Ha varie proprietà:
  1. lettura distruttiva: una in(TT) comporta la rimozione del messaggio appena letto

  2. non-determinismo: la scelta del messaggio da leggere sarà non deterministica quando si ha più di un match (nel nostro caso viene rimosso il primo messaggio trovato)

– `rd(TT)`: acquisisce un messaggio dallo spazio di tuple che faccia match con il pattern TT
  Proprietà:
  1. lettura non distruttiva: la rd(TT) legge il messaggio dallo spazio di tuple se ne è presente almeno 1 con il pattern definito senza rimuoverlo dallo spazio di tuple stesso

  2. non-determinismo: la scelta del messaggio da leggere sarà non deterministica quando si ha più di un match (nel nostro caso viene rimosso il primo messaggio trovato)

Per quanto riguarda le semantiche con cui le primitive agiscono sono state implementate sia la sospensiva sia la non sospensiva: la descrizione è presente nel capitolo 4.

3 Game Engines

Un game engine è un framework di lavoro creato appositamente per facilitare la progettazione e l'implementazione di videogiochi. In genere un engine fornisce supporto per rendering grafico (2D o 3D), la fisica (movimenti, collision detection, ecc...), suoni, scripting e molto altro.
Qui analizzeremo due tra gli engine più utilizzati, Unity e Unreal, cercando quelle features che possano rappresentare meglio le astrazioni tipiche dei *MAS*.

3.1 Unity3D

Unity, sviluppato nel 2005 dalla Unity Technologies, alla versione attuale (5.3.4), oltre alle tipiche features dei Game Engines (rendering, fisica, scripting, luci, suoni, ecc...) mette anche a disposizione un cross compiler che lo render appetibile per chi vuole rilasciare il proprio gioco su piattaforme differenti, con il minimo sforzo.
Di seguito analizzeremo il funzionamento del game loop e altre funzionalità interessanti, con focus sulle possibili astrazioni che ci permetteranno di implementare un *MAS*.

Di seguito verrà descritta una panoramica di funzionamento di questo engine grafico.

3.1.1 GameObject All'interno di Unity esiste una sola astrazione base per tutto: il GameObject.
Qualunque cosa verrà aggiunta alla scena, che sia una Camera, una luce, o un giocatore, sarà un GameObject al quale sarà poi possibile aggiungere uno o più componenti che gli permettano di assumere funzionalità aggiuntive.

3.1.2 Scheduling Unity, in maniera superficiale, esegue tutto sequenzialmente senza mettere in campo nessuna politica di parallelizzazione (se sotto, alcune cose siano ottimizzate utilizzando approcci multi-threading, questo non possiamo saperlo); inoltre, non è thread-safe.
Da una parte avere tutto sequenziale abbassa l'entry point per chiunque voglia progettare un videogioco, senza il bisogno di particolari conoscenze teoriche di certi aspetti, dall'altra però limita terribilmente la possibilità di progettare qualcosa di estremamente più efficiente.
Avere un game loop sequenziale significa che ogni entità presente nel gioco non avrà un proprio flusso di controllo, quindi non solo la progettazione diventa terribilmente più complessa ma questo limita anche la possibilità di avere entità completamente autonome e indipendenti come gli Agenti.

Gli script in Unity sono dei file sorgente che estendono MonoBehaviour e che al suo interno avranno i metodi tipici di quella classe (Awake, Start, Update, ecc...). Ogni script strutturato in questo modo è possibile aggiungerlo a un GameObject per potergli dare un comportamento che viene eseguito ad ogni frame del gioco.
Quello che Unity fa in un dato istante, è eseguire in maniera sequenziale tutti i metodi, e.g. tutti gli Update, presenti nei vari scripts. Il problema della sequenzialità è sentito particolarmente quando è necessario progettare un comportamento per una certa entità, questo perché ad ogni frame del gioco devono essere chiamati tutti i metodi Update di tutti gli script associati ai GameObject e se uno di que-

