ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea in Ingegneria Elettronica, Informatica e delle Telecomunicazioni

Spatial Tuples nel mondo reale: il caso di Unity e Google Maps

Tesi di laurea in
Sistemi Autonomi

Relatore
Prof. Andrea Omicini

Correlatore
Dott. Giovanni Ciatto

Candidato
Marco Pastore

III Sessione di Laurea
Anno Accademico 2019-2020

Sommario

Spatial Tuples è un modello di coordinazione tuple-based fortemente basato sul concetto di posizione fisica nel mondo e sul movimento di un componente situato all'interno dello spazio reale.

Il mondo dei videogiochi si muove con sempre più convinzione verso situazioni ludiche in cui i concetti di spazio e movimento si pongono come pilastri su cui strutturare il game design e la filosofia di gioco.

Questa tesi si pone come obiettivo quello di riprodurre il più fedelmente possibile il modello *Spatial Tuples* all'interno del Game Engine *Unity*, utilizzando le astrazioni e i meccanismi offerti dal motore di gioco.

Si ritiene, infatti, che i due mondi possano offrire grandi vantaggi, l'uno all'altro. Portando Spatial Tuples in Unity, si possono ottenere possibilità di interazioni potenti ed espressive fra oggetti. Portando Unity in Spatial Tuples, si offre un supporto tecnologico costantemente sviluppato e all'avanguardia, spinto dagli investimenti di un'industria in forte salute e crescita.

A Lui *e a* Lei,
che mi hanno sempre sostenuto nonostante io non sia facile,
non facendomelo mai pesare,
standomi sempre accanto e
volendomi sempre bene.
Spero che questo vi regali grande gioia

A Te,
che sarai sempre parte di me,
costante nella mia mente
e nel mio cuore.
Questo traguardo è anche merito tuo

A Voi,
che mi state vicini e mi volete bene anche se sono un disastro,
che mi date forza nel brutto e nel bello.
Continueremo a ridere di questo mio percorso

A Me,
che nonostante tutto, sono qua e vado avanti
che inciampo e cado, ma mi rialzo e proseguo.
Non dimenticare mai,
sii fiero di quello che sei e di quello che fai

***A* Baldo,**
che avrebbe festeggiato più di tutti.
Guardami dall'alto,
sorridimi e abbracciami

MIFILI

Indice

Elenco delle figure

Elenco dei listati

Capitolo 1

Introduzione

Il continuo avanzamento e progresso tecnologico portano i sistemi software a dover soddisfare esigenze sempre più grandi, complesse e strutturate. Ne deriva una continua introduzione di nuovi strumenti e di nuovi paradigmi, atti a cercare di risolvere o semplificare tali difficoltà.

La **programmazione agent-oriented** [11] si presta efficacemente ad affrontare la crescente complessità e articolazione dei moderni sistemi software e degli scenari applicativi odierni.

La visione tecnologica attuale vede un numero costantemente maggiore di oggetti situati in un ambiente, connessi e interagenti fra di loro. Uno scenario che si presta molto bene ad essere rappresentato attraverso **sistemi multi-agente**. Ne consegue che, all'aumentare degli agenti che interagiscono fra di loro e con l'ambiente circostante, è necessario un modello di coordinazione valido, espressivo ed efficiente, tale da semplificare la modellazione, la struttura e l'efficacia del sistema multi-agente.

Il modello di coordinazione denominato **Spatial Tuples**[16] è stato concepito proprio tenendo a mente la crescente diffusione di tecnologie mobili e indossabili, unito ad una maggiore pervasività dell'accesso a Internet. Questi fattori portano a concepire applicazioni distribuite e consapevoli del contesto e della posizione, modellando comportamenti e obiettivi del sistema software in base alla locazione dell'utente nello spazio fisico.

Altri scenari in cui ricopre fondamentale importanza la posizione dell'utente sono quelli relativi al campo della **mixed/augmented reality**, altro fenomeno che sta ottenendo una continua crescita in termini di sviluppo e di risonanza. Il mondo reale viene arricchito con dati e informazioni digitali. Pertanto, una determinata posizione del mondo, avente quindi le proprie coordinate geografiche, viene decorata da strutture proprie del mondo digitale.

