

ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

GAME ENGINES E MAS: SPATIAL TUPLES IN
UNITY3D

Tesi in:
Sistemi Autonomi

Relatore:
Chiar.mo Prof.
ANDREA OMICINI

Co-relatore:
Dott. STEFANO MARIANI

Presentata da:
ALESSANDRO BAGNOLI

I SESSIONE
ANNO ACCADEMICO 2017–2018

PAROLE CHIAVE

Unity3D

Spatial Tuples

BDI

Linda

MAS

*“Any sufficiently advanced technology is
indistinguishable from magic.”*

— Arthur C. Clarke

Indice

Sommario	ix
1 Introduzione	1
2 Stato dell'Arte	5
2.1 Sistemi Multi-Agente	5
2.1.1 Agenti BDI	6
2.1.2 Società e Coordinazione: Linda	7
2.1.3 Ambiente	9
2.2 Unity3D	9
2.3 MAS in Unity3D	10
2.3.1 Agenti BDI in Unity3D	11
2.3.2 Linda in Unity3D	12
2.4 Il Modello Spatial Tuples	13
2.4.1 Primitive	15
2.4.2 Collocazione e Mobilità	16
2.4.3 Formalizzazione	17
3 Spatial Tuples in Unity3D	19
3.1 Il Mapping dello Spazio Fisico Reale su Unity3D	20
3.1.1 WRLD SDK	20
3.1.2 Geocoding	23
3.2 Primitive	27
3.2.1 Primitiva out	28
3.2.2 Primitiva in	38
3.2.3 Primitiva rd	46
3.3 Pattern di Coordinazione	46

3.3.1	Situated Knowledge Sharing	46
3.3.2	Sincronizzazione Spaziale e Mutua Esclusione	52
3.4	Caso di Studio: Rescue	57
3.4.1	Setup su Unity3D	59
4	Analisi delle performance	67
4.1	Il Profiler	67
4.1.1	Il Profiler nel Caso di Studio Rescue	69
5	Conclusioni e Sviluppi Futuri	73
Ringraziamenti		75
Bibliografia		77
Elenco delle figure		81

Sommario

Questa tesi ambisce ad integrare due diversi modelli computazionali: quello proprio del game engine Unity3D – che, come sarà illustrato nel corso della digressione, si basa su esecuzione sequenziale e coroutine – con quello proprio degli agenti BDI. Il motivo che spinge a fare ciò è poter sfruttare le potenzialità dei game engine per ingegnerizzare l’astrazione di ambiente nei MAS. Dunque, considerando che spesso l’ambiente rappresenta un mezzo di coordinazione, nel lavoro di integrazione vengono considerati anche i modelli di coordinazione tuple-based. Come riferimento per concretizzare il tutto viene usato il modello Spatial Tuples.

I game engine si sono già dimostrati adatti a supportare molti dei requisiti richiesti dalla coordinazione di agenti situati, ossia supportano in maniera soddisfacente la coordinazione mediata dall’ambiente e collocata nello spazio. L’obiettivo finale è quindi quello di migliorare lo stato dell’arte rappresentato da due librerie rese disponibili ai programmatori Unity3D, frutto del lavoro di due tesi di laurea magistrale – nello specifico un motore BDI-like e un servizio Linda-like di coordinazione e comunicazione. Il miglioramento è mirato a fornire il miglior supporto possibile al modello di coordinazione Spatial Tuples recentemente proposto in letteratura, fornendo al programmatore finale delle API che permettano di utilizzare il suddetto modello all’interno di Unity3D con il minimo sforzo.

Capitolo 1

Introduzione

I modelli e le tecnologie agent-oriented si sono dimostrati essere utili in tutte quelle situazioni in cui è richiesta la presenza di entità autonome le cui azioni e interazioni determinano l'evoluzione del sistema. Un sistema multi-agente (**MAS**) 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, modificandolo. La caratteristica degli agenti di essere situati nell’ambiente in cui si trovano permette loro di percepire e produrre cambiamenti su di esso.

Studi successivi hanno portato ad ideare un nuovo meta-modello per i MAS, in cui le entità immerse nell’ambiente non sono solo gli agenti, ma anche gli artefatti, strumenti che possono influenzare i processi cognitivi degli agenti e che possono essere utilizzati per i loro obiettivi. Questo meta-modello viene chiamato Agents and Artifacts (A&A).

Come il nome lascia intuire, secondo questo meta-modello i MAS sono modellati sulla base di due fondamentali astrazioni computazionali, gli agenti e gli artefatti. Gli agenti rimangono le entità autonome, pro-attive che encapsulano il controllo e determinano il comportamento dell’intero MAS, come descritto precedentemente. Gli artefatti invece sono le entità passive e reattive che

forniscono i servizi e le funzioni che permettono ai singoli agenti di lavorare insieme in un MAS, e danno forma all’ambiente in accordo con le esigenze del MAS. Viene inoltre introdotta una nuova astrazione, chiamata *workspace*, definita come container concettuale degli agenti e degli artefatti, utile per definire la topologia dell’ambiente e che fornisce un modo per specificare una nozione di località [1]. Si può quindi pensare al workspace come ad una porzione dell’ambiente.

In un contesto agent-based, la coordinazione rappresenta la chiave di volta in grado di assicurare la corretta esecuzione del sistema e grazie alla quale le attività e le interazioni degli agenti vengono gestite correttamente, ovvero facendo lavorare insieme gli agenti per raggiungere i goal.

I Game Engines (**GE**) sono framework resi disponibili per i designer di videogiochi che permettono di programmare e pianificare un gioco senza costruirne uno completamente da zero, ed offrono strumenti per coadiuvare la creazione e la disposizione delle risorse di gioco. Forniscono aspetti avanzati quali rendering di modelli 2D e 3D, motori fisici, rilevamento delle collisioni, etc. La creazione di videogiochi tuttavia non rappresenta il solo campo applicativo messo a disposizione dai GE: essi possono essere sfruttati per gli scopi più svariati, permettendo il loro utilizzo anche in varie aree di ricerca scientifica, compresa quella dei MAS [2].

Il lavoro svolto in questa tesi ha lo scopo di rafforzare ulteriormente un tipo di legame che ha già dimostrato di poter emergere ed essere sfruttabile: l’utilizzo dei GE come supporto alla progettazione dell’ambiente nei MAS, e dunque come mezzo abilitante per la coordinazione mediata dall’ambiente. Nello specifico si cercherà di dare concretezza ad un recente modello di coordinazione per agenti proposto solo in letteratura: *Spatial Tuples* [3]. Il goal sarà raggiunto sfruttando le potenzialità del GE Unity3D [4] e avvalendosi del lavoro svolto precedentemente da altri due tesisti [5] [6].

La tesi sarà quindi organizzata come segue: il Capitolo 2 si aprirà con una digressione sui sistemi multi-agente, con particolare attenzione riguardo all’architettura BDI degli agenti e alla coordinazione tramite Linda. Analizzerà poi lo stato dell’arte circa l’integrazione tra MAS e GE, fornendo una descrizione dei lavori già svolti citati poco fa ed introducirà nel dettaglio i formalismi, la semantica e la sintassi previsti da Spatial Tuples. Nel Capitolo 3 verrà illustrata la soluzione proposta dal punto di vista del design e dell’implementazione ottenuta durante il lavoro progettuale svolto, mostrando anche come è stato possibile ottenere un mapping dello spazio fisico reale all’interno di Unity3D,

passo fondamentale per poter arrivare ad una rappresentazione di un modello di coordinazione basato sullo spazio come Spatial Tuples. Si proseguirà mostrando tutti i pattern previsti da Spatial Tuples implementati attraverso la soluzione progettata e verrà analizzato un caso di studio previsto dalla letteratura in cui questi pattern saranno sfruttati per la sua implementazione. Il Capitolo 4 tenterà di fornire un'analisi delle performance del MAS implementato per il caso di studio descritto nel precedente capitolo, fornendosi degli strumenti messi a disposizione da Unity3D. Infine, nel Capitolo 5 si discuterà circa le conclusioni e verranno forniti spunti di riflessione per eventuali lavori futuri.

Capitolo 2

Stato dell'Arte

Questo capitolo introduce e riesamina le basi del lavoro principale, riassumendo alcuni dei concetti teorici fondamentali dei sistemi multi-agente (MAS) e di Unity3D, il Game Engine (GE) preso in esame. L'obiettivo è discutere ed evidenziare la loro importanza in diversi campi di ricerca, per poi mostrare come alcune delle astrazioni previste dai MAS siano state mappate all'interno di Unity3D. Il capitolo si concluderà con una digressione su Spatial Tuples, il modello di coordinazione per agenti che questa tesi vuole concretizzare e del quale si vogliono mostrare le potenzialità.

2.1 Sistemi Multi-Agente

La crescente complessità nell'ingegnerizzazione dei sistemi software ha portato alla necessità di modelli e astrazioni in grado di rendere più facile la loro progettazione, lo sviluppo e il mantenimento. In questa direzione, la computazione orientata agli agenti viene in aiuto agli ingegneri ed informatici per costruire sistemi complessi, virtuali o artificiali permettendo una loro agevole e corretta gestione [7].

In particolare, la ricerca e le tecnologie per MAS hanno introdotto nuove astrazioni per affrontare la complessità durante la progettazione di sistemi o applicazioni composte da individui che non agiscono più da soli ma all'interno di una società. Le tecnologie e i modelli agent-oriented sono attualmente diventati una potente tecnica in grado di affrontare molti problemi che vengono alla luce durante la progettazione di sistemi computer-based in termini

di entità che condividono caratteristiche quali l'autonomia, l'intelligenza, la distribuzione, l'interazione, la coordinazione, etc.

L'ingegnerizzazione dei MAS si occupa infatti di costruire sistemi complessi dove più entità autonome chiamate agenti cercano di raggiungere in maniera proattiva i loro scopi sfruttando le interazioni tra di essi (come una società), e con l'ambiente circostante. Questo modello può essere visto come un paradigma general-purpose, il quale prevede l'utilizzo di tecnologie agent-oriented in diversi scenari applicativi [8].

Gli agenti basano la loro definizione pratica sul concetto di autonomia come loro caratteristica fondamentale e definiscono il MAS, concepito come un'aggregazione di agenti che interagiscono. Un agente incapsula la complessità in termini di informazioni (cosa deve sapere per fare qualcosa), azioni (il processo per raggiungere alcuni obiettivi), intelligenza, mobilità, collocazione, interattività. In questo modo, la comunicazione con altri agenti e l'interazione con l'ambiente stesso diventano un'astrazione fondamentale dell'ingegneria del sistema, oltre ad essere potenziali precursori di novità comportamentali e azioni [9].

Quindi, il ruolo chiave svolto dai modelli di coordinazione ed interazione non è visto solo come un'astrazione di supporto al comportamento autonomo dell'agente, che risolve in modo efficace problemi di coordinazione e che consente interazioni tra la società formata da agenti scambiando informazioni e conoscenze, ma anche come un fondamentale strumento che aiuta direttamente la fase di progettazione del MAS [10].

Ora segue una descrizione delle tre astrazioni fondamentali previste dai MAS, ponendo particolare attenzione su due loro modelli specifici: il modello BDI per l'astrazione agente, e il modello di coordinazione Linda per trattare con la complessità delle interazioni previste dall'astrazione società.

2.1.1 Agenti BDI

Gli agenti rappresentano l'astrazione principale dei MAS: entità proattive che incapsulano il controllo, governandolo attraverso azioni che consentono all'agente stesso di perseguire i propri obiettivi (ciò che vuole ottenere) usando e cambiando qualcosa nel mondo in cui sono immersi (percependo lo stato dell'ambiente e adattando le proprie azioni e il proprio comportamento in base ad esso). In questo senso, gli agenti sono situati, strettamente accoppiati con il contesto e l'ambiente circostante e, cosa più importante, sono sociali,

esprimendo così l'autonomia con le interazioni tra agenti come avviene in una società.

Un particolare tipo di agente è quello sviluppato secondo l'architettura BDI. È possibile, e abbastanza comune, pensare ad un agente BDI come un sistema razionale con atteggiamenti mentali [11], vale a dire credenze (Beliefs), desideri (Desires), e intenzioni (Intentions), i quali rappresentano rispettivamente ciò che l'agente conosce del mondo, cosa lo motiva e cosa sta facendo per raggiungere i propri obiettivi. Come proposto in [11], il ciclo di reasoning di un agente è composto da quattro fasi principali:

1. Generazione di opzioni: l'agente restituisce un insieme di opzioni in base a cosa conosce riguardo al mondo e quali sono i suoi desideri;
2. Deliberazione: l'agente seleziona un sottoinsieme delle opzioni precedentemente selezionate;
3. Esecuzione: l'agente esegue, se è presente la relativa intenzione, un'azione tra le opzioni precedentemente selezionate;
4. Percezione: l'agente infine aggiorna la sua conoscenza del mondo (ed eventualmente se stesso).

2.1.2 Società e Coordinazione: Linda

La capacità sociale degli agenti riguarda la possibilità di interagire tra di loro per ottenere comportamenti collettivi coordinati, dal momento che alcuni obiettivi possono essere raggiunti solo con la collaborazione e l'interazione tra agenti. In questo senso, la complessità dei MAS può essere affrontata utilizzando modelli di interazione e coordinazione, che rappresentano il problema principale quando si progettano MAS e si affronta la complessità delle interazioni.

Lo spazio delle interazioni, qui presente come un requisito fondamentale per l'astrazione società, è una delle principali cause di complessità quando si tratta di progettazione e implementazione dei MAS. Nel corso degli anni sono stati introdotti ed esaminati diversi modelli, e ora se ne analizza uno nello specifico: il modello *Linda*.

Linda è uno dei primi (e uno dei più usati) modelli di interazione e coordinazione basato su spazi di tuple. I modelli di coordinazione basati su tuple presentano proprietà importanti che li rendono adatti per il coordinamento

di sistemi eterogenei e distribuiti, affrontando la complessità con concetti e astrazioni semplici ma consolidati, facendo giocare loro un ruolo chiave nella progettazione e nella costruzione di MAS complessi.

Nei modelli basati su tuple, i coordinabili interagiscono tra di loro scambiando tuple come blocchi di informazioni, tramite i quali le entità coordinate sono in grado di sincronizzarsi utilizzando gli spazi di tuple come intermediari di coordinazione. Più in dettaglio, *Linda* formalizza la comunicazione generativa da utilizzare come linguaggio di coordinazione, fornendo concetti e primitive semplici ma espressive.

Un programma in *Linda* è una raccolta ordinata e possibilmente eterogenea di tuple, che incorporano le informazioni destinate a essere scambiate tra gli agenti, e sono disponibili negli spazi di tuple, che fungono da astrazione per il media di coordinazione e da contenitore di tuple. Inoltre, al fine di recuperare tuple specifiche, *Linda* consente l'uso di un'astrazione di accesso associativo, rendendo possibile la manipolazione dello spazio di tuple condiviso tramite meccanismi di abbinamento di tuple, come il pattern matching, l'unificazione, e così via.

Il linguaggio di coordinazione introdotto con *Linda* fornisce 3 primitive di base [12], capaci di gestire la manipolazione delle tuple e dello spazio di tuple stesso in maniera semplice ed espressiva:

- **out(t)**: inserisce la tupla t nello spazio di tuple.
- **in(tt)**: recupera dallo spazio di tuple una tupla che corrisponde al template tt. Ha diverse proprietà:
 1. Lettura distruttiva: la tupla viene recuperata con la rimozione dal centro di tuple.
 2. Semantica sospensiva: se non viene trovata nessuna tupla con il pattern fornito, l'esecuzione dell'agente è sospesa e viene ripresa quando viene trovata una tupla valida.
 3. Non determinismo: se più di una tupla corrisponde al pattern fornito, se ne sceglie una in modo non deterministic.
- **rd(tt)**: si comporta come in(tt), con la sola differenza che la lettura è non distruttiva.

2.1.3 Ambiente

Un'altra astrazione chiave con cui si ha a che fare durante lo sviluppo di un MAS è l'ambiente, che può essere modificato dalle azioni dell'agente [13], fornendo la mediazione alle interazioni di componenti e consentendo la comunicazione / coordinazione indiretta tra risorse esterne ed agenti. L'ambiente possiede proprietà importanti come la possibilità di collocare i componenti del MAS al suo interno, e la presenza di dinamiche autonome che influenzano le interazioni e i coordinamenti tra i componenti.

La modellazione MAS e la complessità ingegneristica risiedono anche nel concetto ambientale [9] che, insieme ad opportune astrazioni ben stabilite fornite da un corretto modello di interazione e coordinazione, rende possibile governare le azioni che sono strettamente accoppiate con proprietà ambientali, sotto forma di azioni situate e sensibilità a cambiamenti ambientali.

2.2 Unity3D

Unity3D è un GE cross-platform sviluppato da Unity Technologies, utilizzato 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 programmatore, rendendolo uno dei GE più utilizzati per produrre in maniera veloce ed efficace applicazioni e giochi.

Inoltre, questo 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 così via.

Le astrazioni 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. Tutto ciò che è presente sulla scena è un GameObject;
- *Script*: Codice sorgente applicato a un GameObject, grazie al quale è possibile assegnare a quest'ultimo un comportamento. 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.

- *Couroutine*: 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.

2.3 MAS in Unity3D

Lo stato dell'arte dell'integrazione tra GE e MAS comprende l'implementazione in Unity dei due modelli tipici dei MAS introdotti poco fa:

- Il modello Beliefs, Desires, Intentions (BDI) per la programmazione degli agenti [5].
- Un modello di coordinazione degli agenti tramite spazio di tuple e primitive Linda [6];

Il cuore pulsante di entrambi i lavori risiede nell'uso intensivo di un interprete Prolog fatto ad hoc per Unity, *UnityProlog* [14]. 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 Unity3D, 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 [5], 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 più 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.

Nelle prossime due sottosezioni segue una breve descrizione dei due lavori, e di come è stato sfruttato *UnityProlog* in ognuno di essi.

2.3.1 Agenti BDI in Unity3D

Le analogie tra il ciclo di reasoning di un agente BDI – sottosezione 2.1.1 – e il reasoning logico sono evidenti, e per questo motivo l'autore in [5] ha deciso di usare il paradigma logico come cardine del suo sistema BDI.

L'obiettivo è quello di ottenere un nuovo layer di astrazione che permette al programmatore finale di dichiarare comportamenti ad alto livello in maniera semplice e senza il bisogno di sincronizzazione esplicita con l'event loop di Unity 3D. Il comportamento di un agente potrà essere così definito interamente in un file Prolog, che avrà questa struttura:

```

1  /* Belief iniziali */
2  belief yourBelief.
3  ...
4
5  /* Desire iniziali */
6  desire yourDesire.
7  ...
8
9  /* Piani definiti */
10 add yourDesire && (preConditions) => [
11     action,
12     ...
13 ].
```

Listato 2.1: Esempio di file Prolog contenente la struttura comportamentale di un agente.

Sfruttando questo design molto simile a quello usato in *Jason* [15], l'autore si è inoltre occupato di creare un'implementazione per gli artefatti, ottenendo un primo prototipo funzionante per il meta-modello A&A [1] su Unity 3D. In maniera simile agli agenti, le caratteristiche di un artefatto sono descritte all'interno di un file Prolog, e comprendono belief e piani assimilabili da parte di un agente.