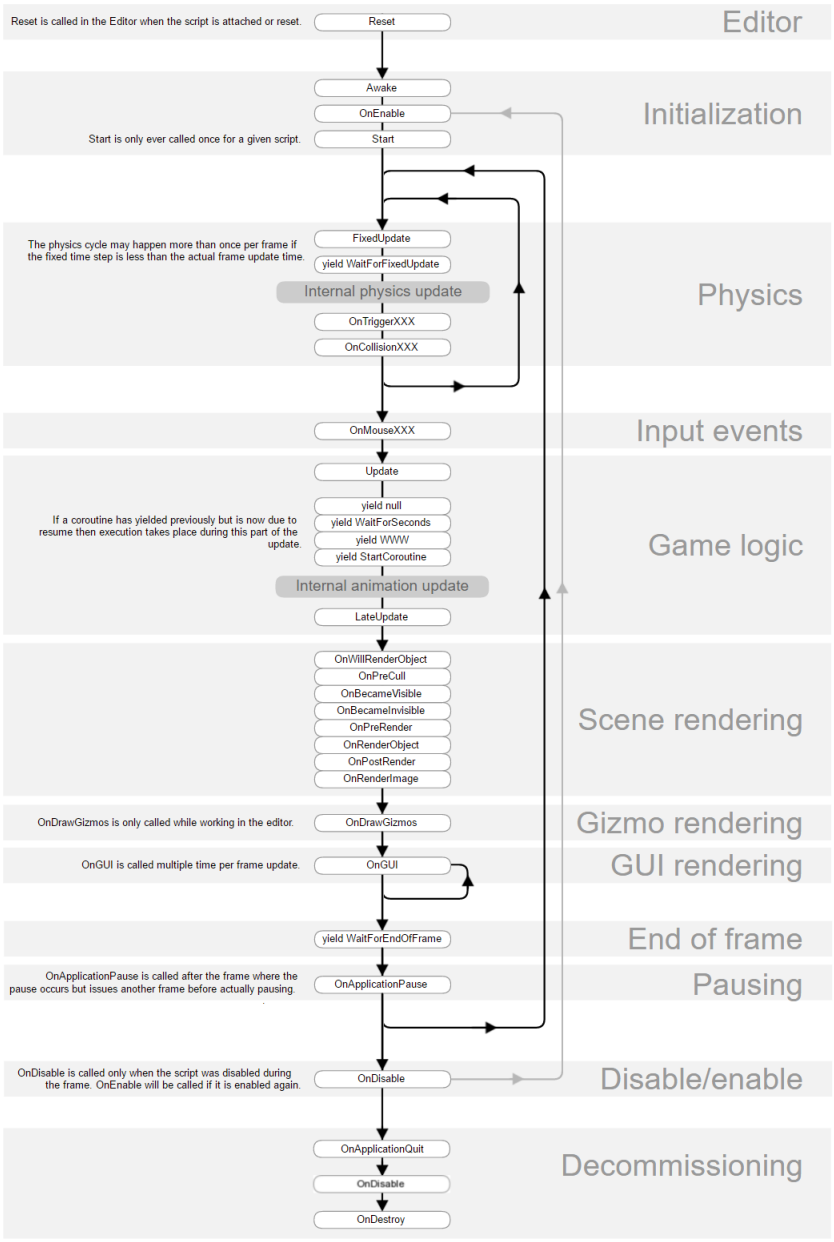

Figura 1. MonoBehavior flowchart

sti Update risulta particolarmente oneroso computazionalmente, le prestazioni del gioco caleranno drasticamente.

3.1.3 Coroutine Visto il problema della sequenzializzazione, spesso il programmatore deve pensare a dei workaround che gli permettano di progettare un comportamento complesso senza per forza ammazzare le prestazioni del gioco.
Le coroutine vengono utilizzate per distribuire la computazione di un certo algoritmo su più frame; permettono di sospendere l'esecuzione, di metterla in pausa, e ripristinarla al frame successivo, riprendendo dal punto in cui viene sospesa (attraverso uno yield).