Mixed/augmented reality trova terreno fertile anche nel contesto dei videogiochi. Nel corso degli anni si stanno introducendo sempre più giochi che basano la

loro struttura e la loro filosofia di intrattenimento sulla posizione del giocatore. Ciò comporta un notevole aumento della immersione e del coinvolgimento dell'utente, nonché nuove strutture di game design. Inoltre, le coordinate geografiche dell'utilizzatore non solo assumono carattere fondante sul funzionamento del gioco, ma permettono di aggiungere informazioni digitali nella esatta posizione nella quale si trova il giocatore. Se a ciò si aggiunge anche la possibilità di visualizzare l'ambiente circostante mediante fotocamera dello smartphone o lenti di occhiali e visori predisposti alla realtà aumentata o mixed, il mondo digitale del videogioco assume ulteriore incisività e immersività.

Il modo più efficace per produrre un videogioco è attraverso ciò che viene chiamato **Game Engine**. Tali framework di sviluppo stanno avendo un successo, mediatico e di utilizzo, in continua crescita e hanno dato una notevole spinta verso la democratizzazione della creazione di software ludici. Infatti, un numero consistente di motori grafici sono diventati liberamente accessibili a chi vuole addentrarsi nella creazione di un gioco, senza dover pagare alcun tipo di licenza. Inoltre, l'industria videoludica è in costante crescita e il giro di affari coinvolto supera persino quello di altre forme di intrattenimento, come cinema o musica.[1] Quando sono coinvolti enormi fonti di denaro significa che l'attenzione è massima per cercare di creare prodotti validi, efficienti e costantemente foriere di novità, al passo delle ultime scoperte tecnologiche. Particolare cura è rivolta anche all'usabilità, in quanto sempre più persone, esperte o neofite, fanno utilizzo di questi framework per cimentarsi in tale fiorente mercato.

Tutte queste caratteristiche fanno sì che i motori grafici vengano utilizzati anche al di fuori del mondo dei videogiochi. A prova di ciò, non è raro incontrare l'utilizzo di game engines nel modo accademico e in ricerche scientifiche. Tali framework risultano particolarmente utili nella creazione di simulazioni o di rappresentazioni virtuali di uno scenario reale. Sono stati oggetti di studio anche nella loro efficacia nella creazione di sistemi multi-agente[10].

Questa tesi si pone come obiettivo la riproduzione del modello di coordinazione *Spatial Tuples* mediante gli strumenti forniti da **Unity**. In questo modo si cerca di dare supporto tecnologico concreto e moderno, di un motore grafico affidabile, alla sperimentazione del modello *Spatial Tuples* su rappresentazioni 3D del mondo reale.

I concetti di spazio fisico e posizione nella realtà, caratteristiche fondanti del modello di coordinazione preso in analisi, sono stati riproposti in Unity mediante l'utilizzo del plug-in di **Google Maps**, attraverso il quale vengono caricate mappe

[1] iideassociation.com; marketwatch.com; newzoo.com

3D del mondo reale. Mappe in cui gli agenti possono spostarsi liberamente e sulle quali, poi, vengono implementati i principi proposti nel modello *Spatial Tuples*. Conseguenza di tale lavoro risulta in un duplice guadagno:

- da un lato si estendono le funzionalità di Unity aggiungendo un nuovo livello di astrazione, offerto dal modello di coordinazione. In questo modo si offrono nuovi modi e nuove soluzioni per affrontare la progettazione di sistemi complessi, dando un notevole contributo all'interazione degli oggetti presenti nella scena di un videogioco o di una simulazione;

- dall'altro lato, si sfrutta il progresso tecnologico raggiunto dai Game Engines in generale, per dare una rappresentazione concreta e materiale dei modelli teorizzati, in modo da poterli testare e verificare attraverso strumenti utilizzati da milioni di persone. Il settore videoludico, infatti, può vantare un enorme supporto economico e innumerevoli sviluppatori, impegnati a migliorare costantemente la stabilità, le prestazioni e l'usabilità dei propri prodotti.