Le azioni contenute in un piano possono essere tradizionali istruzioni Prolog, ma la funzionalità chiave di questa API è la possibilità di chiamare metodi e far partire coroutine definiti all'interno dello script C# associato al GameObject rappresentante l'agente nella scena Unity.

Per richiamare un metodo sarà sufficiente scrivere all'interno di un piano

act methodname

Mentre per far partire una coroutine sarà sufficiente scrivere sempre all'interno di un piano

```
cr coroutinename
```

Il meccanismo di gestione delle coroutine ideato da [5] sarà sfruttato per l'implementazione della semantica sospensiva prevista dalle primitive **in** e **rd** di Spatial Tuples come si vedrà nel corso del Capitolo 3.

2.3.2 Linda in Unity3D

In [6], *UnityProlog* è stato utilizzato come tecnologia abilitante per l'implementazione del modello di coordinazione *Linda*. Come dichiarato nella sottosezione 2.1.2, *Linda* è basato su spazi di tuple, dove piccole porzioni di informazioni sono usate per fornire un supporto di coordinazione basato su informazioni, e sulle primitive *Linda* come le principali azioni che permettono di interagire con gli spazi di tuple aggiungendo, rimuovendo o leggendo tuple. Le primitive *Linda*, le tuple e gli spazi di tuple sono concetti già utilizzati in **TuCSoN** [16] e **LuCe** [17]. In questi progetti i concetti appena menzionati sono specificati con un modello di comunicazione logic-based e utilizzando la teoria della logica del primo ordine, quindi sfruttando le caratteristiche dell'interprete Prolog – come la definizione parziale di un template utilizzando variabili logiche, il backtracking, la sintassi dichiarativa e l'unificazione come meccanismo di matching.

Una volta ottenuto pieno accesso all'interprete Prolog, l'autore in [6] si è occupato di progettare e sviluppare una libreria *Linda* in Prolog, pienamente accessibile dal meccanismo di scripting C# di Unity 3D, e di fornire meccanismi ad alto livello per l'interazione e coordinazione sfruttabili durante la definizione comportamentale di un agente.

Ciò che è messo a disposizione del programmatore e sviluppatore Unity3D sono due librerie C# e API che supportano il modello di coordinazione ed interazione appena descritto: **LindaLibrary** e **LindaCoordinationUtilities**. La prima fornisce tutte le funzionalità a basso livello che interagiscono con *UnityProlog* e le primitive Linda scritte in Prolog. Queste funzioni C# sono sfruttabili direttamente dai programmatori Unity3D per coadiuvarli nello sviluppo di giochi o sistemi multi-agente. La seconda migliora e sfrutta il modello *Linda* di base sviluppato in **LindaLibrary** per fornire agli sviluppatori e programmatore dei servizi di interazione e coordinazione più astratti e più ad alto livello.

Uno di questi servizi che verrà usato in maniera intensiva per l'implementazione di Spatial Tuples che si mostrerà nel Capitolo 3, è tutta la parte di interazione con le `Region`. Una `Region` è una classe C# che rappresenta un'area, collocata in una posizione precisa della scena di Unity3D, composta da una forma, una posizione espressa in termini di coordinate Unity x , y e z , provvista di un Collider trasparente e personalizzabile, e di uno script C# chiamato `situatedKB` (dove KB sta per Knowledge Base) che abilita il motore Prolog e tutti i servizi Linda disponibili.

2.4 Il Modello Spatial Tuples

Spatial Tuples [3] rappresenta un'estensione del modello basato su tuple per la coordinazione degli agenti nei sistemi multi-agente (MAS), secondo il quale:

1. Le tuple sono localizzate in regioni del mondo fisico ed eventualmente si muovono insieme ad un dispositivo mobile;
2. Il comportamento delle primitive di coordinazione di Linda è esteso in modo tale da dipendere sulle proprietà spaziali degli agenti, delle tuple, e sulla topologia dello spazio;
3. Lo spazio di tuple può essere pensato come ad un layer virtuale che aumenta la realtà fisica – si veda Figura 2.1.

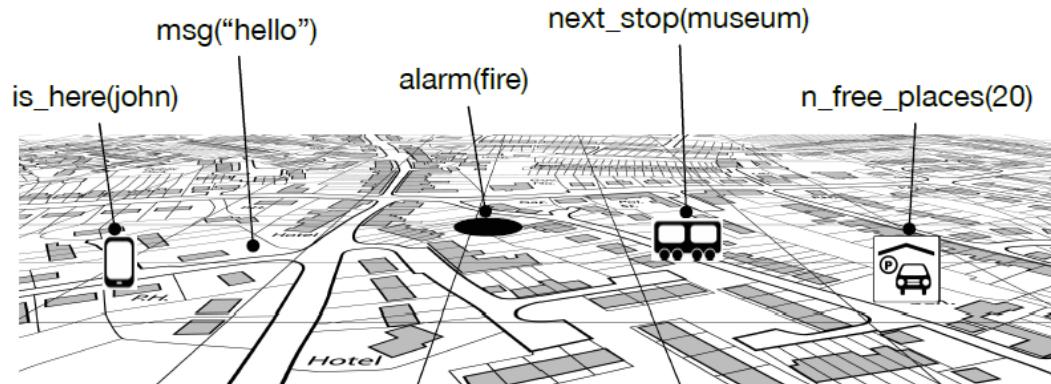


Figura 2.1: Spatial Tuples come layer virtuale che aumenta la realtà fisica [3].

Spatial Tuples punta a supportare la coordinazione space-aware e space-based in scenari di pervasive computing basati sugli agenti.

Il modello ha a che fare principalmente, come il nome suggerisce, con tuple spaziali, ovvero tuple associate ad una informazione spaziale. Le informazioni spaziali possono essere di qualunque tipo, come ad esempio coordinate GPS, posizionamento metrico (100 metri attorno alla mia posizione corrente), o amministrativo (via Zamboni 33, Bologna, Italia). In qualunque caso, il modello associa la tupla a qualche regione nello spazio fisico. Una volta che una tupla spaziale è associata ad un luogo o regione, le sue informazioni possono essere concepite come proprietà descrittive di qualunque tipologia, che vengono attribuite a quella specifica porzione di spazio fisico, aggiungendo in maniera implicita proprietà osservabili allo spazio fisico stesso. Attraverso i meccanismi di *Spatial Tuples*, le informazioni delle tuple possono essere osservate da qualunque tipo di agente che interagisce con il relativo spazio fisico, in modo da comportarsi di conseguenza.

Il punto cruciale dei modelli basati su tuple consiste nel linguaggio di comunicazione usato per definire la sintassi delle informazioni contenute nelle tuple, così come il meccanismo di matching tra le tuple e i template delle tuple. Con *Spatial Tuples*, si vuole introdurre un linguaggio di descrizione spaziale per specificare le informazioni spaziali che decorano le tuple e il relativo meccanismo di matching. Questo linguaggio spaziale è ortogonale al linguaggio di comunicazione, e il suo scopo è quello di fornire l'ontologia di base per definire concetti spaziali, come posizioni, regioni, luoghi.

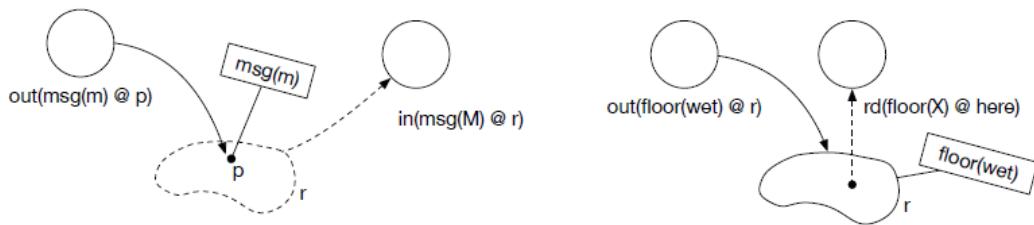


Figura 2.2: Agenti che inseriscono e recuperano tuple. A sinistra una **out** posiziona la tupla spaziale in una posizione specifica; una **in** recupera una tupla specificando una regione. A destra una **out** posiziona la tupla spaziale in una regione più grande; una **rd** osserva una tupla dalla posizione corrente [3].

2.4.1 Primitive

Come in tutti i modelli basati su tuple, le operazioni base di *Spatial Tuples* sono **out(t)**, **rd(tt)** e **in(tt)**, dove **t** è una tupla, mentre **tt** è il template di una tupla [3] – si veda Figura 2.2 per una prima intuizione sulla loro semantica¹.

Le tuple spaziali sono emesse – in altre parole, associate ad una regione **r** – per mezzo di una operazione **out**. La seguente invocazione

$$\text{out}(t @ r)$$

emette una tupla spaziale **t @ r**, ovvero una tupla **t** associata alla regione **r**.

Ad esempio, se **p1** è una regione formata da un singolo punto che rappresenta un punto preciso, e **r2** una regione, esempi di **out** sono

$$\text{out}(\text{info(message)} @ p1) \quad \text{out}(\text{floor(wet)} @ r2)$$

che associano rispettivamente la tupla **info(message)** alla posizione fisica **p1**, e decorano la regione **r2** con la tupla **floor(wet)**.

Nei modelli basati su tuple, le operazioni *getter* – **in(tt)**, **rd(tt)** – cercano tuple che fanno matching con il template **tt**. *Spatial Tuples* estende queste operazioni con la specifica del template spaziale, ovvero la regione in cui la tupla spaziale è ricercata. Le operazioni getter in *Spatial Tuples* avranno quindi questa forma:

$$\text{rd}(tt @ rt) \quad \text{in}(tt @ rt)$$

I getter cercano una tupla **t** che faccia match con il template **tt**, collocata in qualsiasi regione **r** che faccia match con il template **rt**. L’intersezione è usata per definire il meccanismo di matching tra regioni: una regione **r** fa match con un template **rt** di una regione se la loro intersezione non è vuota. Come in *Linda*, sia **in** che **rd** ritornano una tupla spaziale che fa match con il template, ma quest’ultima viene consumata solo da **in**.

Seguendo la semantica standard dei modelli basati su tuple, le primitive getter in *Spatial Tuples* sono:

- Sospensive: se non viene trovata nessuna tupla che fa match con **tt** in qualunque regione che fa match con **rt**, allora l’operazione è sospesa fino a che una tupla che fa match non viene resa disponibile in qualche modo;

¹Spatial Tuples non include la primitiva **eval** prevista dal modello tuple-based originale poiché il suo scopo è quello di valutare espressioni e processi concorrenti, e non ha quindi a che fare con la computazione distribuita (spazialmente) in nessun caso [3].

- Non deterministiche: se viene trovata più di una tupla che fa match con il template in una regione o posizione che fa match con il template spaziale, allora ne viene restituita una in maniera non deterministica.

2.4.2 Collocazione e Mobilità

Le tuple spaziali possono essere associate a regioni sia direttamente che indirettamente. Le primitive introdotte precedentemente associano direttamente una tupla spaziale t a una regione r . In *Spatial Tuples*, tuttavia, una tupla può essere associata ad una entità del sistema posizionata nello spazio fisico. Da ora in avanti, queste entità saranno chiamate *componenti situati* [3] (o collocati). Quindi, se **id** identifica un componente in un sistema Spatial Tuples, ed **id** è un componente situato, una tupla spaziale t può essere associata a **id** per mezzo di una operazione **out**

$$\text{out}(t @ \text{id})$$

che associa **t** a qualunque sia la regione **r** dove **id** è collocato.

Uno dei concetti principali che si vuole mettere in luce è che un'associazione indiretta fatta in questo modo permane anche quando la regione dove **id** è collocato cambia nel tempo: fintanto che non viene rimossa da una operazione **in**, **t** è associata a qualunque sia la posizione nello spazio in cui **id** è collocato durante il suo ciclo di vita.

In alcuni casi può capitare che gli agenti che eseguono le operazioni siano essi stessi associati a componenti situati. Si consideri ad esempio l'agente assistente personale in esecuzione sullo smartphone dell'utente. In questo caso, vengono introdotte due parole chiave predefinite – **here** e **me** – ad indicare il componente situato dell'agente. Quindi, le operazioni

$$\text{out}(t @ \text{here}) \quad \text{out}(t @ \text{me})$$

associano la tupla **t** alla regione **r** in cui il componente situato dell'agente è collocato attualmente. Nel primo caso (**here**), la tupla spaziale è associata permanentemente ad **r**, anche se l'agente (il componente situato) si muove. Nel secondo caso (**me**), la regione della tupla spaziale cambia in base al movimento dell'agente (il componente situato).

Allo stesso modo, le operazioni

$$\text{rd}(tt @ \text{here}) \quad \text{rd}(tt @ \text{me}) \quad \text{in}(tt @ \text{here}) \quad \text{in}(tt @ \text{me})$$

ritornano una tupla che fa match con il template tt se e quando disponibile nella posizione corrente del componente situato che ospita l'agente.

Come generalizzazione delle primitive descritte, un agente potrebbe voler collocare o cercare tuple spaziali specificando una regione che potrebbe non essere esattamente una di quelle già definite, ma una funzione di esse. Per esempio, un agente potrebbe voler rendere una tupla osservabile in una regione circolare di raggio specificato. Per gestire questo caso generale, viene introdotto il costrutto **region(id,s)**, dove id è l'identificatore del componente situato e s è una funzione che definisce la forma della regione, che prende come parametro la regione del componente situato e che ritorna la regione estesa.

2.4.3 Formalizzazione

Sempre secondo [3], una regione ρ viene definita come una porzione di spazio non vuota e senza buchi. Ogni punto dello spazio è una possibile regione, e anche l'intero spazio è considerato una regione.

Si definisce poi una funzione di estensione ext utilizzata per specificare la regione di una tupla spaziale basandosi sulla posizione del componente. Per esempio, una possibile funzione di estensione d_l in uno spazio con sistema di riferimento metrico, prende una regione ρ (in cui un componente inserisce una tupla) e produce l'insieme di punti la cui distanza da ρ è minore di l unità di spazio (in cui la tupla è posizionata e visibile).

Oltre alle meta-variabili ρ ed ext appena descritte, la variabile id spazia tra gli identificatori dei componenti, e t sul contenuto delle tuple. Una tupla situata τ è il contenuto di una tupla (t) accoppiata con la specifica di una regione r :

$$\tau ::= t @ r$$

La specifica di una regione combina una posizione (sia una regione definita ρ o la posizione dinamica di un componente id) e una funzione di estensione ext , che può essere valutata in fase di esecuzione per determinare la regione vera e propria della tupla situata:

$$\begin{aligned} r &::= \text{region}(l, ext) \\ l &::= \rho \mid id \end{aligned}$$

Le variabili speciali here e me possono essere usate rispettivamente per intendere “la regione corrente del componente” e “l’ id del componente”.

Per quel che riguarda il matching, dal momento che una tupla situata è composta dalla coppia contenuto della tupla-specifica della regione, il matching deve avvenire per entrambi gli elementi della coppia: il contenuto della tupla è controllato tramite l'unificazione sintattica, mentre le regioni devono sovrapporsi. Essenzialmente, due tuple situate fanno match se le loro regioni si sovrappongono e i loro contenuti possono essere sostituiti l'uno con l'altro; le regioni sono ottenute semplicemente dalla specifica della regione tramite una funzione di valutazione, che in altre parole applica la funzione di estensione alla posizione corrente. Per gestire le regioni che si muovono insieme ad un componente, espresse tramite le specifiche di regione associate ad un *id*, la funzione di valutazione esegue anche la sostituzione di qualunque *id* con la sua regione ρ .

Capitolo 3

Spatial Tuples in Unity3D

Questo capitolo è il fulcro della tesi, in cui viene fornito un modello per Spatial Tuples all'interno di Unity3D e nel quale vengono spiegate le soluzioni proposte per far fronte alle problematiche sorte durante il suo sviluppo.

L'obiettivo del lavoro svolto è quello di sfruttare ed estendere le API di Cerbara [6] e Poli [5] precedentemente descritte, permettendo al programmatore di utilizzare il modello Spatial Tuples in fase di programmazione di un agente BDI costruito con la libreria di Poli. L'idea principale è quella di poter utilizzare le primitive previste da Spatial Tuples direttamente all'interno di un piano di un agente scritto in Prolog.

Come già affermato nella Sezione 2.4, Spatial Tuples prevede tre tipi di primitive:

- `out(t @ r)` che emette la tupla `t` nella regione `r`;
- `in(t @ r)` che legge e cancella la tupla `t` nella regione `r`;
- `rd(t @ r)` che legge ma non cancella la tupla `t` dalla regione `r`.

La regione, nella sua forma più generica, è sintatticamente espressa come `region(l, ext)`, dove `l` può essere il nome di una regione, o di un'altra entità che si muove nel MAS (ad esempio un altro agente), un punto preciso dello spazio espresso tramite latitudine e longitudine, un indirizzo amministrativo, o una delle variabili speciali `me` e `here`; `ext` definisce invece la forma e l'estensione della regione, ovvero fino a dove la tupla sarà osservabile.

L'organizzazione del capitolo è la seguente: la Sezione 3.1 si occupa di mostrare come si è affrontato il problema della gestione dei punti precisi dello

spazio specificati tramite coordinate geografiche o indirizzo amministrativo – ossia tutto ciò che riguarda il mapping dello spazio fisico reale all'interno di Unity3D.

La Sezione 3.2 tratta l'implementazione delle primitive previste da Spatial Tuples, lato Prolog e lato scripting C# su Unity3D.

La Sezione 3.3 mostra i pattern di coordinazione spaziale previsti da Spatial Tuples all'opera su Unity3D.

Infine, la Sezione 3.4 illustra la risoluzione di un caso di studio tipico tramite i pattern e le primitive precedentemente definiti.

3.1 Il Mapping dello Spazio Fisico Reale su Unity3D

Spatial Tuples introduce un nuovo modello basato su tuple in cui il singolo pezzo di informazione – la tupla spaziale – ha una posizione e un'estensione nello spazio fisico: ossia, in qualunque momento una tupla è legata ad una regione dello spazio. Pertanto, quest'ultimo è considerato un concetto di prima classe dell'intero modello, in grado di migliorare la comunicazione esprimendo diversi pattern di coordinazione spaziale in maniera naturale ed efficace.

Una delle cose mancanti in Unity3D ma fondamentale per cercare di proporre un modello concreto per Spatial Tuples è la possibilità di mappare le coordinate del mondo reale all'interno della scena di Unity, ovvero un modo per convertire *latitudine* e *longitudine* (e *altitudine*) in coordinate *x* e *y* (e *z*).

3.1.1 WRLD SDK

Questa mancanza è colmata da un SDK scaricabile direttamente dall'Asset Store di Unity3D: **WRLD** [18]. WRLD fornisce mappe 3D costruite usando dati geografici di alta qualità da poter utilizzare per la creazione di visualizzazioni 3D, esecuzione di simulazioni, o per lo sviluppo di giochi o esperienze dinamiche, basati sulla posizione geografica. Il servizio offre il proprio SDK su tutte le più famose piattaforme, mobile e non: Android, iOS, JavaScript per applicazioni web e Unity3D. Le mappe 3D messe a disposizione da WRLD sono già pronte all'uso, texturizzate, e basate su un sistema di coordinate real-world.