Utilizzando il metodo StartCoroutine("<nomecoroutine>") viene istanziata una coroutine e Unity si ricorderà al frame successivo che dovrà eseguirne una parte, senza bisogno che venga espressamente esplicitato.

3.2 Astrazioni *MAS* in Unity

Vedremo ora quelli che sono i supporti nativi di Unity che ci permetteranno di iniziare a implementare una prima versione di un *MAS*.

3.2.1 Ambiente L'ambiente è l'unica vera astrazione che Unity ci mette a disposizione per cercare di implementare un *MAS*: è tutto ciò che è presente in una scena, è possibile interagirci, cercare oggetti attraverso tecniche come il ray-casting o semplicemente conoscendone il nome o il tag associato, inviare messaggi e molto altro.

3.2.2 Società Una vera e propria astrazione dedicata non esiste. Esiste però la possiblità di scambiare messaggi tra oggetti in maniera più o meno selettiva: si possono scambiare messaggi per tag (Unity da' la possiblità di taggare gli oggetti presenti nella scena); si possono scambiare messaggi per nome (attraverso i nomi che diamo agli oggetti nella scena).
Lo scambio di messaggi avviene specificando il metodo che il ricevente dovrà eseguire e eventualmente un messaggio vero e proprio.

È quindi possibile implementare coordinazione tra agenti attraverso il message passing.
In maniera più creativa, si potrebbe anche pensare di inviare i messaggi a un singolo oggetto che funge da "storage" e inviare poi un messaggio in broadcast agli altri agenti dicendogli che c'è un nuovo messaggio da leggere, implementando così una sorta di spazio di tuple, similmente a quello che vedremo nel caso di studio più avanti.

3.3 Agenti

Considerando il fatto che in Unity l'unica vera astrazione è il GameObject, non esiste niente che permetta di implementare un Agente. L'unico modo è attaccare uno script che implementi un comportamento che possa essere simile a quello che dovrebbe avere un Agente nei *MAS*. Esistono alcuni componenti e funzionalità che possono aiutare nell'implementazione di un Agente ma la maggior parte del comportamento andrà sviluppato ex-novo.
Una cosa che tornerà utile sarà il componente chiamato NavMesh Agent che permette di calcolare automaticamente un percorso, una volta settata una destinazione, facendo obstacle avoidance in maniera trasparente.

3.4 Unreal Engine

Unreal Engine (la cui ultima versione del motore grafico è la 4) è una suite per il game development sviluppata da Epic Games, introdotta per la prima volta nel 1998 per la realizzazione del famoso FPS Unreal: gli sviluppi del motore grafico hanno seguito di pari passo le generazioni dei vari hw presenti sul mercato, adattando il software alle potenzialità che essi presentavano.
Unreal utilizza il C++ e il Blueprint Visual Scripting come linguaggi con cui creare degli script.
Il lavoro svolto su questo engine è stato di analisi e ricerca delle astrazioni caratterizzanti un *MAS*, le quali non sono direttamente presenti ed utilizzabili (come in Unity, non esistono dirette rappresentazioni di tali astrazioni); tuttavia sono presenti dei meccanismi che, se opportunamente sfruttati, possono risultare utili, descritti di seguito come possibile astrazione di un *MAS*; il lavoro di implemen-

tazione è stato poi proseguito esclusivamente utilizzando Unity.

3.4.1 Ambiente Come per Unity, l'ambiente è l'unica vera astrazione che il game engine ci fornisce direttamente: tutto ciò che è possibile inserire in una scena è parte integrante dell'ambiente, con cui i vari bot possono interagire.