Struttura della tesi. La tesi è organizzata come riportato di seguito.

Nel Capitolo 2 vengono illustrati gli argomenti utili alla trattazione di questa tesi. Si introduce il concetto di sistema multi-agente. Successivamente, si spiega l'importanza dei modelli di coordinazione e se ne fa una panoramica generale, introducendo anche il modello preso in esame da questa trattazione: *Spatial Tuples*. In seguito, si spiegano le astrazioni e funzionalità del motore di gioco utilizzato nel progetto, Unity, e si presentano le caratteristiche del plug-in di Google Maps. Infine, viene offerta una panoramica del lavori precedenti, presentando differenze e similitudini.

Nel Capitolo 3 viene mostrato il cuore di questa tesi. Si illustrano le implementazioni in Unity delle astrazioni del modello *Spatial Tuples*, prendendo in esame il concetto di tupla spaziale, di regione e di componente situato. Successivamente, vengono definite le specifiche di funzionamento delle primitive del modello, e la loro implementazione.

Nel Capitolo 4 vengono trasportati in Unity i pattern di coordinazione definiti nel modello *Spatial Tuples*.

In conclusione, nel Capitolo 5 si discute del lavoro svolto in questa tesi e si offre una visione su eventuali sviluppi futuri.

Capitolo 2

Background

In questo capitolo viene dato contesto al lavoro svolto dalla tesi. Si descrivono i concetti relativi ai sistemi multi-agente. Viene fornita una panoramica dei modelli di coordinazione e si esplora il modello di *Spatial Tuples*. Inoltre, si offre una visione generale del Game Engine Unity (motore grafico utilizzato da questa tesi).

Infine, si illustrano i lavori precedenti con argomenti affini a questa trattazione, con scopi e obiettivi assimilabili al lavoro svolto dalla tesi, evidenziando differenze e analogie, ispirazioni e diversità di vedute.

2.1 Sistemi multi-agente

I sistemi software moderni stanno diventando sempre più complessi da progettare, da sviluppare e da mantenere. La programmazione *agent-oriented* cerca di venire in soccorso a programmatori e ingegneri, in modo da rendere più semplice e agibile la costruzione di sistemi artificiali complessi e articolati [11].

I *sistemi multi-agente* pongono l'accento su una moltitudine di **entità computazionali** che lavorano interagendo all'interno di una **società**, non più, quindi, considerando solamente una unica entità che lavora da sola. I sistemi sono visti come un insieme di agenti che lavorano in maniera proattiva (sono quindi autonomi) per raggiungere il loro scopo, con la possibilità di interagire con altri agenti e con l'ambiente circostante (Figura 2.1) [20].

L'agente, quindi, può essere definito come una entità computazionale *autonoma*, che ha bisogno di *informazioni* per decidere quali *azioni* intraprendere per raggiungere il proprio *scopo*. Inoltre, è *situato* e immerso in un ambiente, nel quale si può spostare(è quindi dotato di *mobilità*) e nel quale compie *interazioni*, con altri agenti o con l'ambiente stesso. Non sono solamente le comunicazioni fra agenti ad assumere un ruolo fondamentale nel sistema da sviluppare, ma anche