Per iniziare ad utilizzare WRLD su Unity3D è sufficiente seguire pochi passi:

- Importare l'SDK nel proprio progetto dopo averlo scaricato dall'Asset Store di Unity3D;
- Creare una nuova scena;
- Creare un nuovo GameObject e aggiungere ad esso il componente **Wrld Map**;
- Incollare la propria API key – ottenuta registrandosi gratuitamente su <https://wrld3d.com/> – nel campo corretto del componente Wrld Map appena aggiunto;
- Impostare il campo Camera del componente Wrld Map come camera principale della scena e spuntare l'opzione Use Built-in Camera Controls per abilitare lo scrolling nella mappa tramite mouse o touch su dispositivi mobile.

Per poter posizionare un GameObject sulla mappa utilizzando latitudine e longitudine, è necessario creare un GameObject padre e aggiungergli un componente **GeographicTransform** grazie al quale è possibile specificare latitudine, longitudine e la direzione iniziale espressa in gradi. Il "corpo" del GameObject, quindi ciò che si vedrà sulla mappa, è rappresentato dal GameObject figlio di GeographicTransform, che potrà avere qualunque componente classico di Unity: Collider, Rigidbody, etc. Nelle Figure 3.1 e 3.2 si può osservare la gerarchia appena descritta.

In sintesi, GeographicTransform è usato per posizionare un GameObject sulla mappa definendo latitudine e longitudine. Il suo compito è quello di mantenere l'oggetto correttamente posizionato e orientato a prescindere dal sistema di coordinate o della posizione della camera utilizzata dalla mappa. Questo GameObject funge da contenitore per il GameObject figlio che potrà essere posizionato e spostato normalmente.

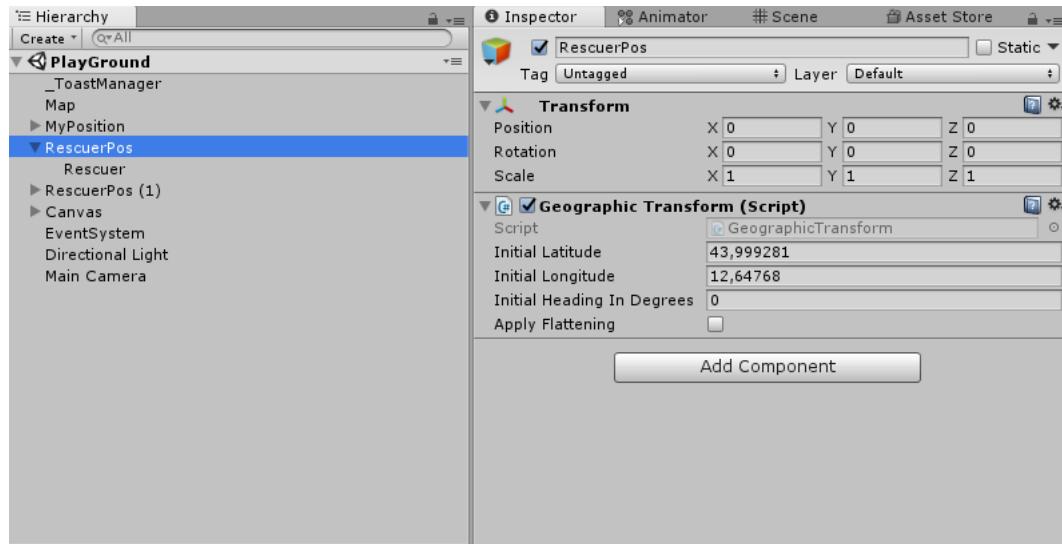


Figura 3.1: GameObject padre con componente GeographicTransform

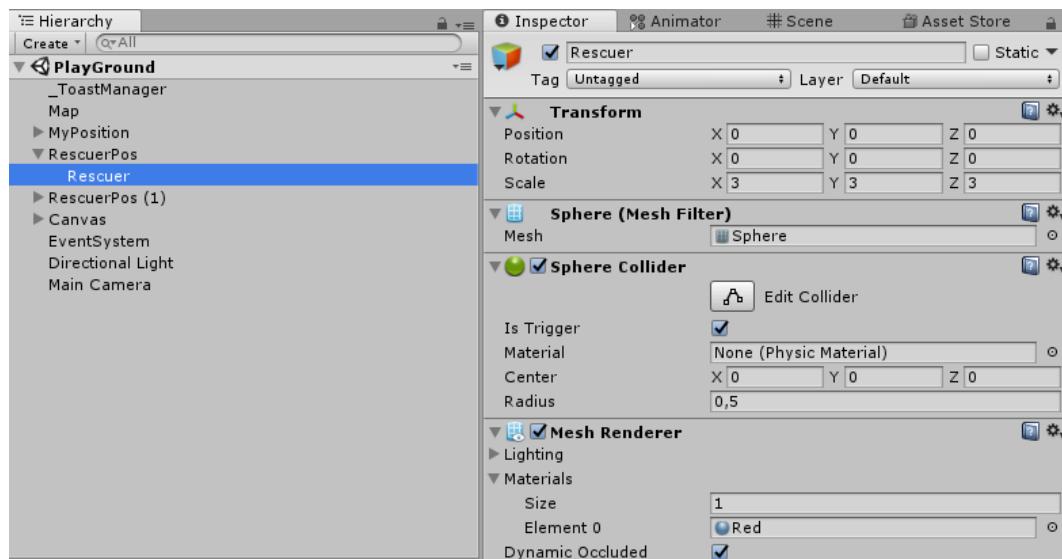


Figura 3.2: GameObject figlio a cui vengono applicati tutti i componenti Unity standard necessari

3.1.2 Geocoding

La conversione *coordinate real world - Unity3D* non è l'unico problema che emerge: Spatial Tuples prevede l'utilizzo di informazioni spaziali di qualsiasi tipo, che possono andare dalle coordinate GPS ad indirizzi amministrativi, e sono proprio quest'ultimi a creare nuovi interrogativi, dal momento che WRLD non fornisce funzionalità di *geocoding*.

Il geocoding è il processo di conversione di un indirizzo testuale – ad esempio: Via Sacchi 3, Cesena – in coordinate geografiche, ossia latitudine e longitudine. Per poter dare al modello di Spatial Tuples proposto la possibilità di usare primitive su indirizzi amministrativi si rivela quindi necessario l'utilizzo di un servizio esterno che dia la possibilità di effettuare la geocodifica.

Funzionalità di questo tipo sono offerte da *Google* con le proprie *Geocoding API*, e da *Nominatim*, che sfrutta i dati di *OpenStreetMap (OSM)*. Di seguito verranno messe a confronto queste due piattaforme:

- Entrambe erogano il servizio sotto forma di web service esponendo un'interfaccia per richiedere i dati di cui si ha bisogno, tramite HTTP, e rispondono con i dati in formato JSON o XML, a discrezione dell'utente che ne fa uso.
- Le Geocoding API di Google hanno dei limiti di utilizzo oltre i quali è necessario pagare delle quote¹, mentre Nominatim è in esecuzione su server donati, di conseguenza ha capacità decisamente minori².
- Nominatim, contrariamente a ciò che offre Google, è installabile ed esegibile su server di proprietà dell'utente, annullando così i limiti di utilizzo appena menzionati³.

¹Il piano standard prevede 2,500 richieste giornaliere gratuite, calcolate come la somma delle richieste client-side e server-side. 50 richieste al secondo, calcolate nella stessa maniera. Una volta superata la soglia giornaliera, saranno richiesti 0.50\$ per 1000 richieste giornaliere addizionali, fino ad un massimo di 100,000.

Il piano premium prevede invece 100,000 richieste al giorno e un limite iniziale espandibile di 50 richieste al secondo server-side. <https://developers.google.com/maps/documentation/geocoding/usage-limits>.

²Si raccomanda di evitare utilizzi pesanti, rimanendo nell'ordine di 1 richiesta al secondo <https://operations.osmfoundation.org/policies/nominatim/>.

³Guida all'installazione: <http://nominatim.org/release-docs/latest/admin/Installation/>

- Per utilizzare le API di Google, in generale, è necessario avere un account e richiedere una API key, per utilizzare Nominatim non è necessaria nessuna registrazione.

L'utilizzo di Nominatim sarà quindi analogo a quello di Google Geocoding API, lasciando al designer la scelta finale tra i due. Dal momento che non si dispone di server su cui poter installare un'istanza di Nominatim per abbattere i limiti particolarmente stringenti che impone la relativa interfaccia web pubblica, d'ora in avanti si prenderà in considerazione solo il geocoding offerto dal piano standard di Google, più che sufficiente per un deploy su scala ristretta previsto in questa dissertazione.

La gestione delle richieste e delle risposte web in Unity

Per ottenere il contenuto di una pagina web in Unity si utilizza la classe `WWW`. Si fa partire il download in background chiamando `WWW(url)`, il quale ritorna un nuovo oggetto di tipo `WWW`. Per sapere se il download è terminato, è possibile controllare la proprietà `isDone`. Per evitare che Unity si blocchi in attesa di ricevere la risposta, si sfrutta una coroutine, utilizzando l'istruzione `yield` all'interno di un ciclo che controlla il termine del download.

Una volta che il download è terminato, si effettua un'operazione di parsing sul JSON ottenuto come risposta e si salvano le informazioni di cui si ha bisogno – in questo caso latitudine, longitudine e una forma correttamente formattata dell'indirizzo cercato.

Per definire una qualsiasi coroutine in Unity è necessario esplicitare `IEnumerator` come tipo di ritorno, impedendo di fatto di ritornare un valore in maniera esplicita deciso a priori dal programmatore. Per risolvere il problema, si utilizza il meccanismo delle closure/callback sfruttando il tipo delegato `Action` messo a disposizione da C#.

Un delegato in C# è un tipo che rappresenta riferimenti ai metodi con un elenco di parametri e un tipo restituito particolari. Quando si crea un'istanza di un delegato, è possibile associare l'istanza a qualsiasi metodo con una firma compatibile e un tipo restituito. Tramite l'istanza di delegato è possibile richiamare (o chiamare) il metodo. I delegati vengono utilizzati per passare metodi come argomenti ad altri metodi. Un tipo delegato `Action` ha la particolarità di non ritornare un valore: in altre parole, `Action` può encapsulare un metodo che non ritorna nessun valore.

Si ipotizzi di voler restituire un booleano da una coroutine, l'implementazione tramite `Action` risulterà la seguente:

```

1 void Foo()
2 {
3     StartCoroutine(Bar((myReturnValue) => {
4         if(myReturnValue) { ... }
5     }));
6
7 }
8
9 IEnumerator Bar(System.Action<bool> callback)
10 {
11     yield return null;
12     callback(true);
13 }
```

Listato 3.1: Utilizzo delle callback per ritornare valori dalle coroutine.

Come si può notare, alla coroutine `Bar` viene passata come parametro una funzione anonima sotto forma di espressione lambda che accetta un parametro. Questa funzione viene richiamata al termine della coroutine `Bar`, passando ad essa un parametro di tipo booleano, ovvero `myReturnValue`.

Il Listato 3.2 mostra l'utilizzo di questi semplici ma fondamentali concetti necessari ad eseguire richieste HTTP in maniera asincrona senza perdere frame all'interno di Unity3D. Il parametro `callback` è la funzione che verrà richiamata non appena il parsing del JSON è terminato (oppure se ci sono stati degli errori nella richiesta HTTP) e che a sua volta ha come parametro un oggetto contenente la stringa dell'indirizzo cercato correttamente formattato e un wrapper delle coordinate latitudine-longitudine.

```

1 public static IEnumerator GetGeocodingFromGoogle(string address,
2     Action<Tuple<string, LatLong>> callback)
3 {
4     string url =
5         "https://maps.googleapis.com/maps/api/geocode/json?address=" +
6         address + "&key=" + API_KEY;
7     WWW www = new WWW(url);
8     while (!www.isDone)
9     {
10         yield return null;
11     }
12     if (www.error == null)
13     {
14         JSONNode j = JSON.Parse(www.text);
15         string formattedAddress =
16             j["results"][0]["formatted_address"].Value;
```

```

13     double lat =
14         j["results"][0]["geometry"]["location"]["lat"].AsDouble;
15     double lng =
16         j["results"][0]["geometry"]["location"]["lng"].AsDouble;
17     callback(Tuple.Create(formattedAddress, new LatLong(lat, lng)));
18 }
19 else
20 {
21     callback(null);
}

```

Listato 3.2: Richiesta di latitudine e longitudine a Google Geocoding APIs, all'interno di una coroutine.

Utilizzando la stessa tecnica appena descritta, si ottiene anche l'altitudine, terza e ultima coordinata spaziale necessaria per poter posizionare correttamente un qualsiasi GameObject sulla mappa di WRLD:

```

1 public static IEnumerator GetAltitudeFromGoogle(double latitude, double
2     longitude, Action<double> callback)
{
3     string url =
4         "https://maps.googleapis.com/maps/api/elevation/json?locations=" +
5         latitude + "," + longitude + "&key=" + API_KEY;
6     WWW www = new WWW(url);
7     while (!www.isDone)
8     {
9         yield return null;
10    }
11    if (www.error == null)
12    {
13        JSONNode j = JSON.Parse(www.text);
14        callback(j["results"][0]["elevation"].AsDouble);
15    }
16    else
17    {
18        callback(0);
19    }
}

```

Listato 3.3: Richiesta dell'altitudine di un luogo a Google Elevation APIs partendo da latitudine e longitudine, all'interno di una coroutine.

Anche per l'altitudine ci si è affidati a Google, utilizzando le *Elevation API*, ma anche in questo caso è presente un servizio analogo basato su OSM, chiamato *Open-Elevation*⁴, caratterizzato dagli stessi pregi e difetti di Nominatim

⁴<https://open-elevation.com/>

precedentemente definiti⁵.

3.2 Primitive

Uno dei desiderata per le API che si andranno a fornire, è che la sintassi delle primitive previste da Spatial Tuples – e il relativo comportamento – sia il più simile possibile a come descritto dalla letteratura [3]. Nel corso di questa sezione verranno descritte tutte le varianti implementate e rese disponibili. Come si potrà notare, l'operatore `@` usato per specificare la regione, è stato sostituito dall'operatore `at`, questo perché `@` è stato già utilizzato da Poli [5] per uno scopo diverso. La prima modifica necessaria per utilizzare delle nuove API all'interno di un agente Prolog, è stata quella di modificare parte del reasoner. La modifica consiste in un'estensione del predicato `extract_task/1`, il cui scopo è quello di verificare la lista di azioni scritte nel piano di un agente, un elemento alla volta. In sintesi, è il predicato che interpreta gli operatori `act` e `cr` menzionati nella sottosezione 2.3.1. L'estensione è volta a rendere il reasoner capace di interpretare anche le primitive di Spatial Tuples.

```

1  extract_task(A) :-
2    (A = [use_artifact(Ref,Action)|N], !,
3     use_artifact(Ref,Action,Ret),
4     append(Ret,N,Res),
5     set_active_task(Res))
6   ;
7   (A = [cr (@Ref,M)|N], !,
8    set_active_task((@Ref,M,N)))
9   ;
10  (A = [cr M|N], !,
11   set_active_task((M,N)))
12  ;
13  (A = [out(M)|N], !,
14   manage_out_primitive(M,N))
15  ;
16  (A = [in(M)|N], !,
17   manage_in_primitive(M,N))
18  ;
19  (A = [rd(M)|N], !,
20   manage_rd_primitive(M,N))
21  ;
22  (A = [M|N],
23   M,
24   set_active_task((N))
25   ;
26   /current_desire/current:D,
27   del_desire(D)
28 ).
```

Listato 3.4: Il predicato `extract_task/1` esteso per supportare le primitive di Spatial Tuples.

⁵<https://github.com/Jorl17/open-elevation/issues/3>

La gestione di ogni singola primitiva che avviene grazie alle valutazioni dei predicati `manage_out_primitive/2`, `manage_in_primitive/2` e `manage_rd_primitive/2` osservabili nel Listato 3.4 sarà illustrata nelle prossime sottosezioni del capitolo.

3.2.1 Primitiva out

```

1 manage_out_primitive(M,N) :-
2   M = C at R,
3   Ref is '$'LindaCoordination.Linda4SpatialTuples',
4   (
5     R = region(L, Ext), !,
6     (
7       L = coord(Lat, Long, Name),
8       set_active_task(@(Ref, 'SendMessageToCoordinates'(Lat, Long, Name,
9                     Ext, C), N))
10      ;
11      L = here,
12      call_method(Ref, 'SendMessageInCurrentRegion'($this, Ext, C), _),
13      set_active_task((N))
14      ;
15      L = me,
16      call_method(Ref, 'SendMessageToRegionAroundMe'($me, Ext, C), _),
17      set_active_task((N))
18      ;
19      string(L),
20      call_method(Ref, 'SendMessageToSpatialEntity'(L, Ext, C), _),
21      set_active_task((N))
22      ;
23      atom(L),
24      set_active_task(@(Ref, 'SendMessageToAddress'(L, Ext, C), N))
25    )
26    ;
27    R = here, !,
28    call_method(Ref, 'SendMessageInCurrentRegion'($this, square(-1), C),
29                _),
30    set_active_task((N))
31    ;
32    R = me, !,
33    call_method(Ref, 'SendMessageToRegionAroundMe'($me, square(1), C), _),
34    set_active_task((N))
35    ;
36    string(R), !,
37    call_method(Ref, 'SendMessageToSpatialEntity'(R, square(-1), C), _),
      set_active_task((N))
).

```

Listato 3.5: Il predicato `manage_out_primitive/2`, incaricato a richiamare il corretto metodo o coroutine C# per rilasciare una tupla spaziale.

Il Listato 3.5 mostra in dettaglio l’implementazione del predicato Prolog incaricato di elaborare la specifica tupla spaziale, rappresentata dalla variabile logica `M`, che si vuole rilasciare nella scena di gioco.

Come dichiarato dalla sintassi formalizzata nella sottosezione 2.4.3, la tupla spaziale è formata dal contenuto della tupla accoppiata con la specifica della regione, la prima è rappresentata dalla variabile logica `c` mentre la seconda dalla variabile logica `R`. La locazione `R` può essere espressa nella maniera più “esplicita” prevista dalla letteratura oppure in maniera “implicita”.

Con la maniera esplicita si utilizza il costrutto `region(L, Ext)` in cui `L` può rappresentare:

- Un punto preciso dello spazio fisico specificato tramite latitudine e longitudine;
- La posizione corrente dell’agente, identificata tramite la parola chiave `here`;
- La posizione corrente dell’agente aggiornata ad ogni suo cambio di posizione, identificata tramite la parola chiave `me`;
- La posizione di un altro componente situato esistente di cui si conosce il nome – una regione già esistente, o un agente;
- Un luogo dello spazio fisico specificato tramite indirizzo.

La variabile logica `Ext` invece permette di specificare la forma della regione in cui rilasciare la tupla – sferica o cubica – e la relativa estensione in metri, ovvero fino a che punto la tupla sarà osservabile.

La maniera implicita invece permette di rilasciare la tupla spaziale su `here`, `me`, o su un altro componente situato esistente senza la necessità di specificare l’estensione o la forma della regione.

In base alla locazione `R` in cui si vuole rilasciare la tupla `c`, verrà chiamato il metodo o coroutine opportuno, ognuno dei quali è contenuto nella classe `Linda4SpatialTuples`. Di seguito verranno illustrati sintassi e comportamento di tutti i costrutti disponibili per la primitiva in esame, suddivisi in base alla locazione in cui poter rilasciare una tupla.