3.4.2 Società Simile a Unity, non esiste una propria astrazione e modellazione di società, ma esistono dei meccanismi per effettuare comunicazioni tra Blueprints.
Come forme di comunicazione abbiamo gli Events, i quali consistono in chiamate dovute a determinate situazioni di gameplay (come inizio livello, fine livello, danno preso, ecc...), tali chiamate permettono a dei Blueprints di eseguire del codice in risposta a tali eventi.
Inoltre, esistono i Direct Blueprint Communication, i quali permettono a due Blueprints di comunicare tra di loro (comunicazione sempre one-to-one); gli EventDispatchers comunicano un messaggio in broadcast a tutti i Blueprints che sono ad esso registrati; inoltre, esistono le Blueprint Interface le quali permettono a oggetti diversi di reagire in maniera diversa ad una stessa chiamata (ad esempio un oggetto Car e un oggetto Tree implementano una stessa Blueprint Interface con un metodo weaponFireDamage, quindi tale metodo può essere chiamato e ogni oggetto risponderà diversamente).

3.4.3 Agenti Gli Actors sono oggetti che possono essere posizionabili nei livelli e consistono in una classe generica con supporto alla modellazione 3D: possono essere controllati da un player oppure possedere comportamenti automatici (e agire quindi come bot, NPC).
Pawn: classe base di tutti gli Actors che possono essere controllati da un opportuno controller, sia del player sia con IA, quindi il Pawn determina sia come il player viene interpretato in maniera grafica sia come interagisce con il mondo (collisioni, ecc...). Esiste una variante del Pawn, chiamata Character, il quale è inteso come Pawn ma con aspetti più umanoidi (ad esempio camminare). Ogni Pawn ha un suo controller: per il controller artificiale si utilizza la classe Behavior Tree (si creano i bot), con la quale è possibile creare NavMesh,

AI Controllers per i Characters, ecc... Attraverso le NavMesh si è in grado di dotare il Pawn di eseguire pathfind, obstacole avoidance, saltare, ecc...

4 Spazio di Tuple: una prima implementazione

4.1 Proprietà

Questa prima implementazione ha tre campi:

- `tupleSpace`: dove vengono salvate le tuple (implementate come semplici stringhe)
- `waitQueue`: per memorizzare le richieste ancora in attesa (tutte le richieste che appena arrivate non potevano essere servite per mancanza della tupla richiesta)
- `inQueue`: per memorizzare le richieste appena arrivate

4.2 Comportamento

Il comportamento viene specificato nel metodo Update dello script: ad ogni frame si cercherà prima di gestire tutte le richieste in attesa sulla waitQueue e succesivamente quelle nella inQueue.
Gestione richieste:

1. viene controllato il tipo della richiesta (`RD`, `IN`, `OUT`)
2. in caso sia di tipo RD e la tupla presente in tupleSpace, allora viene recuperato il riferimento del mittente e gli viene inviato un messaggio con la tupla richiesta
3. in caso sia di tipo IN e la tupla presente in tupleSpace, uguale al caso RD ma la tupla viene anche rimossa da tupleSpace
4. in caso sia di tipo OUT, la richiesta viene servita subito inserendo la tupla nel tupleSpace
5. le richieste in ingresso, dentro inQueue, che non possono essere servite vengono spostate in waitQueue

4.3 Agenti Sincroni

L'agente sincrono, ogni volta che effettuerà una richiesta (invierà un messaggio allo spazio di tuple), sia questa di IN, RD o OUT, si "sospenderà" fino a quando lo spazio di tuple non invierà a sua volta un messaggio all'agente (chiamando il metodo ReceiveMessage) con la risposta alla richiesta.

Una nota sulla "sospensione": essendoci un solo flusso di controllo, l'unico modo per implementare chiamate sincrone è quello di uscire subito dal metodo Update quando si sta aspettando una risposta, implementando quindi una sorta di busy waiting.