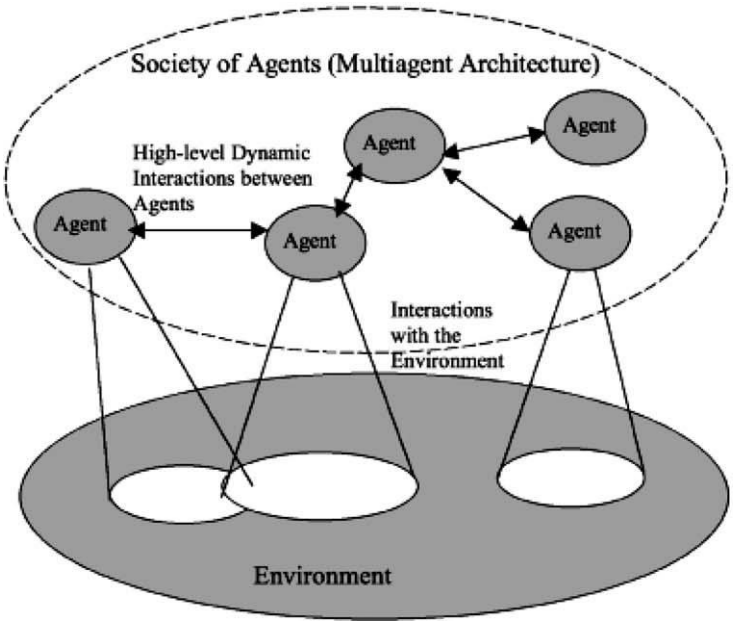


Figura 2.1: Architettura di un sistema multi-agente, tratto da [20]

le interazioni con l'**ambiente** costituiscono un aspetto fondante e astrazione di prima classe di un sistema multi-agente[19].

Inoltre, non basta considerare solamente il lavoro del singolo agente ma occorre prendere in esame anche le funzioni svolte dalla moltitudine di entità, vista come una società e portatrice di comportamenti collettivi. Quest'ultimi sono quindi uno strato ulteriore da aggiungere al comportamento individuale del singolo agente, e sono regolati attraverso lo strato di comunicazione e interazione. Lo spazio dell'interazione fra agenti non è più visto come solo spazio di comunicazione e scambio di informazioni, ma anche come spazio di coordinazione e regolatore di comportamenti sociali [5].

Riassumendo, si può sostenere che un sistema multi-agente sia costituito da tre astrazioni fondamentali.

Agenti: le entità autonome che compongono il sistema. Hanno uno scopo da raggiungere in modo proattivo – in maniera autonoma –, sono situati in un ambiente – quindi strettamente legati al contesto circostante –, nel quale sono mobili e dinamici, e sono in grado di comunicare e interagire con altri agenti – assumendo anche comportamenti sociali – e con l'ambiente nel quale sono immersi – ne possono percepire lo stato e adattare le proprie azioni e comportamenti di conseguenza;

Società: gruppo di entità computazionali il cui comportamento collettivo coordinato emerge dall'interazione tra i singoli agenti. Lo spazio delle interazioni

è un requisito fondamentale per la definizione di società, in quanto è proprio in tale spazio di coordinazione che avvengono i comportamenti sociali e si possono raggiungere obiettivi non definiti da un singolo agente, e quindi perseguibili solo attraverso la collaborazione e interazione fra più agenti. I modelli di coordinazione saranno trattati nella sezione 2.2.

Ambiente: lo spazio in cui gli agenti sono immersi e con il quale interagiscono, modificandone lo stato [15]. Gli agenti si definiscono *situati* proprio perché vivono in un ambiente, ne percepiscono lo stato e sono in grado di cambiarlo. L'ambiente consente la comunicazione e la coordinazione indiretta tra agenti e risorse esterne. Costituisce un'astrazione di prima classe [19] e, mediante un modello di coordinazione ben definito, permette lo sviluppo di azioni e meccaniche strettamente correlate con proprietà ambientali, sotto forma di azioni situate e percezione di cambiamenti ambientali.

2.2 Modelli di coordinazione

La coordinazione può essere definita come la gestione di interazioni e dipendenze tra attività, e quindi tra agenti, in un sistema multi-agente [9, 4]. Un modello di coordinazione si occupa della comunicazione fra agenti, della loro distribuzione e mobilità nello spazio, e della loro sincronizzazione e distribuzione di azioni nel tempo [5].

Coordinazione e interazione permettono, quindi, di ottenere meccanismi e astrazioni in grado di affrontare e gestire la complessità di un insieme di agenti dinamico ed eterogeneo. Gli agenti sono liberi da responsabilità di coordinazione e protocolli di interazioni. Questi, infatti, vengono delegati allo spazio di comunicazione fra gli agenti, elevandolo di conseguenza come *astrazione di prima classe* ed elemento fondante del processo di ideazione di un sistema multi-agente [5]. La coordinazione ha il ruolo fondamentale di incarnare regole e protocolli sociali, fornendo astrazioni in grado di gestire e governare le interazioni fra agenti [18]. I modelli di coordinazione determinano tali regole e protocolli.