Coordinate geografiche (latitudine e longitudine)

Sintassi:

```
out(t at region(coord(lat, long, "name"), shape(radius)))
```

Comportamento: crea una `Region` denominata `name`, con centro `lat` e `long`, di forma `shape` – a scelta tra `circle` per una regione sferica e `square` per una regione cubica –, con raggio lungo `radius` metri, in cui viene inserita la tupla `t` sfruttando il metodo `LindaLibrary.Linda_OUT`. L'implementazione è effettuata sotto forma di coroutine poiché è necessario attendere la risposta della richiesta web che avviene alla chiamata della coroutine `GetAltitudeFromGoogle` prima di poter posizionare sulla scena di gioco la `Region`.

Come si può notare dalle righe da 30 a 35 del Listato 3.6, per poter mappare latitudine e longitudine viene sfruttato WRLD, nello specifico il `GameObject geographic` viene posizionato sulla scena tramite il componente `GeographicTransform`, del quale si setta la posizione tramite un oggetto di tipo `LatLong`, contenitore di latitudine e longitudine.

```

1  public static IEnumerator SendMessageToCoordinates(double latitude, double longitude, string regionName,
2          Structure extension, Structure tuple)
3  {
4      LatLong pointA = LatLong.FromDegrees(latitude, longitude);
5      float alt = 0;
6      IEnumerator enumerator = LocationUtils.GetAltitudeFromGoogle(latitude, longitude, (myReturnValue) =>
7      {
8          alt = (float)myReturnValue;
9      });
10     while (enumerator.MoveNext())
11     {
12         yield return null;
13     }
14     PrimitiveType shape = FromStringToPrimitiveType(extension.Functor.Name);
15     float ext = Convert.ToSingle(extension.Argument(0));
16     Region region = new Region(regionName, new Vector3(0f, alt, 0f), new Vector3(ext, ext < 20 ? ext : 20,
17         ext), shape, false, Quaternion.identity);
18
19     GameObject regionGo = GameObject.CreatePrimitive(region.Type);
20
21     //Setting properties for regionGo
22     ...
23
24     GameObject geographic = new GameObject(regionGo.name + "geo");
25     GeographicTransform geographicTransform = geographic.AddComponent<GeographicTransform>();
26     geographicTransform.SetPosition(pointA);
27     regionGo.transform.SetParent(geographic.transform);
28     regionGo.transform.localPosition = region.Centre;
29     regionGo.transform.localScale = region.Scale;
30
31     SituatedPassiveKB script = regionGo.AddComponent<SituatedPassiveKB>();
32     script.path = "";
33     script.InitKB();
34     yield return LindaLibrary.Linda_OUT(tuple.ToString(), script.LocalKB);
35 }
```

Listato 3.6: Il rilascio di una tupla in una posizione specificata da latitudine e longitudine, lato C#

Here

Sintassi:

```
out(t at region(here, shape(radius)))
```

Comportamento:

- Se l'agente non si trova in nessuna regione, ne viene creata una con centro la posizione dell'agente, di forma `shape` – a scelta tra `circle` per una regione sferica e `square` per una regione cubica –, con raggio lungo `radius` metri in cui viene inserita la tupla `t` sfruttando il metodo `LindaCoordinationUtilities.SendMessageToRegion`.
- Se l'agente si trova già in una regione ma `radius` non corrisponde all'estensione della regione in cui si trova, ne viene creata una con le stesse specifiche descritte nel punto precedente.
- Se invece l'agente si trova già in una regione, e la sua estensione è uguale a `radius`, non ne viene creata una nuova, ma viene effettuata una `out` sulla regione corrente in cui si trova l'agente tramite il metodo `LindaCoordinationUtilities.SendMessageToRegionName`.

È bene specificare che la regione in cui si trova attualmente l'agente è tracciata tramite i metodi `OnTriggerEnter` e `OnTriggerExit` forniti da Unity e di cui è stato effettuato l'override. Ciò permette all'agente di ignorare il fatto di trovarsi all'interno di una regione che si sposta insieme ad esso, creata ad esempio tramite `out(t at region(me, shape(radius)))` che si vedrà in seguito.

È possibile rilasciare una tupla su `here` anche tramite forma implicita:

```
out(t at here)
```

In questo caso:

- Se l'agente non si trova in nessuna regione, ne viene creata una cubica che ha come centro la posizione dell'agente, con raggio unitario (1 metro) in cui viene inserita la tupla `t` sfruttando il metodo `LindaCoordinationUtilities.SendMessageToRegion`.
- Se invece l'agente si trova già in una regione, viene fatta una `out` su di essa tramite il metodo `LindaCoordinationUtilities.SendMessageToRegionName`.

```

1  public static bool SendMessageInCurrentRegion(Agent agent, Structure extension, Structure
2      tuple)
3  {
4      PrimitiveType shape = FromStringToPrimitiveType(extension.Functor.Name);
5      float ext = Convert.ToSingle(extension.Argument(0));
6      if (ext < 0)
7      {
8          if (agent.CurrentRegion != null)
9          {
10              return LindaCoordinationUtilities.SendMessageToRegionName(
11                  tuple.ToString(), agent.CurrentRegion.name) == ReturnTypeKB.True ? true : false;
12          }
13          else
14          {
15              Vector3 regionCenter = agent.transform.position;
16              Region region = new Region(Guid.NewGuid().ToString(), regionCenter, new
17                  Vector3(1, 1, 1), shape, false, Quaternion.identity);
18              return LindaCoordinationUtilities.SendMessageToRegion(
19                  tuple.ToString(), region);
20          }
21      }
22      else
23      {
24          if (agent.CurrentRegion != null)
25          {
26              if (agent.CurrentRegion.transform.localScale.x == ext)
27              {
28                  return LindaCoordinationUtilities.SendMessageToRegionName(
29                      tuple.ToString(), agent.CurrentRegion.name) == ReturnTypeKB.True ? true : false;
30              }
31              else
32              {
33                  Vector3 regionCenter = agent.transform.position;
34                  Region region = new Region(Guid.NewGuid().ToString(), regionCenter, new
35                      Vector3(ext, ext < 20 ? ext : 20, ext), shape, false,
36                      Quaternion.identity);
37                  return LindaCoordinationUtilities.SendMessageToRegion(
38                      tuple.ToString(), region);
39              }
40          }
41          else
42          {
43              Vector3 regionCenter = agent.transform.position;
44              Region region = new Region(Guid.NewGuid().ToString(), regionCenter, new
45                  Vector3(ext, ext < 20 ? ext : 20, ext), shape, false, Quaternion.identity);
46              return LindaCoordinationUtilities.SendMessageToRegion(
47                  tuple.ToString(), region);
48          }
49      }
50  }

```

Listato 3.7: Il rilascio di una tupla su here, lato C#

Me

Sintassi:

```
out(t at region(me, shape(radius)))
```

Comportamento: crea una **Region** con centro la posizione dell'agente, di forma **shape** – a scelta tra **circle** per una regione sferica e **square** per una regione cubica –, con raggio lungo **radius** metri in cui viene inserita la tupla **t** sfruttando il metodo **LindaLibrary.Linda_OUT**. In questo caso, come accennato nella sottosezione 2.4.2, si vuole che la tupla si sposti insieme all'agente. Dal punto di vista gerarchico in Unity, la regione è quindi un **GameObject** figlio del **GameObject** rappresentante il corpo dell'agente: quando quest'ultimo si muove, la regione si muoverà insieme ad esso – si veda Figura 3.3. Se è già presente una regione creata in queste modo ed ha la stessa estensione, viene sfruttato il metodo **LindaCoordinationUtilities.Tell** per rilasciare la tupla **t** su di essa, senza creare una nuova regione.

È possibile rilasciare una tupla su **me** anche tramite forma implicita:

```
out(t at me)
```

In questo caso:

- Viene creata una regione cubica con le stesse specifiche sopra descritte, ma con raggio unitario (1 metro) in cui viene inserita la tupla **t** sfruttando il metodo **LindaLibrary.Linda_OUT**.
- Se la suddetta regione è già stata creata, la tupla **t** viene rilasciata al suo interno tramite il metodo **LindaCoordinationUtilities.Tell**.



Figura 3.3: Regione contenente una tupla rilasciata su **me** che si muove insieme ad un agente.

```

1  public static bool SendMessageToRegionAroundMe(GameObject entity,
2      Structure extension, Structure tuple)
3  {
4      PrimitiveType shape = FromStringToPrimitiveType(extension.Functor.Name);
5      float ext = Convert.ToSingle(extension.Argument(0));
6
7      foreach (GameObject regiongo in
8          GameObject.FindGameObjectsWithTag("Region"))
9      {
10         if (regiongo.name.Contains(entity.name + "region") &&
11             Math.Abs(regiongo.transform.localScale.x - ext) <= 0.7)
12         {
13             return LindaCoordinationUtilities.Tell(tuple.ToString(),
14                 regiongo);
15         }
16     }
17
18     Region reg = new Region(entity.name + "region" +
19         Guid.NewGuid().ToString(), new Vector3(0f, 0f, 0f), new
20         Vector3(ext, ext < 20 ? ext : 20, ext), shape, false,
21         Quaternion.identity);
22     GameObject region = GameObject.CreatePrimitive(reg.Type);
23
24     //Setting properties for region
25     ...
26
27     region.transform.localScale = reg.Scale;
28     region.transform.SetParent(entity.transform);
29     region.transform.localPosition = reg.Centre;
30
31     SituatedPassiveKB script = region.AddComponent<SituatedPassiveKB>();
32     script.path = "";
33     script.InitKB();
34     return LindaLibrary.Linda_OUT(tuple.ToString(), script.LocalKB);
35 }
```

Listato 3.8: Il rilascio di una tupla su me, lato C#

Componente situato

Sintassi:

```
out(t at region("name", shape(radius)))
```

Comportamento: viene cercato all'interno della scena il GameObject di nome `name`

- Se non viene trovato, la tupla `t` va perduta.
- Se viene trovato ed è di tipo `Region` controlla che la sua estensione sia uguale a `radius`. In caso affermativo inserisce la tupla `t` al suo interno, altrimenti crea una nuova `Region` con lo stesso centro, forma `shape` ed estensione `radius` e inserisce la tupla `t` al suo interno.
- Se il GameObject trovato non è di tipo `Region`, viene creata una `Region` figlia del GameObject trovato, in maniera analoga a quella descritta precedentemente per la primitiva `out(t at region(me, shape(radius)))`, che si sposta insieme al componente `name`.

È possibile rilasciare una tupla su un componente situato anche tramite forma implicita:

```
out(t at "name")
```

In questo modo si cerca il GameObject di nome `name`

- Se non viene trovato, la tupla va persa.
- Se viene trovato ed è di tipo `Region`, la tupla `t` viene inserita al suo interno senza ulteriori controlli su forma o estensione.
- Se il GameObject trovato non è di tipo `Region`, viene creata una `Region` cubica e di estensione unitaria (1 metro) figlia del GameObject trovato, in maniera analoga a quella descritta precedentemente per la primitiva `out(t at region(me, shape(radius)))`, che si sposta insieme al componente `name`.

```

1  public static bool SendMessageToSpatialEntity(string name, Structure
2      extension, Structure tuple)
3  {
4      PrimitiveType shape = FromStringToPrimitiveType(extension.Functor.Name);
5      float ext = Convert.ToSingle(extension.Argument(0));
6
7      GameObject gameObject = GameObject.Find(name);
8      if (gameObject != null)
9      {
10         if (gameObject.tag == "Region")
11         {
12             if (Math.Abs(gameObject.transform.localScale.x - ext) <= 0.7 || ext < 0)
13             {
14                 return LindaCoordinationUtilities.Tell(tuple.ToString(),
15                     gameObject);
16             }
17             else
18             {
19                 Region region = new Region(Guid.NewGuid().ToString(),
20                     gameObject.transform.position, new Vector3(ext, ext < 20
21                         ? ext : 20, ext), shape, false, Quaternion.identity);
22                 return LindaCoordinationUtilities.SendMessageToRegion(
23                     tuple.ToString(), region);
24             }
25         }
26         else
27         {
28             return SendMessageToRegionAroundMe(gameObject, ext < 0 ? new
29                     Structure(extension.Functor.Name, 1) : extension, tuple);
30         }
31     }
32     else
33     {
34         return false;
35     }
36 }
```

Listato 3.9: Il rilascio di una tupla su un componente situato, lato C#

Indirizzo

Sintassi:

```
out(t at region('address', shape(radius)))
```

Comportamento: l'indirizzo passato alla primitiva viene convertito in latitudine e longitudine tramite il geocoding descritto nella sottosezione 3.1.2, il quale restituisce anche l'indirizzo formattato in maniera corretta – ad esempio se `address` è ‘*Ospedale Bufalini Cesena*’, la sua forma correttamente formattata sarà ‘*Viale Giovanni Ghirotti, 286, 47521 Cesena FC, Italia*’ – che verrà utilizzato come nome per la regione che si andrà a creare. In questo modo, è possibile controllare se esiste già una regione che ha come nome lo stesso indirizzo formattato richiamando il metodo del Listato 3.9 già descritto precedentemente e in caso negativo se ne crea una nuova nella posizione geocodificata richiamando la coroutine del Listato 3.6, di forma `shape` – a scelta tra `circle` per una regione sferica e `square` per una regione cubica – e con raggio lungo `radius` metri in cui viene inserita la tupla `t`.

```

1  public static IEnumerator SendMessageToAddress(object address, Structure extension,
2  											 Structure tuple)
3  {
4  	Tuple<string, LatLong> addressTuple = null;
5  	IEnumerator enumeratorLatLong = LocationUtils.GetGeocodingFromGoogle(address.ToString(),
6  							 (myReturnValue) =>
7  	{
8  		addressTuple = myReturnValue;
9  	});
10 	while (enumeratorLatLong.MoveNext())
11 	{
12  	yield return null;
13 	}
14 	string formattedAddress = addressTuple.Item1;
15 	double lat = addressTuple.Item2.GetLatitude();
16 	double lng = addressTuple.Item2.GetLongitude();
17 	if (SendMessageToSpatialEntity(formattedAddress, extension, tuple))
18 	{
19  	yield return true;
20 	}
21 	else
22 	{
23  	IEnumerator enumerator = SendMessageToCoordinates(lat, lng, formattedAddress,
24  												 extension, tuple);
25  	while (enumerator.MoveNext())
26  	{
27  	yield return null;
28  	}
29  }
30 }
```

Listato 3.10: Il rilascio di una tupla su un indirizzo, lato C#

3.2.2 Primitiva in

```

1 manage_in_primitive(M,N) :-
2   M = C at R,
3   Ref is $'LindaCoordination.Linda4SpatialTuples',
4   (
5     R = region(L, Ext), !,
6     (
7       L = coord(Lat, Long),
8       set_active_task((@Ref,
9         'RetrieveMessageFromRegionsAroundCoordinates'(Ext, C, Lat,
10        Long, $this), N))
11      ;
12      L = here,
13      set_active_task((@Ref, 'RetrieveMessageFromCurrentRegion'(Ext, C,
14        $this), N))
15      ;
16      L = me,
17      set_active_task((@Ref, 'RetrieveMessageFromRegionsAroundMe'(Ext,
18        C, $this), N))
19      ;
20      string(L),
21      set_active_task((@Ref, 'RetrieveMessageFromSpatialEntity'(L, Ext,
22        C, $this), N))
23      ;
24      atom(L),
25      set_active_task((@Ref, 'RetrieveMessageFromAddress'(L, Ext, C,
26        $this), N))
27    )
28    ;
29    R = here, !,
30    set_active_task((@Ref, 'RetrieveMessageFromCurrentRegion'(circle(0),
31      C, $this), N))
32    ;
33    R = me, !,
34    set_active_task((@Ref,
35      'RetrieveMessageFromRegionsAroundMe'(circle(1), C, $this), N))
36    ;
37    string(R), !,
38    set_active_task((@Ref, 'RetrieveMessageFromSpatialEntity'(R,
39      circle(0), C, $this), N))
40  .

```

Listato 3.11: Il predicato `manage_in_primitive/2`, incaricato a richiamare la corretta coroutine C# per leggere ed eliminare una tupla spaziale.

La primitiva *in*, così come la primitiva *rd*, viene sempre processata come una coroutine, questo per poter supportare la semantica sospensiva prevista da Linda e da Spatial Tuples. È bene specificare che allo stato attuale un agente Prolog non può eseguire piani in parallelo, quindi se è sospeso su una operazione

di *in*, o di *rd*, non può fare nulla fintanto che l'operazione è terminata – ossia finché la tupla non è stata trovata.

La ricerca di una tupla, può essere eseguita considerando diverse locazioni spaziali, esattamente come l'inserimento:

- Un punto preciso dello spazio fisico specificato tramite latitudine e longitudine;
- La posizione corrente dell'agente, identificata tramite la parola chiave `here`;
- La posizione corrente dell'agente aggiornata ad ogni suo cambio di posizione, identificata tramite la parola chiave `me`;
- La posizione di un altro componente situato esistente di cui si conosce il nome – una regione già esistente, o un agente;
- Un luogo dello spazio fisico specificato tramite indirizzo.

Anche qua è stata fornita la possibilità di utilizzare la primitiva nella sua forma implicita, permettendo di ricercare una tupla spaziale su `here`, `me`, o su un altro componente situato esistente senza la necessità di specificare l'estensione o la forma della regione di ricerca. Di seguito verranno illustrati sintassi e comportamento di tutti i costrutti disponibili per la primitiva *in* esame, in maniera analoga a come già fatto per la primitiva *out*.

Coordinate geografiche (latitudine e longitudine)

Sintassi:

```
in(t at region(coord(lat, long), shape(radius)))
```

Comportamento: cerca la tupla `t` in una zona di forma `shape` – a scelta tra `circle` per una zona sferica e `square` per una zona cubica – di centro `lat` e `long` ed estensione `radius` metri. La ricerca vera e propria avviene all'interno di tutti i `GameObject` di tipo `Region` che collidono con la zona utilizzata per la ricerca. L'agente si sospende fintanto che la tupla `t` non viene trovata. Quando ciò avviene, essa viene eliminata dalla relativa `Region`.