4.4 Agenti Asincroni

L'agente asincrono, ogni volta che effettuerà una richiesta, a differenza della controparte sincrona, non si sospenderà ma inserirà il messaggio inviato in una waitQueue locale, quindi alla ricezione di un qualche messaggio l'agente andrà a controllare se effettivamente è il messaggio che stava aspettando.

5 Caso di Studio - Dining Philosophers in Unity

Ora vediamo una possibile implementazione del problema dei Dining Philosophers su Unity utilizzando i concetti precedentemente discussi: l'obiettivo del nostro lavoro consiste nel verificare quanto tali concetti siano effettivamente compatibili con le astrazioni tipiche di un *MAS*, quindi riuscire a implementare comunicazione e coordinazione tra i vari filosofi/agenti presenti nella scena.

5.1 La Scena

La scena del gioco è composta dei seguenti elementi:

- 5 Robot Umanoidi che rappresentano i Filosofi
- 5 Sedie
- 1 Tavolo che rappresenta il medium di coordinazione (Spazio di Tuple)

– 5 Chopsticks/Forchette che rappresentano le risorse condivise dai Filosofi, necessarie per poter mangiare
– Alcuni elementi decorativi quali: 5 piatti per i Filosofi e 1 grande da portata al centro del tavolo.

Visivamente si può capire quale azione sta compiendo ogni Filosofo in base al colore che l'agente assume: quando il filosofo si muove per la scena ricercando una sedia e sta pensando assume il colore grigio (Figura 2), quando si siede e aspetta di mangiare assume il colore rosso (Figura 3 e 4), quando mangia assume il colore verde e le "chopsticks" a lui assegnate gli si avvicinano (più precisamente, si incrociano sopra alla sua ciotola) (Figura 3 e 4), per poi ritornare al loro punto iniziale quando il Filosofo ha terminato la fase di Eating; inoltre, ogni Filosofo presente nella scena ha animazioni specifiche per camminare, sedersi e alzarsi dalla sedia. In maniera del tutto analoga anche le sedie cambiano colore in base alle azioni dei filosofi: diventano verdi quando risultano libere e quindi disponibili ad un Filosofo che intercettandola decide di sedersi, diventano rosse quando sono occupate da un Filosofo.