In [4] viene spiegato che un modello di coordinazione può essere pensato e concepito come costituito da tre elementi:

- i **coordinables**, entità coordinabili – agenti, nel caso di sistemi multi-agente – in grado di cooperare, interagire e competere all'interno di uno spazio di coordinazione, in cui le regole sono definite dal modello;
- i **media di coordinazione**, astrazioni che permettono le interazioni fra agenti. Rappresenta il nucleo attorno al quale sono organizzati i componenti di un sistema coordinato (alcuni esempi possono essere semafori, monitor, spazi di tuple e blackboard);

- le **leggi di coordinazione**, che definiscono il comportamento dei media di coordinazione in risposta agli eventi di interazione. Queste leggi possono essere espresse in termini di *linguaggio di comunicazione*, sintassi usata per esprimere e scambiare strutture di dati, e *linguaggio di coordinazione*, un insieme di primitive di interazione e la loro semantica.

Secondo [12], i modelli di coordinazione possono suddividersi in due classi: *control-driven* e *data-driven*.

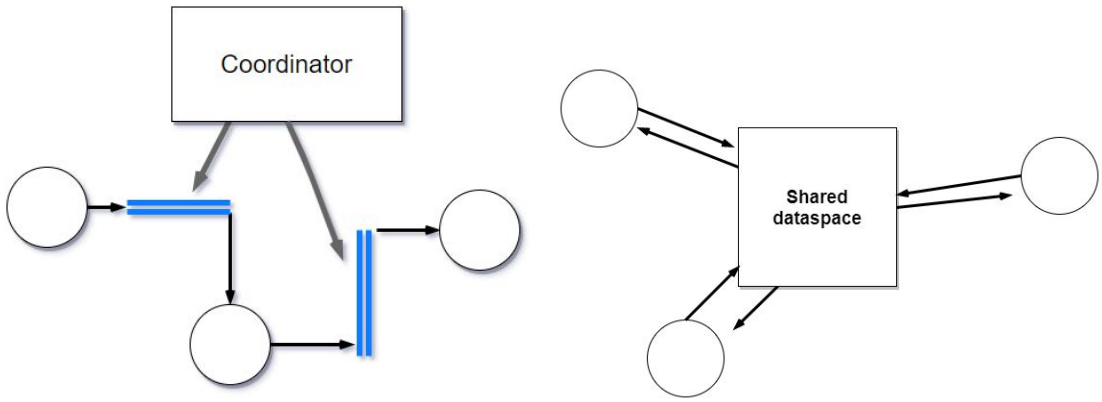
Control-driven. In un modello di coordinazione control-driven, la comunicazione fra componenti avviene mediante l'utilizzo di canali/porte. L'osservazione degli *eventi* e dei *cambi di stato* che avvengono su queste porte determinano l'interazione dei componenti. Input e output sono ben definiti e le leggi di coordinazione stabiliscono in che modo gli eventi e i cambi di stato possano avvenire e come debbano propagarsi. Conseguenza di tutto ciò è che i media di coordinazione gestiscono l'interazione fra i componenti senza curarsi dei tipi di dati e informazioni che vengono scambiati(Figura 2.2a).

Data-driven. In un modello di coordinazione data-driven, le entità coordinabili interagiscono con lo spazio esterno scambiando *strutture di dati* e pezzi di informazioni attraverso i media di coordinazione, che, quindi, agiscono come *spazi di dati condivisi*. Le leggi di coordinazione stabiliscono in che modo debbano essere rappresentate le strutture dati e come vengono organizzate, accedute e consumate. I media di coordinazione non forniscono nessuna connessione fra coordinabili e non ne percepiscono i cambi di stato. Conseguenza di tutto ciò è che l'interazione è definita dalla rappresentazione delle strutture dati, dal loro uso, accesso, manipolazione e sincronizzazione, sfruttando primitive di coordinazione fornite dal modello (Figura 2.2b).