Utilizza i metodi `LindaCoordinationUtilities.GetSituatedObjectsFromArea` e `LindaCoordinationUtilities.Retrieve`.

```

1  public static IEnumerator RetrieveMessageFromRegionsAroundCoordinates(Structure extension,
2      Structure tuple, double latitude, double longitude, Agent agent)
3  {
4      PrimitiveType shape = FromStringToPrimitiveType(extension.Functor.Name);
5      float radius = Convert.ToSingle(extension.Argument(0));
6      float altitude = 0;
7      IEnumerator enumerator = LocationUtils.GetAltitudeFromGoogle(latitude, longitude,
8          (myReturnValue) =>
9      {
10         altitude = (float)myReturnValue;
11     });
12     while (enumerator.MoveNext())
13     {
14         yield return null;
15     }
16     GameObject geographic = new GameObject(Guid.NewGuid().ToString());
17     GeographicTransform geographicTransform = geographic.AddComponent<GeographicTransform>();
18     geographicTransform.SetPosition(new LatLong(latitude, longitude));
19     GameObject inner = new GameObject(Guid.NewGuid().ToString());
20     inner.transform.SetParent(geographic.transform);
21     inner.transform.localPosition = new Vector3(0f, altitude, 0f);
22
23     Vector3 location = inner.transform.position;
24     while (location.x == 0 && location.z == 0)
25     {
26         location = inner.transform.position;
27         yield return null;
28     }
29
30     bool found = false;
31     GameObject[] regionsInRange;
32     while (!found)
33     {
34         regionsInRange = LindaCoordinationUtilities.GetSituatedObjectsFromArea(location,
35             shape, radius, 50);
36         yield return null;
37         foreach (GameObject region in regionsInRange)
38         {
39             if (LindaCoordinationUtilities.Retrieve(tuple.ToString(), region))
40             {
41                 found = true;
42                 agent.TupleFoundIn = region;
43                 break;
44             }
45             yield return null;
46         }
47     }
}

```

Listato 3.12: La ricerca di una tupla partendo da una posizione specificata da latitudine e longitudine, lato C#

Here

Sintassi:

```
in(t at region(here, shape(radius)))
```

Comportamento: se l'agente che effettua l'operazione si trova all'interno di una `Region`, la tupla `t` viene cercata in una zona di forma `shape` – a scelta tra `circle` per una zona sferica e `square` per una zona cubica – che ha centro coincidente con il centro della `Region` in questione ed estensione la somma tra l'estensione della `Region` e `radius`. Se l'agente non si trova in nessuna `Region`, il centro di ricerca considerato è la posizione corrente dell'agente, mentre l'estensione considerata è `ext`. La ricerca vera e propria avviene all'interno di tutti i `GameObject` di tipo `Region` che collidono con la zona utilizzata per la ricerca. L'agente si sospende fintanto che la tupla `t` non viene trovata. Quando ciò avviene, essa viene eliminata dalla relativa `Region`.

Utilizza i metodi `LindaCoordinationUtilities.GetSituatedObjectsFromArea` e `LindaCoordinationUtilities.Retrieve`.

È possibile ricercare una tupla su `here` anche tramite forma implicita:

```
in(t at here)
```

In questo caso la tupla viene ricercata nella zona sferica avente centro ed estensione coincidenti con quelli della regione in cui si trova attualmente l'agente.

```

1 public static IEnumerator RetrieveMessageFromCurrentRegion(Structure
2     extension, Structure tuple, Agent agent)
3 {
4     Vector3 location;
5     PrimitiveType shape = FromStringToPrimitiveType(extension.Functor.Name);
6     float radius = Convert.ToSingle(extension.Argument(0));
7     GameObject currentRegion = agent.CurrentRegion;
8
9     if (currentRegion != null)
10    {
11        location = currentRegion.transform.position;
12        radius += currentRegion.transform.localScale.x;
13    }
14    else
15    {
16        location = agent.transform.position;
17    }
18
19    bool found = false;
20    GameObject[] regionsInRange;
21    while (!found)
```

```
21  {
22      regionsInRange =
23          LindaCoordinationUtilities.GetSituatedObjectsFromArea(location,
24              shape, radius, 50);
25      yield return null;
26      foreach (GameObject region in regionsInRange)
27      {
28          if (LindaCoordinationUtilities.Retrieve(tuple.ToString(),
29              region))
30          {
31              found = true;
32              agent.TupleFoundIn = region;
33              break;
34          }
35      }
36 }
```

Listato 3.13: La ricerca di una tupla su here, lato C#

Me

Sintassi:

```
in(t at region(me, shape(radius)))
```

Comportamento: cerca la tupla `t` in una zona di forma `shape` – a scelta tra `circle` per una zona sferica e `square` per una zona cubica – che ha come centro la posizione dell’agente ed estensione `radius`. La ricerca vera e propria avviene all’interno di tutti i `GameObject` di tipo `Region` che collidono con la zona utilizzata per la ricerca. L’agente si sospende fintanto che la tupla `t` non viene trovata. Quando ciò avviene, essa viene eliminata dalla relativa `Region`. Utilizza i metodi `LindaCoordinationUtilities.GetSituatedObjectsFromArea` e `LindaCoordinationUtilities.Retrieve`.

È possibile ricercare una tupla su `me` anche tramite forma implicita:

```
in(t at me)
```

In questo caso la tupla viene ricercata nella zona sferica avente centro la posizione corrente dell’agente ed estensione unitaria (1 metro).

```

1 public static IEnumerator RetrieveMessageFromRegionsAroundMe(Structure extension, Structure
2   tuple, Agent agent)
3 {
4   PrimitiveType shape = FromStringToPrimitiveType(extension.Functor.Name);
5   float radius = Convert.ToSingle(extension.Argument(0));
6
7   bool found = false;
8   Vector3 location;
9   GameObject[] regionsInRange;
10  while (!found)
11  {
12    location = agent.transform.position;
13    regionsInRange = LindaCoordinationUtilities.GetSituatedObjectsFromArea(location,
14      shape, radius, 50);
15    yield return null;
16    foreach (GameObject region in regionsInRange)
17    {
18      if (LindaCoordinationUtilities.Retrieve(tuple.ToString(), region))
19      {
20        found = true;
21        agent.TupleFoundIn = region;
22        break;
23      }
24      yield return null;
25    }
26  }
}
```

Listato 3.14: La ricerca di una tupla su me, lato C#

Componente situato

Sintassi:

```
in(t at region("name", shape(radius)))
```

Comportamento: cerca la tupla `t` in una zona di forma `shape` – a scelta tra `circle` per una zona sferica e `square` per una zona cubica – che ha come centro il centro del GameObject `name` ed estensione la somma tra l'estensione del GameObject e `radius`. La ricerca vera e propria avviene all'interno di tutti i GameObject di tipo `Region` che collidono con la zona utilizzata per la ricerca. L'agente si sospende fintanto che la tupla `t` non viene trovata. Quando ciò avviene, essa viene eliminata dalla relativa regione.

Utilizza i metodi `LindaCoordinationUtilities.GetSituatedObjectsFromArea` e `LindaCoordinationUtilities.Retrieve`.

È possibile ricercare una tupla su un componente situato anche tramite forma implicita:

```
in(t at "name")
```

In questo caso la tupla viene ricercata nella zona sferica avente centro e raggio rispettivamente la posizione e l'estensione del GameObject `name`.

```

1 public static IEnumerator RetrieveMessageFromSpatialEntity(string name, Structure extension, Structure
2     tuple, Agent agent)
3 {
4     GameObject entity = GameObject.Find(name);
5     while (entity == null)
6     {
7         entity = GameObject.Find(name);
8         yield return null;
9     }
10    Vector3 location = entity.transform.position;
11    PrimitiveType shape = FromStringToPrimitiveType(extension.Functor.Name);
12    float radius = Convert.ToSingle(extension.Argument(0)) + entity.transform.localScale.x;
13    bool found = false;
14    GameObject[] regionsInRange;
15    while (!found)
16    {
17        regionsInRange = LindaCoordinationUtilities.GetSituatedObjectsFromArea(location, shape, radius,
18        50);
19        yield return null;
20        foreach (GameObject region in regionsInRange)
21        {
22            if (LindaCoordinationUtilities.Retrieve(tuple.ToString(), region))
23            {
24                found = true;
25                agent.TupleFoundIn = region;
26                break;
27            }
28        }
29    }
30 }
```

Listato 3.15: La ricerca di una tupla su un componente situato, lato C#

Indirizzo

Sintassi:

```
in(t at region('address', shape(radius)))
```

Comportamento: cerca la tupla `t` in una zona di forma `shape` – a scelta tra `circle` per una zona sferica e `square` per una zona cubica – che ha come centro l'indirizzo `address` geocodificato ed estensione `radius`. La ricerca vera e propria avviene all'interno di tutti i `GameObject` di tipo `Region` che collidono con la zona utilizzata per la ricerca. L'agente si sospende fintanto che la tupla `t` non viene trovata. Quando ciò avviene, essa viene eliminata dalla relativa regione. Utilizza i metodi `LindaCoordinationUtilities.GetSituatedObjectsFromArea` e `LindaCoordinationUtilities.Retrieve`.

```

1  public static IEnumerator RetrieveMessageFromAddress(object address, Structure extension, Structure tuple,
2  											  Agent agent)
3  {
4  	LatLong latLng = new LatLong(0, 0);
5  	IEnumerator enumeratorLatLong = LocationUtils.GetGeocodingFromGoogle(address.ToString(),
6  								 (myReturnValue) =>
7  	{
8  	latLng = myReturnValue.Item2;
9  	});
10  while (enumeratorLatLong.MoveNext())
11  {
12  	yield return null;
13  }
14  PrimitiveType shape = FromStringToPrimitiveType(extension.Functor.Name);
15  float radius = Convert.ToSingle(extension.Argument(0));
16  //...
17  GameObject geographic = new GameObject(Guid.NewGuid().ToString());
18  //...
19  GameObject inner = new GameObject(Guid.NewGuid().ToString());
20  inner.transform.SetParent(geographic.transform);
21  //...
22  Vector3 location = inner.transform.position;
23  //...
24  bool found = false;
25  GameObject[] regionsInRange;
26  while (!found)
27  {
28  	regionsInRange = LindaCoordinationUtilities.GetSituatedObjectsFromArea(location, shape, radius,
29  							 50);
30  	yield return null;
31  	foreach (GameObject region in regionsInRange)
32  	{
33  	if (LindaCoordinationUtilities.Retrieve(tuple.ToString(), region))
34  	{
35  	found = true;
36  	agent.TupleFoundIn = region;
37  	break;
38  	}
39  	yield return null;
40  }
41  yield return null;
42 }
```

Listato 3.16: La ricerca di una tupla su un indirizzo, lato C#

3.2.3 Primitiva rd

I costrutti resi disponibili per la primitiva *rd* e relative implementazioni sono analoghi a quelli appena illustrati per la primitiva *in*, con la sola differenza che la tupla non viene eliminata una volta trovata. Viene utilizzato quindi il metodo di libreria `LindaCoordinationUtilities.Ask` al posto del metodo `LindaCoordinationUtilities.Retrieve`

Come si può notare da tutti i listati illustrati per la primitiva *in*, la `Region` in cui viene trovata la tupla cercata viene salvata come attributo C# all'interno dell'agente, in modo tale da poterla sfruttare in futuro: ad esempio, l'agente potrebbe voler spostarsi verso di essa. Questo dettaglio implementativo è mantenuto anche per la primitiva *rd*.

3.3 Pattern di Coordinazione

Per mostrare come Spatial Tuples può supportare la coordinazione degli agenti le cui azioni includono aspetti strettamente legati allo spazio, in questa sezione si mostrano i pattern di coordinazione descritti in letteratura capaci di facilitare la risoluzione di problemi tipici [3], implementati sfruttando il modello ottenuto descritto nella sezione precedente.

3.3.1 Situated Knowledge Sharing

Secondo il concetto di *situated knowledge sharing* le tuple spaziali possono essere usate per rappresentare informazioni condivise o conoscenza riguardo al mondo fisico, collocate nella regione di spazio in cui questa conoscenza è emersa.

Per esempio, se un agente **control_room** in un'operazione di soccorso ha bisogno di avvisare tutti i soccorritori della presenza di qualche persona ferita in una **region**, può invocare un'operazione

```
out(warning(injured) @ region)
```

in modo tale che tutti gli agenti soccorritori situati in **region**, che aspettano avvisi tramite un'operazione

```
rd(warning(W) @ me)
```

possano ricevere immediatamente l'informazione.

Awareness, or: *Find me!*

Un caso specifico di knowledge sharing riguarda la posizione di agenti situati che possono muoversi nello spazio. Ovvero, un agente potrebbe avere bisogno di essere consapevole se/quando altri agenti arrivano in qualche regione.

Per esempio, se un certo numero di dispositivi mobili deve convergere nella regione **meetingPlace**, un agente **meetingControl** potrebbe controllare l'arrivo di tutti i dispositivi attesi, sapendo che ognuno di essi deposita una tupla attraverso

```
out(hereIAm(id) @ here)
```

e ottenendo ognuna di esse eseguendo ripetutamente una

```
in(hereIAm(Device) @ meetingPlace)
```

fino a che tutti i dispositivi attesi non sono arrivati in zona. Da notare che mentre i dispositivi devono essere componenti situati – in modo tale che possano usare la parola chiave **here** della primitiva di Spatial Tuples –, **meetingControl** può essere un agente di qualunque tipo, dal momento che la primitiva da esso utilizzata non richiede collocazione.

L'implementazione prevede dunque due tipi di agenti: **meetingControl** non situato, e **meetingAgent** situato e la scena su Unity è stata creata con ovviamente un solo agente **meetingControl**, e cinque agenti **meetingAgent** dislocati in posizione diverse e impostati in maniera tale che si muovano con velocità diverse.

Il codice per i due tipi agenti è, rispettivamente:

```

1 belief toArrive(5).
2
3 desire work.
4
5 add work && true => [
6     cr createregion("meetingPlace"),
7     add_desire(check)
8 ].  

9
10 add check && (belief toArrive(N), N > 0) => [
11     in(hereIAm(_) at "meetingPlace"),
12     Nx is N - 1,
13     del_belief(toArrive(N)),
14     add_belief(toArrive(Nx)),
15     act showtoast("MeetingControl: agente arrivato in zona"),
16     add_desire(repeatcheck),
17     stop

```

```

18 ].  

19 add check && (belief toArrive(N), N =< 0) => [  

20   act showtoast("MeetingControl: tutti gli agenti sono arrivati in zona")  

21 ].  

22  

23 add repeatcheck && true => [  

24   add_desire(check)  

25 ].  

26

```

Listato 3.17: Codice Prolog per l'agente **meetingControl**.

```

1 desire meeting.  

2  

3 add meeting && true => [  

4   cr gotoregion("meetingPlace"),  

5   Name is $me.name,  

6   act showtoast(Name, ": sono arrivato nella regione, rilascio tupla"),  

7   out(hereIAm(Name) at here)  

8 ].  


```

Listato 3.18: Codice Prolog per l'agente **meetingAgent**.

meetingControl sa che devono arrivare cinque agenti in zona tramite il belief `toArrive(5)`, e nel primo piano che eseguirà (`work`) crea una `Region` attraverso la chiamata al metodo `createregion`, estesa 40 metri, chiamata `meetingPlace` e “vuota”, ossia senza tuple al suo interno, dopodiché eseguirà il piano `check`, e si sosponderà su

in(`hereIAm(_)` at “`meetingPlace`”)

fintanto che non troverà una tupla nella regione creata precedentemente. La tupla in questione sarà rilasciata da ciascun agente **meetingAgent** una volta che sarà arrivato nella regione `meetingPlace` con

out(`hereIAm(Name)` at `here`)

A questo punto, **meetingControl** riprende l'esecuzione del suo piano, aggiorna la sua belief base e controlla che siano arrivati tutti gli agenti. In caso affermativo l'agente ha concluso il suo lavoro, in caso negativo, ripete il piano `check` in cui attende l'arrivo del prossimo agente **meetingAgent**. In Figura 3.4 si mostrano i punti salienti della scena Unity mentre il MAS appena descritto è in esecuzione, rappresentati in cinque fotogrammi.

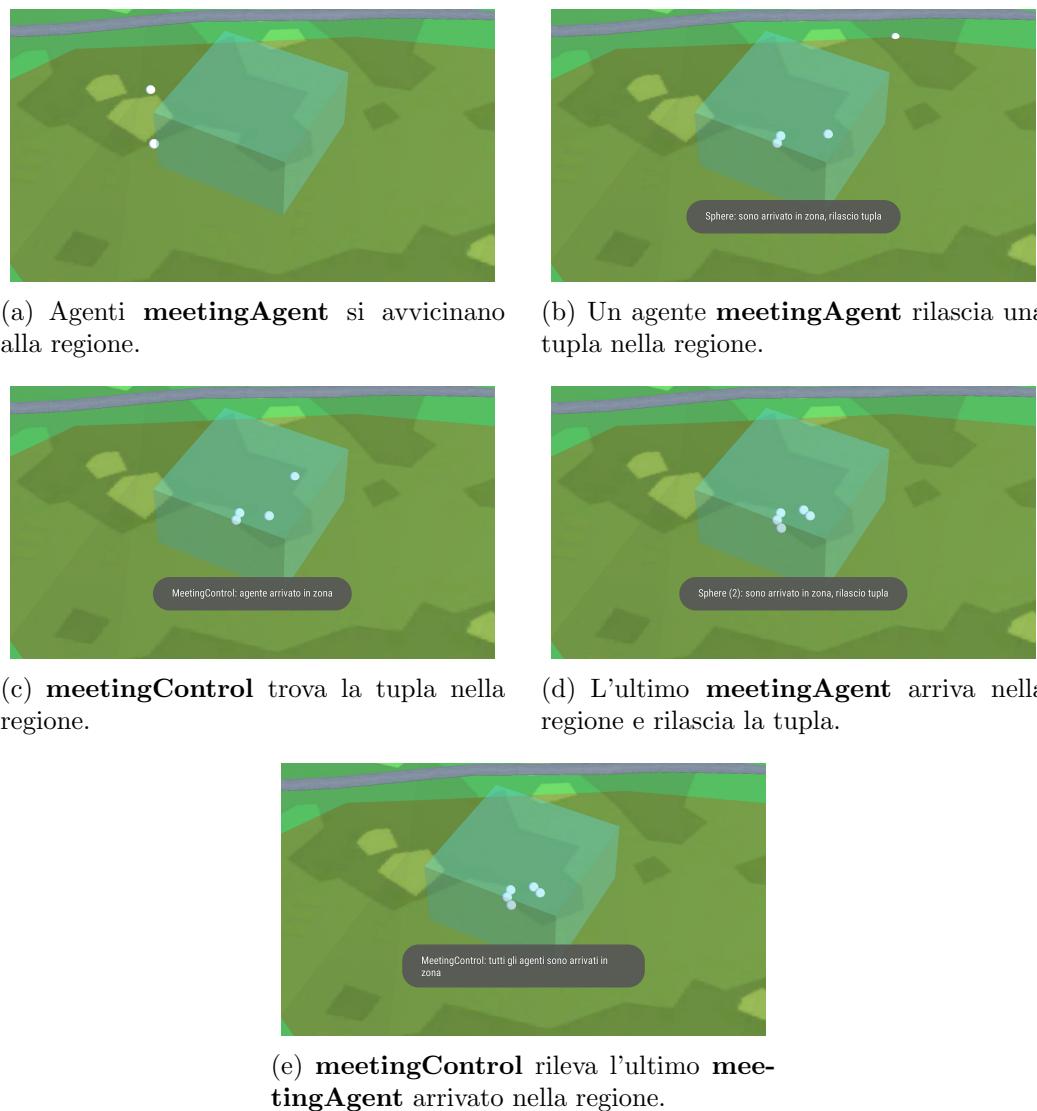


Figura 3.4: Il pattern Awareness, or: *Find me!* su Unity3D.

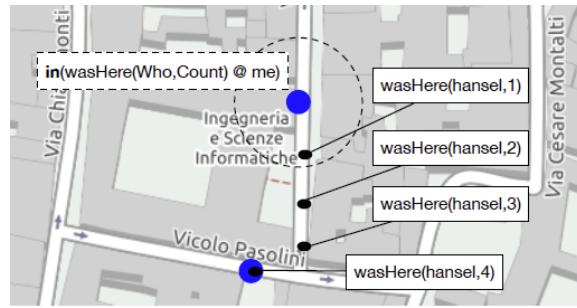


Figura 3.5: Il pattern Breadcrumbs, or: *Follow me!* come esempio di situated knowledge sharing [3].