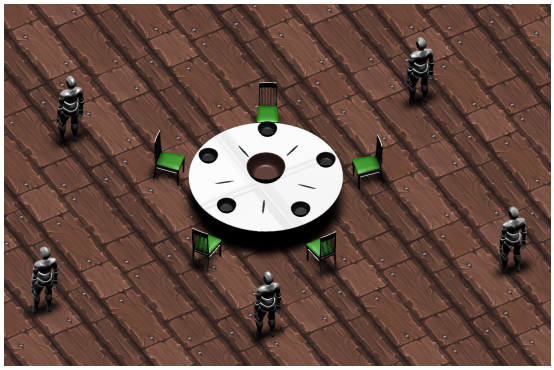

Figura 2. Situazione al GameStart

Figura 3. Situazione all'inizio del turno 1

Figura 4. Situazione alla fine del turno 1

5.2 Implementazione Filosofi

Ogni Filosofo rappresenta un Agente in grado di muoversi nell'ambiente, pensare, cercare una sedia per sedersi e mangiare. Abbiamo implementato il Filosofo utilizzando principalmente Coroutine perché era necessario un meccanismo per dare tempo al Filosofo di pensare e mangiare. Ogni Filosofo, all'inizio del gioco, inizierà una Coroutine per pensare che impiegherà un tempo casuale, dai 5 ai 15 secondi, per essere eseguita completamente. Quando un Filosofo terminerà di pensare, inizierà un'altra Coroutine per cercare una sedia disponibile (cioè che non sia già stata presa da un altro Filosofo), attraverso il casting di un raggio e cercando oggetti taggati "Chair", dove sedersi e mangiare. Trovata la sedia si metterà in movimento verso di essa e si sederà. A questo punto abbiamo implementato due versioni differenti del Filosofo: una versione Situata e una Non Situata. Nella versione Non Situata, il Filosofo catturerà l'id della sedia e sarà lui stesso a fare una richiesta al Tavolo (lo spazio di tuple) esplicitando il posto dove è seduto e la volontà di mangiare, quindi lo spazio di tuple capisce quali "chopsticks" dare ad ogni filosofo in base al suo identificativo. Nella versione Situata, invece, abbiamo fatto in modo di disaccoppiare anche questo aspetto facendo la richiesta di mangiare alla Sedia (rendendola quindi attiva) la quale cercherà poi di contattare il Tavolo al posto del Filosofo: l'approccio situato permette all'astrazione di coordinazione di essere perfettamente conscia dell'ambiente che la circonda, avvicinando tale soluzione al mondo reale (nel quale le chopsticks vengono assegnate in base al posto a tavola). Il Filosofo a questo punto aspetterà di essere informato dal Tavolo (o dalla Sedia) per poter iniziare a mangiare. Infine, quando avrà finito di mangiare, rilascerà la Sedia, si alzerà e si muoverà verso una posizione dove poter iniziare a pensare.

5.3 Coordinazione

Vediamo adesso i meccanismi che abbiamo utilizzato per realizzare la coordinazione tra i Filosofi.

5.3.1 Spazio di Tuple La struttura base dello Spazio di Tuple è la stessa spiegata nel capitolo precedente (si utilizzano perciò

waitQueue e inQueue), quindi, ad ogni frame, ogni volta che verrà chiamato il metodo Update dal Game Loop, cercherà di servire prima le richieste nella lista di attesa e poi quelle appena arrivate.
Cercando di servire le richieste, lo Spazio di Tuple verificherà che siano disponibili le "chopsticks" corrispondenti all'ID della richiesta (l'ID della Sedia in questo caso): se le "chopsticks" sono disponibili allora manderà un messaggio a chi ha fatto la richiesta (Sedia o Filosofo) consegnando le risorse; se non sono presenti metterà la richiesta in lista d'attesa.

5.3.2 Message Passing Tutte le richieste vengono fatte attraverso lo scambio di messaggi. Il Filosofo invierà un messaggio alla Sedia, o al Tavolo nel caso della versione non situata, comunicando la sua volontà di mangiare. Nella versione situata, la Sedia, dopo aver ricevuto la comunicazione dal Filosofo, si appresterà a inviare un messaggio al Tavolo passando a lui la stessa richiesta.
Nella versione non situata il Filosofo farà la richiesta direttamente al Tavolo dopo aver acquisito l'ID della Sedia su cui è seduto. Quando il Tavolo riuscirà a servire una delle richiesta, invierà un messaggio al richiedente (Sedia o Filosofo) consegnando le risorse.

6 Conclusioni e sviluppi futuri

Con questo progetto abbiamo mostrato una possibile integrazione tra Game Engine moderni, in particolare Unity, e i Sistemi Multi-Agente.
Il supporto che i Game Engine attualmente forniscono per questo tipo di simulazioni è piuttosto basso ma abbiamo dimostrato che non è comunque impossibile realizzarle lo stesso. Avere un unico flusso di controllo va in opposizione all'idea di base degli Agenti nei *MAS* ma è inevitabile avendo a che fare con questo tipo di framework; quindi, sebbene il gap tra astrazioni tipiche di un *MAS* e Game Engines non sia stato del tutto colmato, è stato comunque possibile sviluppare un'implementazione interessante che sicuramente, in futuro, non potrà fare altro che essere ulteriormente migliorata.

Riferimenti bibliografici

1. Distributed Systems Course, http://apice.unibo.it/xwiki/bin/view/Courses/Sd1516

   Unreal Engine Documentation, https://www.unrealengine.com/blog

   Unity3D Documentation, http://docs.unity3d.com

   APICe Project page, http://apice.unibo.it/xwiki/bin/view/Courses/Sd1516Projects-GamingMasPoliCerbara