Numerosi modelli sono stati proposti e utilizzati in letteratura e la ricerca scientifica è molto attiva da questo punto di vista. In [6] viene offerta una panoramica degli ultimi venti anni per quanto riguarda modelli e tecnologie relativi alla coordinazione. Come riportato nella figura 2.3, si può notare come la maggior parte delle tecnologie di coordinazione derivino da due modelli differenti, LINDA e IWIM, rispettivamente data-driven e control-driven. Nella prossima sezione si prende brevemente in analisi il modello LINDA, in quanto base del modello di coordinazione *Spatial Tuples*, preso come riferimento da questa tesi.

2.2.1 LINDA e i modelli basati su spazi di tuple

LINDA è alla base di ogni *modello di coordinazione basato su tuple* ed è il primo modello ad aver fornito meccanismi ed astrazioni per spazi di dati, programmazione



(a) Coordinazione *control-driven* (b) Coordinazione *data-driven*

Figura 2.2: Le due classi di modelli di coordinazione

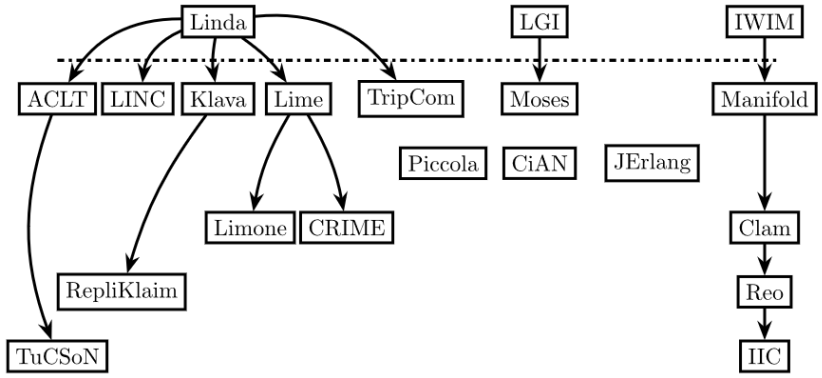


Figura 2.3: Linea di evoluzione dei modelli di coordinazione, tratto da [6]

di agenti concorrenti e comunicazione generativa. Utilizzata inizialmente nel campo della programmazione parallela [9], i modelli di coordinazione basati su tuple espongono proprietà che li rendono adatti alla coordinazione di sistemi *eterogenei* e *distribuiti*, affrontandone la complessità con astrazioni e concetti semplici ma allo stesso tempo ben congegnati e consolidati.

Nei modelli basati su tuple, le entità coordinabili interagiscono fra di loro scambiando tuple, cioè pezzi di informazioni. L'interazione fra queste entità avviene, quindi, mediante la produzione, l'accesso e il consumo di questi pezzi di informazioni, usando gli *spazi di tuple* come media di coordinazione. Il linguaggio di comunicazione e coordinazione portano alla formazione di una comunicazione generativa, fornendo primitive semplici ma allo stesso tempo molto espressive.

Riassumendo, LINDA definisce una collezione ordinata ed eterogenea di tuple, che incapsulano informazioni destinate ad essere scambiate fra agenti. Queste informazioni sono disponibili negli spazi di tuple, che costituiscono, quindi, l'astrazione dei media di coordinazione e fungono da contenitori di tuple. Gli spazi di tuple condivisi possono essere manipolati e modificati attraverso astrazioni di accesso associativo, mediante meccanismi di abbinamento di tuple, ad esempio pattern matching e unificazione.

Il linguaggio di coordinazione LINDA fornisce tre *primitive* fondamentali, semplici ed espressive, in grado di manipolare e modificare le tuple e gli spazi di tuple [8]:

- `out(T)`: inserisce la tupla `T` nello spazio di tuple;
- `in(T)`: recupera la tupla `T` dallo spazio di tuple, con le seguenti proprietà:
  - **semantica distruttiva**: la tupla recuperata è rimossa dallo spazio di tuple;
  - **semantica di sospensione**: se non è stata trovata nessuna tupla corrispondente al pattern cercato, l'operazione è bloccata e l'esecuzione sospesa finché non viene trovata una tupla `T` valida;
  - **non-determinismo**: se vengono trovate più tuple corrispondenti al pattern cercato, LINDA ne restituisce una, scelta non deterministicamente.
- `rd(T)`: recupera la tupla `T` dallo spazio di tuple, con le seguenti proprietà:
  - **semantica non-distruttiva**: la tupla recuperata NON viene rimossa dallo spazio di tuple;
  - **semantica di sospensione**: se non è stata trovata nessuna tupla corrispondente al pattern cercato, l'operazione è bloccata e l'esecuzione sospesa finché non viene trovata una tupla `T` valida;

  - **non-determinismo**: se vengono trovate più tuple corrispondenti al pattern cercato, LINDA ne restituisce una, scelta non deterministicamente.