Breadcrumbs, or: *Follow me!*

Come estensione del caso precedente, il knowledge sharing potrebbe coinvolgere il percorso di un agente situato che si muove per lo spazio. In questo caso, l'agente rilascia una traccia di tuple spaziali – *breadcrumbs* (briciole di pane) – mentre è in movimento, ognuna delle quali contiene qualche tipo di informazione sull'ordinamento, come ad esempio un contatore. In Figura 3.5 è osservabile lo scenario descritto.

Per esempio, un agente **hansel** che possiede un contatore C potrebbe depositare periodicamente una tupla spaziale wasHere/2:

out(wasHere(hansel, C) @ here)

incrementando ogni volta il contatore C. Di conseguenza, la traiettoria dell'agente **hansel** può essere rilevata osservando la distribuzione spaziale delle tuple wasHere/2.

L'implementazione prevede quindi due tipi di agenti: **hansel** e **follower**, entrambi situati. Sulla scena Unity sono presenti un agente **hansel** e un agente **follower**.

Il codice per i due agenti è, rispettivamente:

```

1 belief counter(0).
2
3 desire spreadbreadcrumbs.
4
5 add spreadbreadcrumbs && true => [
6   belief counter(C),
7   out(wasHere(hansel, C) at here),
8   Cx is C + 1,
9   del_belief(counter(C)),

```

```

10   add_belief(counter(Cx)),
11   add_desire(movealong)
12 ].
13
14 add movealong && true => [
15   cr movealongfortime(1.5),
16   add_desire(spreadbreadcrumbs)
17 ].
```

Listato 3.19: Codice Prolog per l'agente **hansel**.

```

1 belief counter(0).
2
3 desire followpattern.
4
5 add followpattern && true => [
6   belief counter(C),
7   in(wasHere(hansel, C) at region(me, square(70))),
8   Cx is C + 1,
9   del_belief(counter(C)),
10  add_belief(counter(Cx)),
11  add_desire(gotobreadcrumb)
12 ].
13
14 add gotobreadcrumb && true => [
15   cr gotobreadcrumb,
16   add_desire(followpattern)
17 ].
```

Listato 3.20: Codice Prolog per l'agente **follower**.

Come può essere osservato, **hansel** utilizza un contatore `belief counter(0)` e rilascia nella sua posizione una tupla spaziale contenente il valore corrente del contatore con la primitiva

`out(wasHere(hansel, C) at here)`

che crea una regione estesa per un metro contenente la tupla `wasHere/2`. L'agente **follower**, posizionato a poche decine di metri dalla posizione di **hansel**, rimane in attesa della tupla `wasHere(hansel, C)`, dove `C` parte da 0 e viene incrementato ogni volta che l'operazione

`in(wasHere(hansel, C) at region(me, square(70)))`

va a buon fine. La primitiva sbloccherà l'esecuzione dell'agente **follower** quando trova la tupla giusta in un raggio di 70 metri dalla sua posizione corrente, in modo tale che l'agente possa muoversi verso di essa. Dopo aver depositato

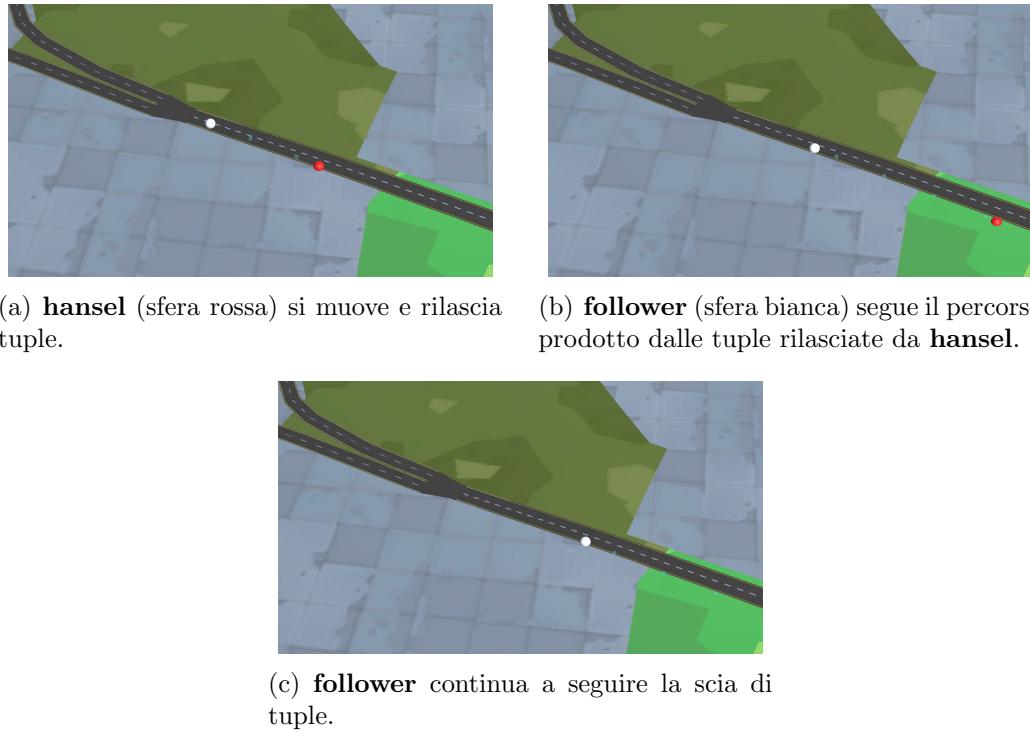


Figura 3.6: Il pattern Breadcrumbs, or: *Follow me!* su Unity3D.

una tupla, **hansel** si muove in avanti per 1.5 secondi grazie alla coroutine richiamata tramite `cr movealongfortime(1.5)` e rilascerà la tupla successiva con il valore del contatore aggiornato. In Figura 3.6 si mostrano i punti salienti della scena Unity mentre il MAS appena descritto è in esecuzione, riassunti in tre fotogrammi.

3.3.2 Sincronizzazione Spaziale e Mutua Esclusione

Per *sincronizzazione spaziale* si intende sincronizzare le azioni degli agenti basandosi sulle informazioni spaziali, in cui il mondo fisico è aumentato grazie alle tuple spaziali. Come dichiarato da [3], il caso più generale è quello in cui un agente A esegue qualche azione o task T non appena un'altra azione o task T' che si verifica in una specifica regione r viene portata a termine da qualche altro agente. L'agente A non ha bisogno di sapere chi sta eseguendo il task T' nella regione r – può essere uno o anche più agenti –, e chiunque

stia eseguendo il task T' non si deve preoccupare di sapere chi è interessato al comportamento e al risultato del task T' .

La sincronizzazione può verificarsi direttamente considerando la posizione dell'agente, per esempio un agente A potrebbe iniziare qualche task non appena un agente B arriva in un posto specifico. Per esempio, utilizzando ancora l'esempio del meeting introdotto nella sottosezione 3.3.1, un agente potrebbe decidere di abbandonare il meeting solo quando un altro agente raggiunge il luogo corrispondente attraverso un'invocazione della forma

```
rd(hereIAm(_) @ meetingPlace)
```

e aspettare che venga trovata la tupla spaziale che faccia match.

Quest'ultimo scenario descritto è stato implementato facilmente in Unity3D partendo dal MAS progettato per mostrare il pattern awareness. Il nuovo MAS avrà un nuovo tipo di agente, che si chiamerà **leavingAgent**, ossia quello che vuole abbandonare la zona dell'incontro dopo che un altro agente lo ha raggiunto. D'altra parte, non sarà più presente l'agente **meetingControl**, poiché non è più richiesto un agente incaricato a monitorare tutta la situazione all'interno della regione **meetingPlace**.

```

1 desire meeting.
2
3 add meeting && true => [
4     cr gotoregion("meetingPlace"),
5     Name is $me.name,
6     Waiting is '$sphere (3)'.name,
7     act showtoast(Name, ": sono arrivato in regione, rilascio tupla e
8         attendo ", Waiting),
9     out(hereIAm(Name) at here),
10    rd(hereIAm(Waiting) at here),
11    act showtoast(Name, ":", Waiting, " arrivato, abbandono la regione"),
12    cr gotoregion("escape")
13 ].
```

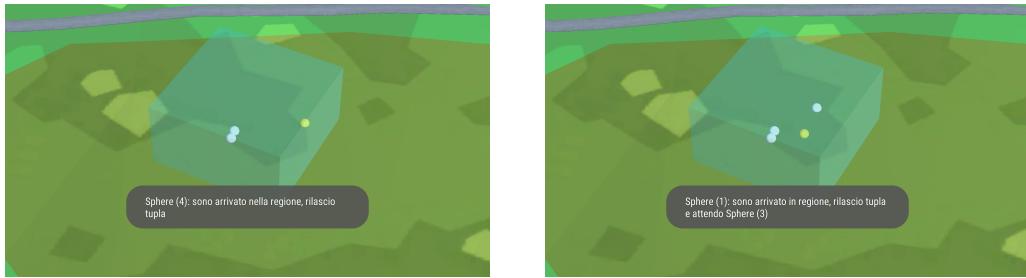
Listato 3.21: Codice Prolog per l'agente **leavingAgent**.

Come si può notare dal Listato 3.21, l'agente **leavingAgent** una volta arrivato in **meetingPlace** rilascia una tupla spaziale tramite

```
out(hereIAm(Name) at here)
```

dove **Name** corrisponde al nome del GameObject dell'agente, e si mette in attesa di una tupla spaziale su **meetingPlace** con l'operazione

```
rd(hereIAm(Waiting) at here)
```



(a) I primi agenti iniziano ad arrivare nella regione in cui rilasciano tuple spaziali.



(b) L'agente **leavingAgent** (sfera gialla, di nome **Sphere (1)**) arriva in regione, rilascia una tupla spaziale e si mette in attesa dell'agente di nome **Sphere (3)**.



(c) L'agente di nome **Sphere (3)** arriva in regione e rilascia la tupla su cui leavingAgent era in attesa. Quest'ultimo percepisce la tupla e abbandona la regione.

Figura 3.7: Il pattern *sincronizzazione spaziale* su Unity3D.

dove **Waiting** corrisponde al nome del GameObject dell'agente che si sta aspettando. È giusto ricordare che quest'ultima operazione sarebbe potuta essere invocata anche come

```
rd(hereIAm(Waiting) at "meetingPlace")
```

rispettando la forma vista poco fa, ma dal momento che l'agente si trova già nella regione **meetingPlace** al momento dell'invocazione, **meetingPlace** è interscambiabile con la parola chiave **here**. In Figura 3.7 viene mostrato il MAS appena descritto in esecuzione su Unity, rappresentato in tre fotogrammi.

Infine, la *mutua esclusione spaziale* può essere implementata in Spatial Tuples al fine di regolare l'accesso degli agenti in qualche luogo fisico: ad

esempio, per accogliere un agente alla volta all'interno di una certa regione (**mutex_region**). A questo fine, una tupla spaziale potrebbe essere usata come lock, richiedendo a qualsiasi agente che vuole entrare nella regione di ottenere una tupla **lock** collocata là, che sarà rilasciata appena l'agente esce dalla regione – rispettivamente attraverso le operazioni

in(lock @ mutex_region) out(lock @ mutex_region)

L'implementazione in Unity3D di questo pattern tramite le API per Spatial Tuples prevede un singolo tipo di agente chiamato **mutualExclusionAgent**, che si avvicina alla regione **mutex_region** – precedentemente istanziata con una tupla **lock** al suo interno – per poi provare ad ottenere la tupla **lock** su di essa a fronte di un'operazione

in(lock(_) at "mutex_region")

Una volta ottenuta la tupla, l'agente entra nella regione, per poi rilasciare la tupla **lock** al suo interno e uscire dalla regione tramite l'operazione

out(lock(0) at "mutex_region")

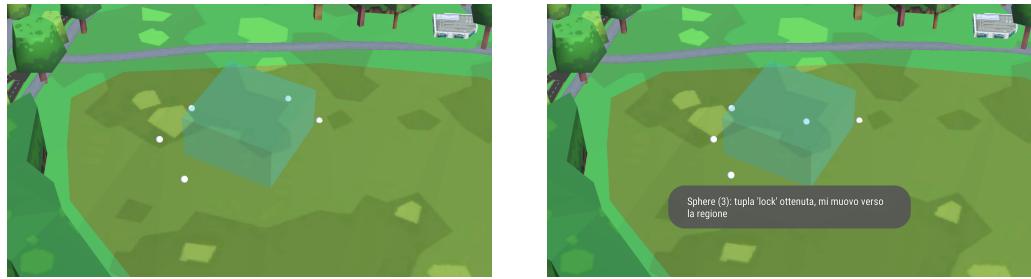
Per una migliore comprensione, il comportamento dell'agente viene riportato nel Listato 3.22, mentre il MAS appena illustrato viene mostrato nei tre fotogrammi di Figura 3.8.

```

1 desire meeting.
2
3 add meeting && true => [
4   cr getclosetoregion("mutex_region"),
5   in(lock(_) at "mutex_region"),
6   Name is $me.name,
7   act showtoast(Name, ": tupla 'lock' ottenuta, mi muovo verso la
8     regione"),
9   cr gotoregion("mutex_region"),
10  act showtoast(Name, ": sono arrivato nella regione, rilascio tupla
11    'lock' e abbandono la regione"),
12  out(lock(0) at "mutex_region"),
13  cr gotoregion("escape")
14].

```

Listato 3.22: Codice Prolog per l'agente **mutualExclusionAgent**.



(a) I cinque **mutualExclusionAgent** si avvicinano alla regione.

(b) L'agente **Sphere (3)** ottiene la tupla **lock** dalla regione grazie al quale può entrare nella regione.



(c) L'agente **Sphere (3)** rilascia la tupla **lock** nella regione in modo tale che il prossimo agente possa accedervi.

Figura 3.8: Il pattern *mutua esclusione* su Unity3D.

3.4 Caso di Studio: Rescue

Giunti a questo punto, si ha tutto il necessario per iniziare a considerare l'utilizzo del modello per alcuni problemi e sistemi reali, organizzati come MAS che sfruttano Spatial Tuples all'interno di Unity3D. In questa sezione se ne vedrà uno – che secondo [3] rappresenta l'esatta motivazione per cui è stata concepita l'intera idea di Spatial Tuples.

Il caso di studio in esame riguarda la coordinazione di un team di soccorritori in caso di disastri. In questi casi infatti, un'efficiente coordinazione delle attività dei soccorritori risulta fondamentale per ottimizzare l'utilizzo di tutte le risorse disponibili (ivi compresi gli umani), che sono tipicamente limitate [19]. Una caratteristica peculiare delle missioni di soccorso è rappresentata dal fatto che le squadre che collaborano, lavorano in un ambiente altamente dinamico e (molto probabilmente) imprevedibile.

In uno scenario del genere, è di estrema importanza condividere informazioni in modo efficiente ed efficace tra i soccorritori, e tra i soccorritori sul campo e una squadra che controlla l'andamento delle operazioni – possibilmente da remoto. Idealmente quindi, l'intera squadra di soccorritori è composta da:

1. un insieme di soccorritori che si muovono sul campo;
2. un insieme di soccorritori che controllano e supervisionano la missione da una sala di controllo remota.

Il compito principale dei soccorritori sul campo è quello di identificare e raccogliere informazioni sulle persone ferite e assegnare loro un grado di urgenza di trattamento – operazione che prende il nome di *triaging* – rendendo queste informazioni disponibili alla sala di controllo, comprese la posizione e lo stato di salute.

A sua volta, il compito principale della sala di controllo è dirigere la coordinazione globale della missione, compresa la determinazione al volo di come gestire le persone ferite, le regioni del campo da esplorare, così come le avvertenze riguardo alle specifiche posizioni del campo stesso.

Quando i soccorritori trovano una persona potenzialmente ferita, i protocolli di sicurezza richiedono di effettuare il triaging su di essa e notificare la squadra nella sala di controllo – o, in alternativa, altri soccorritori sul campo – la sua posizione e il suo stato. A tale scopo, l'agente soccorritore appone una tupla spaziale alla persona soccorsa che segnala il suo stato alla squadra

di controllo e, automaticamente, la sua posizione: ciò è possibile rilasciando una tupla spaziale tramite un'operazione **out** come

```
out(triage(rescuer(1), status('critical')) @ victim3)
```

Non appena viene trovata una nuova persona ferita, l'informazione corrispondente deve comparire su una mappa nella sala di controllo, nella corretta posizione. A questo scopo, l'agente nella sala di controllo recupera periodicamente tutte le tuple di quel tipo, con un'operazione **rd** del tipo

```
rd(triage(Who, Status) @ missionArea)
```

dove **missionArea** rappresenta tutto il campo della missione.

Inoltre, un'ulteriore funzionalità è l'implementazione di forme di comunicazione situata: nello specifico, la sala di controllo (e/o i soccorritori sul campo stessi) potrebbe voler trasmettere messaggi rilevanti solo per i soccorritori in una specifica regione sul campo, dal momento che contengono informazioni (ad esempio avvertimenti) riguardo a quella posizione.

A questo scopo, un agente in sala di controllo (o soccorritore sul campo) inserisce una tupla spaziale contenente le informazioni riguardo alla specifica regione che essa copre. Nel caso dell'agente in sala di controllo (che si ricorda non essere situato sul campo):

```
out(warning(fire) @ circleRegion(coord(lat,lon), rad))
```

dove **circleRegion** è un'espressione usata qui per semplicità per indicare una semplice regione circolare centrata nelle coordinate specificate (latitudine, longitudine) e raggio **rad**. Gli agenti soccorritori invece, essendo situati sul campo, possono fare riferimento alla loro attuale posizione fisica per mezzo di un'operazione

```
out(msg(warning(fire)) @ circleRegion(here, rad))
```

Dal punto di vista degli agenti soccorritori, per venire a conoscenza delle informazioni riguardanti la specifica regione verso cui si stanno muovendo, eseguono una

```
rd(msg(M) @ circleRegion(here, rad))
```

Oltre alla posizione delle persone ferite, la sala di controllo tiene traccia anche delle posizioni dei soccorritori. A questo scopo, gli agenti soccorritori rilasciano una tupla spaziale del tipo

out(rescuer(RescuerID, Status) @ me)

in cui **Status** descrive cosa stanno facendo – ad esempio, può assumere valori come **exploring**, **triaging**, **helpNeeded**.

Questa sarà aggiornata dagli agenti soccorritori sul campo quando il loro stato cambia, ad esempio

in(rescuer(1, Status) @ me) out(rescuer(1, triaging) @ me)

Anche queste informazioni sono raccolte e mostrate da un agente in sala di controllo. Inoltre, potrebbe capitare che un soccorritore abbia bisogno di ottenere le indicazioni riguardo a dov’è localizzato un altro soccorritore in quel momento, per potersi muovere verso di esso – ad esempio, per aiutarlo. A questo scopo, l’agente soccorritore potrebbe invocare una

rd(rescuer(rescuerToFindId, Status) @ missionArea)

Una volta recuperata la tupla, il soccorritore che ha invocato l’operazione viene a conoscenza della posizione in cui si trova l’agente **rescuerToFindId**, in modo tale da calcolare un percorso per raggiungerlo.

3.4.1 Setup su Unity3D

Il caso di studio appena descritto viene ora simulato all’interno di una scena Unity3D⁶ – Figura 3.9 –, in cui il MAS è organizzato come segue. Sono presenti quattro tipi di agenti:

1. **RescuerAgent**: agente soccorritore, situato sul campo. Inizialmente in attesa di una segnalazione di emergenza, quando ciò avviene si muove verso di essa ed effettua il triaging delle persone presenti nella zona segnalata.
2. **ControlRoomAgent**: agente in sala di controllo, non situato sul campo. Genera le segnalazioni di emergenza, per poi monitorare il triaging nella zona segnalata.
3. **SidekickAgent**: agente assistente al soccorso, situato sul campo. Afianca un RescuerAgent e lo raggiunge nella zona di emergenza in cui quest’ultimo ha effettuato il triaging.

⁶Link al video della simulazione: <https://youtu.be/HE7V4dakk1c>



Figura 3.9: La scena Unity3D del caso di studio Rescue.

4. **GodAgent**: agente preposto alla generazione delle situazioni di emergenza pseudocasuali, non situato. Sarà poi compito di ControlRoomAgent segnalarle sul campo tramite il rilascio di tuple spaziali.

Tutte le azioni che avvengono all'interno del MAS partono dalla generazione delle emergenze, si parte quindi osservando l'agente più semplice dei quattro, GodAgent:

```

1 belief counter(1).
2
3 desire randomize.
4
5 add randomize && true => [
6   belief counter(C),
7   act (getcontrolroomagentref(C), Ref),
8   act (getrandomenvironmentevent, Ev),
9   act (getrandomenvironmentlocation, Loc),
10  add_agent_desire(Ref, manage_warning(Ev, Loc)),
11  add_desire(redo)
12].
13
14 add redo && true => [
15   del_belief(counter(C)),
16   Cx is C + 1,
17   Cx < 5,
18   add_belief(counter(Cx)),
19   cr waitforseconds(3.0),
20   add_desire(randomize)
21].

```

GodAgent aggiunge un desiderio `manage_warning` a ciascun agente ControlRoomAgent, contenente informazioni riguardo ad una nuova emergenza (tra cui la posizione) scelta in maniera casuale da una lista preinizializzata all'interno dello script C#. In questo setup genera quattro emergenze, ognuna a distanza di tre secondi dall'altra.

Un agente ControlRoomAgent diventa operativo non appena GodAgent aggiunge ad esso un nuovo desiderio `manage_warning`:

```

1 add manage_warning(Ev, Loc) && true => [
2     MyName is $me.name,
3     act showtoast(MyName, ": nuovo problema rilevato, ", Ev, " in posizione
4         ", Loc),
5     out(warning(Ev) at region(Loc, circle(50))),
6     (
7         atom(Loc),
8         add_desire(control_warning(Loc))
9         ;
10        Loc = coord(Lat, Long, _),
11        add_desire(control_warning(coord(Lat, Long)))
12    )
13].
14 add control_warning(Loc) && true => [
15     MyName is $me.name,
16     in(triage(_, _) at region(Loc, circle(50))),
17     act showtoast(MyName, ": persona soccorsa in zona ", Loc),
18     add_desire(redo(Loc))
19].
20 add redo(Loc) && true => [
21     add_desire(control_warning(Loc))
22].
23]
```

Quando ciò avviene, rilascia una tupla spaziale attraverso

out(warning(Ev) at region(Loc, circle(50)))

creando una regione circolare estesa per 50 metri, centrata in Loc, e si mette in attesa di ricevere informazioni riguardo al triaging che avviene in quella regione tramite ripetute invocazioni di

in(triage(_, _) at region(Loc, circle(50)))

Per quel che riguarda l'agente di tipo RescuerAgent, il suo comportamento è definito all'interno del seguente file Prolog:

```

1 desire wait_for_emergency.
2
3 add wait_for_emergency && true => [
4     ID is $me.name,
5     out(rescuer(ID, waiting_emergency) at me),
6     in(warning(_) at region(here, circle(600))),
7     cr gotoemergencyregion,
8     add_desire(rescue)
9 ]..
10
11 add rescue && true => [
12     ID is $me.name,
13     act (findpersoninregion, Per),
14     Per \= null,
15     in(rescuer(ID, _) at me),
16     out(rescuer(ID, triaging) at me),
17     cr gotopersoninregion(Per),
18     out(triage(rescuer(ID), status('critical')) at Per),
19     add_desire(rescue_again)
20 ]..
21
22 add rescue_again && true => [
23     ID is $me.name,
24     act (findpersoninregion, Per),
25     Per \= null,
26     in(rescuer(ID, _) at me),
27     out(rescuer(ID, triaging) at me),
28     cr gotopersoninregion(Per),
29     out(triage(rescuer(ID), status('critical')) at Per),
30     add_desire(redo)
31 ]..
32
33 add redo && true => [
34     add_desire(rescue_again)
35 ]..
36
37 del rescue && true => [
38     ID is $me.name,
39     act showtoast(ID, ": Nessuno da soccorrere in questa regione"),
40     act (findsidekick, Sidekick),
41     add_agent_desire(Sidekick, follow(ID)),
42     in(rescuer(ID, _) at me),
43     add_desire(wait_for_emergency)
44 ]..
45
46 del rescue_again && true => [
47     ID is $me.name,
48     act showtoast(ID, ": Nessuno rimasto da soccorrere in questa regione"),
49     act (findsidekick, Sidekick),
50     add_agent_desire(Sidekick, follow(ID)),
51     in(rescuer(ID, _) at me),
52     in(sidekickArrived(Sidekick) at here),
53     add_desire(wait_for_emergency)
54 ]..

```

Il suo stato iniziale prevede l'attesa di un'emergenza in un raggio di 600 metri partendo dalla sua posizione attuale tramite

in(warning(_) at region(here, circle(600)))

e una volta rilevata l'emergenza, si muove verso di essa tramite la chiamata alla routine gotoemergencyregion. Di seguito viene riportato un estratto

di questa coroutine, di particolare interesse:

```

1  public IEnumerator GoToEmergencyRegion()
2  {
3      ...
4
5      IList<LatLong> coordinatesList = new List<LatLong>();
6      //Obtain all the directions to arrive to the emergency region
7      ...
8
9      Linda4SpatialTuples.SendMessageInCurrentRegion(this, new
10         Structure("square", -1), new Structure("wasHere", this.name,
11             counter));
12     //Move towards each direction's coordinates
13     foreach (LatLong coordinate in coordinatesList)
14     {
15         this.counter++;
16         ...
17         Linda4SpatialTuples.SendMessageInCurrentRegion(this, new
18             Structure("square", -1), new Structure("wasHere", this.name,
19                 this.counter));
20     }
21     ...
22 }
```

La coroutine in questione calcola il percorso da seguire per arrivare alla regione in cui si è verificata l'emergenza e ad ogni cambio di direzione rilascia una tupla spaziale con l'invocazione

out(wasHere(Name, Counter) @ here)

tramite la chiamata al metodo `SendMessageInCurrentRegion` della classe `Linda4SpatialTuples`. Il rilascio della tupla avviene in questo caso lato C# e non lato Prolog come sempre fatto fino ad ora per una semplice ragione che non può essere ignorata: durante l'esecuzione di una coroutine, l'agente non può eseguire nessuna azione lato Prolog, fintanto che la coroutine stessa non è terminata. Dal momento che tutto il movimento di un RescuerAgent fino alla regione dell'emergenza avviene all'interno di una coroutine, si è costretti ad effettuare ogni azione intermedia all'interno della coroutine stessa, azione rappresentata in questo caso dal rilascio di una tupla spaziale che verrà sfruttata da SidekickAgent per raggiungere RescuerAgent nella regione dell'emergenza. Una volta raggiunta la zona dell'emergenza, l'agente inizia a dirigersi verso ciascuna persona coinvolta per effettuare il triaging (piano `add rescue && true`). Se nella zona non è presente nessuna persona, il piano fallisce e viene

innescata la goal deletion (piano `del rescue && true`), in cui si aggiunge un desiderio a SidekickAgent affinché questo si sposti verso la regione dell'emergenza. RescuerAgent si mette poi subito in attesa di una nuova emergenza, senza aspettare SidekickAgent. Se nella zona invece sono presenti persone, RescuerAgent le raggiunge una alla volta ed appone su di esse una tupla spaziale tramite l'operazione

```
out(triage(rescuer(ID), status('critical')) at Per)
```

dove `ID` è il nome identificativo dell'agente RescuerAgent e `Per` è il nome della persona a cui si sta apponendo la tupla spaziale. Una volta terminato il triaging di tutte le persone in regione (piano `del rescue_again && true`), RescuerAgent richiama a sè SidekickAgent, aspettando il suo arrivo in zona tramite l'invocazione

```
in(sidekickArrived(Sidekick) at here)
```

prima di mettersi in attesa di un'altra emergenza.

Il comportamento di un SidekickAgent invece è definito dal seguente file Prolog:

```

1 belief breadcrumb(1).
2
3 add follow(IDRescuer) && true => [
4   (
5     belief rescuerID(IDRescuer)
6     ;
7     add_belief(rescuerID(IDRescuer))
8   ),
9   belief breadcrumb(C),
10  rd(wasHere(IDRescuer, C) at region(here, circle(300))),
11  Cx is C + 1,
12  del_belief(breadcrumb(C)),
13  add_belief(breadcrumb(Cx)),
14  add_desire(gotobreadcrumb)
15].
16
17 add gotobreadcrumb && true => [
18   cr gotobreadcrumb,
19   act (amiinregion, Flag),
20   Flag = false,
21   belief rescuerID(IDRescuer),
22   add_desire(follow(IDRescuer))
23].
24
25 del gotobreadcrumb && true => [
26   Me is $me,
27   out(sidekickArrived(Me) at here)
28].
```

Come si può notare non è presente nessun desire iniziale. SidekickAgent diventa operativo solo quando RescuerAgent aggiunge ad esso il desire `follow(IDRescuer)`: a questo punto, l'assistente cerca una tupla spaziale tramite un'operazione

```
rd(wasHere(IDRescuer, C) at region(here, circle(300)))
```

per poter iniziare a seguire la traccia di tuple spaziali lasciate sul cammino da **IDRescuer**. Una volta raggiunto **IDRescuer**, rilascerà una tupla spaziale con l'operazione

```
out(sidekickArrived(Me) at here)
```

dove **Me** è il riferimento al GameObject di SidekickAgent. Questa tupla spaziale sarà poi recuperata da RescuerAgent, in attesa su di essa.

In questa simulazione tutti i pattern di coordinazione descritti nella Sezione 3.3 sono messi all'opera:

- Il pattern *breadcrumbs* è utilizzato per consentire a SidekickAgent di seguire il percorso formato dalle tuple rilasciate sul proprio cammino da parte di un RescuerAgent.
- Il pattern *awareness* è fondamentale per i ControlRoomAgent per monitorare lo stato dei RescuerAgent e delle persone ferite.
- La *sincronizzazione spaziale* viene utilizzata dai RescuerAgent per attendere l'arrivo del proprio SidekickAgent nella zona di un'emergenza prima che si dirigano verso un'altra regione.
- La *mutua esclusione* abilita una equa distribuzione dei RescuerAgent sul campo, in modo tale da evitare sovrapposizioni su una stessa regione interessata da un'emergenza.

Capitolo 4

Analisi delle performance

Quando si tratta di videogiochi o simulazioni all'interno di un game engine, c'è un aspetto che non va mai trascurato, quello delle performance. L'approccio utilizzato per l'ingegnerizzazione degli agenti BDI è molto potente ed aiuta a rendere più facile il design e la coordinazione di agenti autonomi quando viene sviluppato un MAS, tuttavia, in un contesto come quello di un videogioco, in cui ogni frame e ogni millisecondo ha importanza, si rende necessaria un'analisi delle performance quanto più dettagliata possibile. L'obiettivo del seguente Capitolo è quello di servirsi degli strumenti messi a disposizione da Unity3D per iniziare a porre una lente d'ingrandimento su questo aspetto cruciale.

4.1 Il Profiler

Lo strumento Profiler, integrato in Unity, permette di catturare dati in tempo reale sulle scene in esecuzione e di analizzarne le relative performance. Segnala quanto tempo viene speso nelle varie aree del proprio gioco o simulazione. Per esempio, può segnalare la percentuale di tempo trascorso per il rendering, per le animazioni o nella logica di gioco. È possibile analizzare le performance di GPU, CPU, memoria, rendering e audio. Per accedere ai dati del Profiler, è sufficiente eseguire la propria scena nell'editor e abilitare la finestra Profiler dal menu Window. La nuova finestra mostrerà i dati in una linea temporale, in modo tale da permettere di analizzare i frame o le aree in cui sono presenti dei picchi, ovvero che impiegano più tempo degli altri. Cliccando su una qualunque area della linea temporale, la sezione inferiore del Profiler mostra informazioni più dettagliate per il frame selezionato, come si può osservare in



Figura 4.1: Il Profiler in esecuzione su Unity3D.

Figura 4.1. Per usare il Profiler su un gioco o un player Unity in esecuzione su un altro dispositivo, è possibile connettere l'editor Unity a quest'ultimo. Il menu a tendina Connected Player mostra tutte le istanze di Unity player in esecuzione sulla rete locale, ognuna delle quali è identificata dal tipo e dal nome dell'host su cui è in esecuzione l'istanza. Per connettersi a un player Unity, è necessario lanciare la relativa istanza come Development build (menu: File > Build Settings). Con l'opzione Autoconnect Profiler è possibile connettere l'Editor e il Player in automatico al lancio del gioco.

4.1.1 Il Profiler nel Caso di Studio Rescue

Il Profiler appena introdotto è stato utilizzato per monitorare la scena del caso di studio descritto in Sezione 3.4 durante la sua intera esecuzione, prima in ambiente desktop, su un PC con le seguenti specifiche hardware:

- CPU: Intel Core i5-6600 (4 core, 4 thread) @ 3.30GHz
- GPU: Nvidia GeForce GTX 970
- RAM: 16GB DDR4 @ 2400MHz
- SSD: Samsung 840 Evo 250GB

e successivamente in ambiente mobile su smartphone Android Samsung Galaxy S5, con le seguenti specifiche hardware:

- CPU: Qualcomm Krait 400 (4 core, 4 thread) @ 2.50 GHz
- GPU: Qualcomm Adreno 330
- RAM: 2GB DDR3 @ 993MHz

Le prove e i monitoraggi sono stati effettuati modificando di volta in volta il numero di agenti presenti nel MAS, per verificare se questa variabile influisce in maniera diretta sulle performance della scena Unity, e di seguito vengono mostrati alcuni grafici estratti direttamente dal Profiler mentre la scena è in esecuzione.

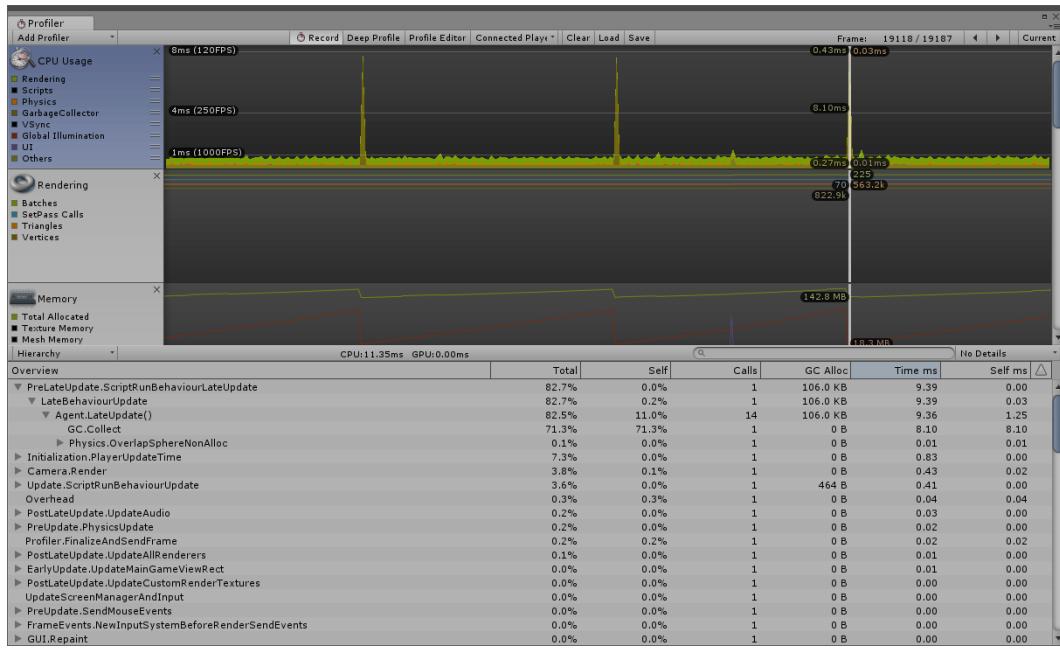


Figura 4.2: Profiler durante l'esecuzione del caso di studio Rescue con 14 agenti presenti nel MAS, su PC.

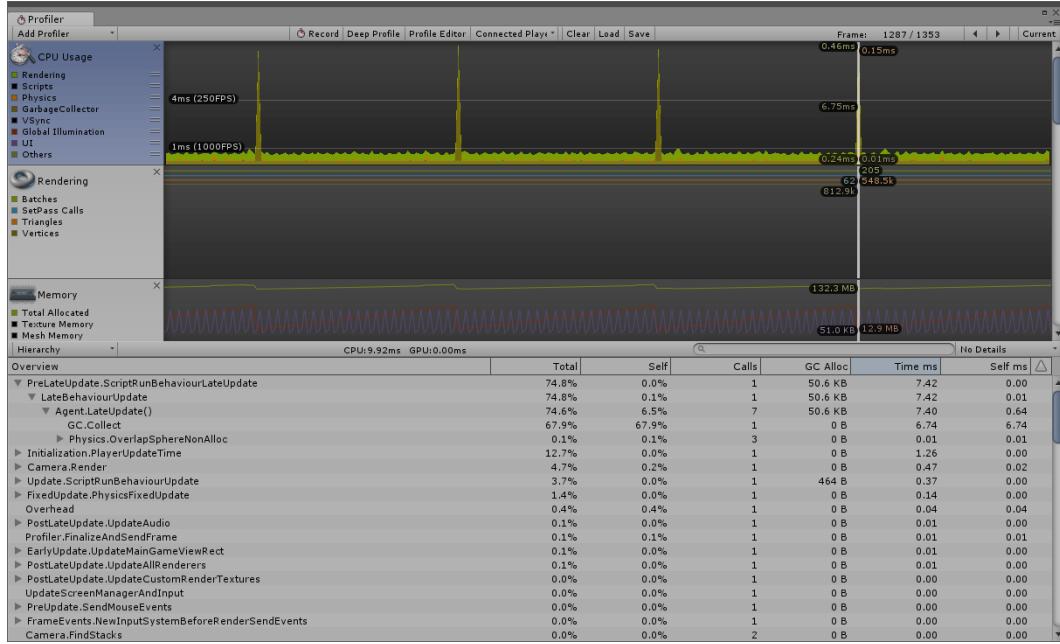


Figura 4.3: Profiler durante l'esecuzione del caso di studio Rescue con 7 agenti presenti nel MAS, su PC.

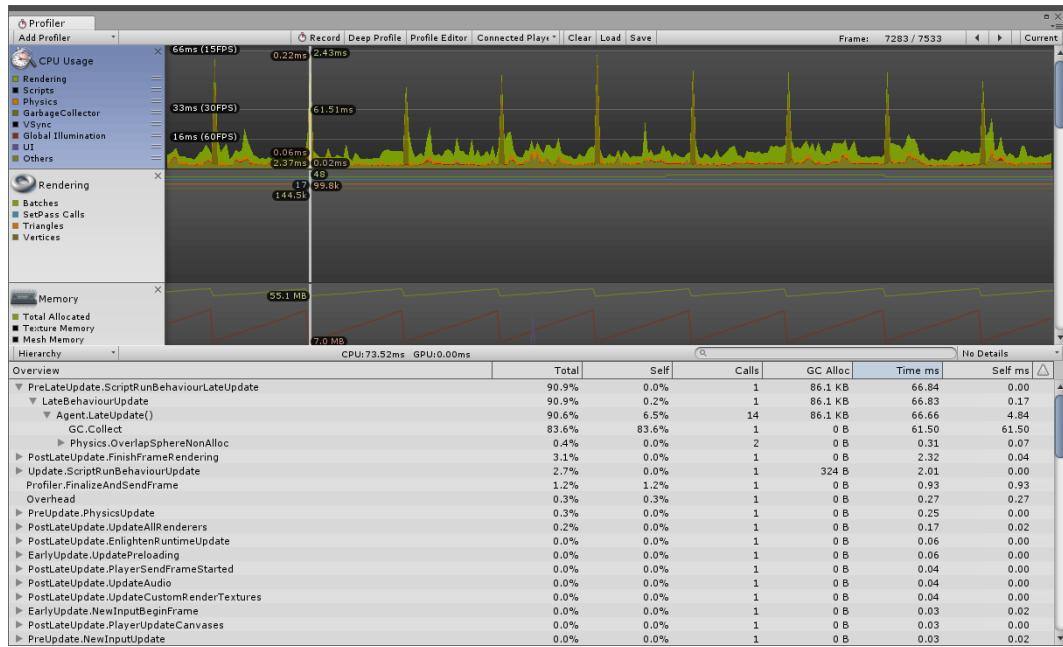


Figura 4.4: Profiler durante l'esecuzione del caso di studio Rescue con 14 agenti presenti nel MAS, su Android.

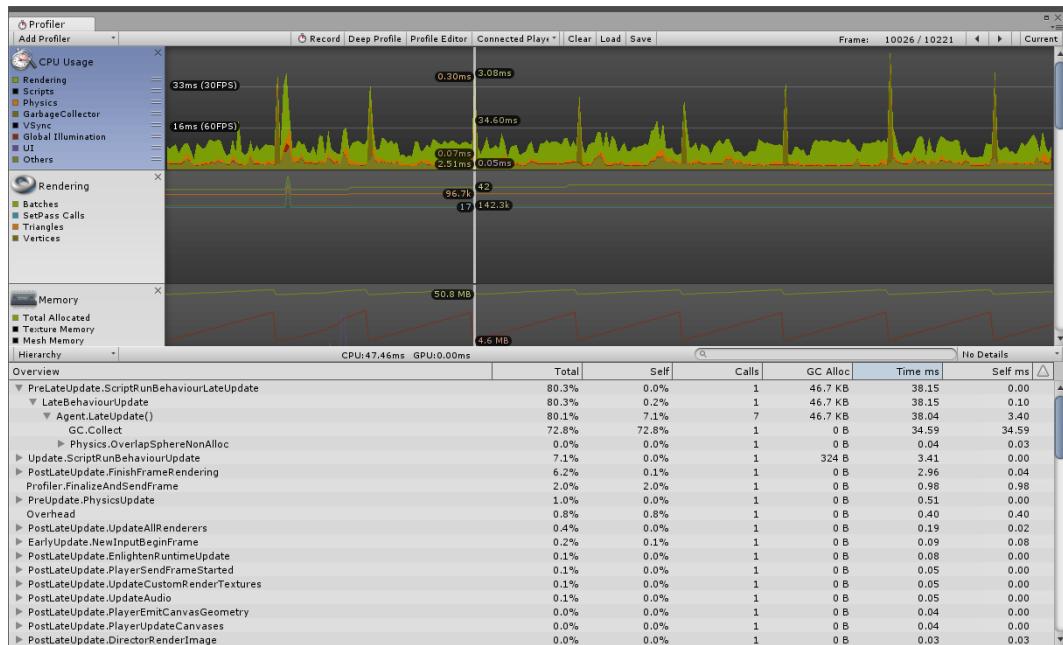


Figura 4.5: Profiler durante l'esecuzione del caso di studio Rescue con 7 agenti presenti nel MAS, su Android.

Come si può facilmente notare con una primo sguardo, tutte le immagini proposte hanno in comune una caratteristica: la presenza di picchi di utilizzo di CPU a intervalli di tempo regolari. Essi rappresentano i frame più impegnativi da renderizzare, e selezionandone uno il Profiler offre più dettagli in merito. Esso fa notare infatti che la maggior parte del tempo necessaria a renderizzare questi frame è utilizzata dal garbage collector con la chiamata di sistema automatica `GC.Collect`, la quale avviene all'interno del metodo `LateUpdate` di ciascun istanza di `Agent`, utilizzato per il corretto funzionamento del reasoner BDI scritto in Prolog. Le principali cause di questi picchi sono quindi da ricercare all'interno del reasoner e alla distribuzione di Prolog utilizzata per la sua progettazione – UnityProlog –, i quali si sono già dimostrati essere più che sufficienti per supportare MAS poco popolati ma che con un aumento considerevole degli agenti al suo interno, possono creare un crollo delle performance sempre più accentuato, nella fattispecie su mobile, dove le risorse computazionali sono più limitate rispetto all'ambiente desktop. Confrontando le Figure 4.2 e 4.3 con le controparti 4.4 e 4.5 si può osservare infatti come il frame rate medio sia decisamente diverso tra le due piattaforme:

- Desktop: circa 800 frame per secondo, che si traducono in un'esperienza fluida senza alcuna presenza di stuttering, anche in concomitanza con le chiamate al garbage collector.
- Mobile: circa 50 frame per secondo – tetto massimo di 60 –, che si traducono in un'esperienza mediamente fluida, con qualche lieve presenza sporadica di stuttering in concomitanza con le chiamate al garbage collector.

Il modello proposto fin qua e le fondamenta su cui si basa si confermano quindi pienamente utilizzabili a scopi sperimentali, ma sono necessarie ulteriori investigazioni a fronte di MAS particolarmente ampi, considerando dapprima un numero di agenti nell'ordine delle centinaia, poi delle migliaia e così via.

Capitolo 5

Conclusioni e Sviluppi Futuri

Il prototipo sviluppato rappresenta un’implementazione preliminare di concetti che fino ad oggi erano stati espressi solo dal punto di vista teorico e della letteratura. L’obiettivo primario di migliorare lo stato dell’arte dell’integrazione tra game engines e MAS è stato raggiunto, mettendo a disposizione del programmatore un primo set di primitive e costrutti che abilitano la coordinazione mediata dall’ambiente prevista da Spatial Tuples sfruttando le potenzialità offerte da Unity3D confermando ancora una volta come le due tecnologie – quella dei game engine e quella dei MAS – si sposino bene insieme. Rimangono tuttavia ampi margini di miglioramento e ulteriori studi da compiere.

Un primo passo da effettuare per rendere questo prototipo ancora più solido e completo, è rappresentato dall’implementazione delle estensioni delle primitive base, ossia le versioni predicative `inp` e `rdp` – che differenziano dalle rispettive versioni base per essere *non* sospensive – e le versioni *bulk* `in_all` e `rd_all`, che consentono di far ritornare più di una tupla alla volta.

Un altro aspetto da non trascurare e menzionato nel Capitolo 4 è identificato dal motore Prolog utilizzato. Prima di utilizzare UnityProlog nello sviluppo delle librerie per gli agenti BDI e Linda, è stato testato `tuProlog` [20], una versione di Prolog estremamente leggera e compilata usando il framework .NET – contrariamente a UnityProlog, che si ricorda essere una versione interpretata. Un problema di incompatibilità noto con la versione disponibile a quel tempo di Unity3D (2017.2)⁷ ne ha tuttavia impedito il funzionamento ed è stato scartato in favore di UnityProlog. Oggi, con l’ultima versione 2018.1 di Unity3D tale problema risulta essere risolto, di conseguenza sarebbe interessante

⁷ <https://issuetracker.unity3d.com/issues/unity-fails-to-load-net-4-dot-6-assemblies-with-typeloadexception>

migrare e riadattare tutto il codice Prolog finora sviluppato con UnityProlog, su tuProlog, analizzandone i benefici in termini di performance – compito non banale come sembra, dal momento che richiederebbe la riscrittura di buona parte del reasoner degli agenti BDI.

Spatial Tuples integra la potenza di una comunicazione generativa all'interno di un framework in cui lo spazio è modellato come un'entità di prima classe, abilitando così diverse forme di coordinazione spaziale. Tutto ciò è stato possibile apprezzarlo con successo grazie alle implementazioni dei pattern nella Sezione 3.3 e al loro utilizzo sinergico nella Sezione 3.4. Queste forme di coordinazione spaziale appena menzionate strizzano l'occhio a domini applicativi che spaziano dal pervasive computing alla realtà aumentata, ed è proprio su quest'ultima che sarebbe interessante iniziare a investigare: Unity3D infatti offre di default il supporto a varie piattaforme e framework di realtà aumentata quali Apple ARKit, Google ARCore e Vuforia che aiuterebbero a rendere ancora più concreto e tangibile ciò che è stato sviluppato fino ad oggi.

Un altro interessante spunto di riflessione sui possibili lavori futuri riguarda la distribuzione: consentire quindi agli agenti BDI e agli spazi di tuple di essere eseguiti su dispositivi diversi, permettendo ai primi di sincronizzarsi tramite Spatial Tuples. A tal proposito, Unity3D offre nativamente il supporto al multiplayer che potrebbe essere sfruttato per indagare più a fondo in questa direzione. Inoltre, l'utilizzo del modulo GPS – oggigiorno integrato su tutti i moderni smartphone e dispositivi smart indossabili – può rivelarsi utile per mantenere traccia della posizione di ciascun agente in esecuzione sui diversi dispositivi sparsi per l'ambiente, requisito che si ribadisce essere di importanza cruciale per un modello di coordinazione basato sulla spazio.

Ringraziamenti

Questo lavoro rappresenta il felice epilogo di un percorso accademico durato più di 5 anni, il quale mi ha permesso di formarmi dal punto di vista professionale e di crescere come persona. Giunto fin qui, è quindi il momento di tirare le somme e ringraziare chi nel corso di tutti questi anni mi è stato affianco, soprattutto nei momenti più faticosi e stressanti.

Il primo ringraziamento va al relatore di questa tesi, il Prof. Andrea Omicini. Non è una semplice formalità, a lui sono riconoscente per avermi offerto la possibilità di lavorare a un progetto stimolante e coinvolgente. Lo stesso ringraziamento ovviamente va anche al correlatore, il Dott. Stefano Mariani, presenza fondamentale che mi ha seguito in prima persona con molta disponibilità e professionalità durante l'intero svolgimento del lavoro.

Un grazie a tutti coloro che mi sono stati vicini in università, in particolare i miei compagni di corso Luca e Filippo con i quali ho condiviso tanti momenti sin dai tempi della triennale, dai meno impegnativi come un semplice pranzo a quelli più seri e delicati come la preparazione di esami o lo sviluppo dei tanti progetti svolti insieme. Senza di loro probabilmente non avrei imparato il significato di lavoro di squadra. Le nostre strade ora si dividono, ma ricorderò questi anni passati insieme sempre con immenso piacere.

Ringrazio chiunque mi sia stato vicino in qualunque modo, anche all'infuori del puro ambiente universitario, pur solo con un messaggio di conforto o per sapere semplicemente come stavo.

Infine, una menzione speciale va ai miei genitori, per quanto mi sono stati vicini, per tutto l'aiuto dato, per avermi sempre fornito i consigli migliori e per avermi sostenuto, sia dal lato meramente economico che quello umano. Questo percorso non sarebbe stato possibile senza la loro costante presenza, non posso fare altro che ringraziarli dal profondo del mio cuore.

Alessandro

Bibliografia

- [1] A. Omicini, A. Ricci, and M. Viroli, “Artifacts in the A&A meta-model for multi-agent systems,” *Autonomous Agents and Multi-Agent Systems*, vol. 17, pp. 432–456, Dec. 2008. Special Issue on Foundations, Advanced Topics and Industrial Perspectives of Multi-Agent Systems.
- [2] S. Mariani and A. Omicini, “Game engines to model MAS: A research roadmap,” in *WOA 2016 – 17th Workshop “From Objects to Agents”* (C. Santoro, F. Messina, and M. De Benedetti, eds.), vol. 1664 of *CEUR Workshop Proceedings*, pp. 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).
- [3] A. Ricci, M. Viroli, A. Omicini, S. Mariani, A. Croatti, and D. Pianini, “Spatial tuples: Augmenting reality with tuples,” *Expert Systems*, Apr. 2018. Special Issues IDC 2016.
- [4] Unity Technologies, “Unity3D.” URL:<https://unity3d.com/>.
- [5] N. Poli, “Game Engines and MAS: BDI & Artifacts in Unity,” Master’s thesis, Alma Mater Studiorum Università di Bologna, 2018.
- [6] M. Cerbara, “Game engines and MAS: tuplespace-based interaction in Unity3D,” Master’s thesis, Alma Mater Studiorum Università di Bologna, 2018. URL:<http://amslaurea.unibo.it/15627/>.
- [7] A. Omicini and F. Zambonelli, “MAS as complex systems: A view on the role of declarative approaches,” in *Declarative Agent Languages and Technologies* (J. A. Leite, A. Omicini, L. Sterling, and P. Torroni, eds.),

- vol. 2990 of *Lecture Notes in Computer Science*, pp. 1–17, Springer Berlin Heidelberg, May 2004. 1st International Workshop (DALT 2003), Melbourne, Australia, 15 July 2003. Revised Selected and Invited Papers.
- [8] F. Zambonelli and A. Omicini, “Challenges and research directions in agent-oriented software engineering,” *Autonomous Agents and Multi-Agent Systems*, vol. 9, pp. 253–283, Nov. 2004. Special Issue: Challenges for Agent-Based Computing.
 - [9] D. Weyns, A. Omicini, and J. J. Odell, “Environment as a first class abstraction in multi-agent systems,” *Autonomous Agents and Multi-Agent Systems*, vol. 14, pp. 5–30, Feb. 2007. Special Issue on Environments for Multi-agent Systems.
 - [10] P. Ciancarini, A. Omicini, and F. Zambonelli, “Multiagent system engineering: The coordination viewpoint,” in *Intelligent Agents VI. Agent Theories, Architectures, and Languages* (N. R. Jennings and Y. Le spérance, eds.), vol. 1757 of *LNAI*, pp. 250–259, Springer, 2000. 6th International Workshop (ATAL’99), Orlando, FL, USA, 15–17 July 1999. Proceedings.
 - [11] A. S. Rao and M. P. Georgeff, “BDI agents: From theory to practice,” in *1st International Conference on Multi Agent Systems (ICMAS 1995)* (V. R. Lesser and L. Gasser, eds.), (San Francisco, CA, USA), pp. 312–319, The MIT Press, 12-14 June 1995.
 - [12] D. Gelernter, “Generative communication in linda,” *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 80–112, Jan. 1985.
 - [13] A. Ricci, A. Omicini, M. Viroli, L. Gardelli, and E. Oliva, “Cognitive stigmergy: Towards a framework based on agents and artifacts,” in *Environments for MultiAgent Systems III* (D. Weyns, H. V. D. Parunak, and F. Michel, eds.), vol. 4389 of *Lecture Notes in Computer Science*, ch. 7, pp. 124–140, Springer Berlin Heidelberg, May 2007. 3rd International Workshop (E4MAS 2006), Hakodate, Japan, 8 May 2006. Selected Revised and Invited Papers.
 - [14] I. Horswill, “UnityProlog,” 2015. URL:<https://github.com/ianhorswill/UnityProlog>.

- [15] R. H. Bordini, H. J. Fred., and M. J. Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason*. Wiley, 2007.
- [16] A. Omicini and F. Zambonelli, “Coordination of mobile agents for information systems: the TuCSoN model,” in *6th Convention of the Italian Association for Artificial Intelligence (AI*IA'98)* (S. Badaloni and C. Minnaja, eds.), (Padova, Italy), pp. 94–98, Edizioni Progetto Padova, 23–25 Sept. 1998. AI*IA'98 Workshop on Knowledge Integration.
- [17] E. Denti and A. Omicini, “Engineering multi-agent systems in LuCe,” in *ICLP'99 International Workshop on Multi-Agent Systems in Logic Programming (MAS'99)* (S. Rochefort, F. Sadri, and F. Toni, eds.), (Las Cruces, NM, USA), 30 Nov. 1999.
- [18] WRLD Team, “WRLD,” 2015. URL:<https://www.wrld3d.com/>.
- [19] P. Brunetti, A. Croatti, A. Ricci, and M. Viroli, “Smart augmented fields for emergency operations,” *Procedia Computer Science*, vol. 63, pp. 392 – 399, 2015. The 6th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2015)/ The 5th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2015)/ Affiliated Workshops.
- [20] E. Denti, A. Omicini, and A. Ricci, “tuProlog: A light-weight Prolog for Internet applications and infrastructures,” in *Practical Aspects of Declarative Languages* (I. Ramakrishnan, ed.), vol. 1990 of *Lecture Notes in Computer Science*, pp. 184–198, Springer Berlin Heidelberg, 2001. 3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 Mar. 2001. Proceedings.

Elenco delle figure

2.1	Spatial Tuples come layer virtuale che aumenta la realtà fisica [3].	13
2.2	Agenti che inseriscono e recuperano tuple. A sinistra una out posiziona la tupla spaziale in una posizione specifica; una in recupera una tupla specificando una regione. A destra una out posiziona la tupla spaziale in una regione più grande; una rd osserva una tupla dalla posizione corrente [3].	14
3.1	GameObject padre con componente GeographicTransform	22
3.2	GameObject figlio a cui vengono applicati tutti i componenti Unity standard necessari	22
3.3	Regione contenente una tupla rilasciata su me che si muove insieme ad un agente.	33
3.4	Il pattern Awareness, or: <i>Find me!</i> su Unity3D.	49
3.5	Il pattern Breadcrumbs, or: <i>Follow me!</i> come esempio di situated knowledge sharing [3].	50
3.6	Il pattern Breadcrumbs, or: <i>Follow me!</i> su Unity3D.	52
3.7	Il pattern <i>sincronizzazione spaziale</i> su Unity3D.	54
3.8	Il pattern <i>mutua esclusione</i> su Unity3D.	56
3.9	La scena Unity3D del caso di studio Rescue.	60
4.1	Il Profiler in esecuzione su Unity3D.	68
4.2	Profiler durante l'esecuzione del caso di studio Rescue con 14 agenti presenti nel MAS, su PC.	70
4.3	Profiler durante l'esecuzione del caso di studio Rescue con 7 agenti presenti nel MAS, su PC.	70
4.4	Profiler durante l'esecuzione del caso di studio Rescue con 14 agenti presenti nel MAS, su Android.	71