LINDA fornisce anche operazioni predicative e non-bloccanti, `inp(T)` e `rdp(T)`: rimangono invariate le proprietà di lettura distruttiva/non-distruttiva e di non-determinismo ma viene a mancare la semantica di sospensione, introducendo una semantica di *fallimento*. Ne consegue che, se non viene trovata alcuna tupla corrispondente al pattern ricercato, si segnala un fallimento; altrimenti si segnala il successo. In ogni caso, dunque, non avviene alcuna sospensione.

Le operazioni LINDA definiscono un linguaggio di coordinazione adatto a gestire le complessità di un sistema multi-agente. Come illustrato in [17], sono fornite le seguenti proprietà:

- **Estensibilità** – sebbene concepito originariamente per sistemi paralleli e chiusi, in seguito a continui processi di sviluppo, è stato esteso con nuovi meccanismi potenti ma allo stesso tempo mantenendone la semplicità, risultando adatto anche a scenari diversi;

- **Espressività** – è efficace a risolvere i problemi tipici della coordinazione e ad affrontare la complessità di progettazione dei sistemi multi-agente, attraverso poche ma semplici ed efficaci primitive;

- **Semantica sospensiva** – le primitive di `rd` e `in` possono portare alla sospensione nel caso non si trovi una tupla corrispondente al pattern di tupla ricercato. Il processo di sospensione può avvenire sia nel medium di coordinazione (attraverso la sospensione dell'operazione e poi la conseguente riattivazione) sia nelle entità coordinabili (mediante uno stato di attesa interno finché l'operazione non ritorna con successo);

- **Comunicazione generativa** – le tuple generate dalle entità coordinabili, e che vivono nello spazio delle tuple, sono indipendenti l'una dall'altra. Le tuple rappresentano il mezzo attraverso il quale avviene la coordinazione e la comunicazione;

- **Disaccoppiamento** – gli agenti non devono necessariamente risiedere nello stesso spazio in cui risiedono le tuple e non occorre alcuna referenza, simultaneità o necessità di un nome. Le tuple hanno esistenza propria e non c'è alcuna correlazione di tempo, di spazio e di referenza fra agenti e tuple;

- **Data-driven** – la coordinazione avviene in base al contenuto dell'informazione che si intende scambiare. L'accesso all'informazione è basato sulla disponibilità o meno del dato cercato;

- **Separazione dei ruoli** – LINDA si occupa solo dei problemi relativi alla coordinazione, e gli agenti non hanno bisogno di preoccuparsi di tali problemi;
- **Asincronia** – conseguenza del disaccoppiamento. Non c'è garanzia che una informazione ricercata esista e non c'è garanzia che una informazione fornita sia ricercata da qualcuno al momento del rilascio;
- **Concorrenza** – le tuple vivono di vita propria e non hanno alcun legame di referenza, e in quanto tale possono essere accedute da più elementi simultaneamente.

Come ulteriore prova dell'estensibilità ed espressività appartenente al modello di coordinazione LINDA, nella figura 2.3 si può notare una grande quantità di modelli e tecnologie che sono derivati da LINDA. Tutti questi modelli di interazioni offrono gli stessi semplici ed espressivi meccanismi ed astrazioni di base del modello da cui derivano, con l'aggiunta di variazioni ed estensioni volte a soddisfare requisiti di aree di ricerca differenti.

2.2.2 Spatial Tuples

Introdotto in [16], il modello di coordinazione per sistemi distribuiti multi-agente denominato *Spatial Tuples* consiste in una estensione del modello basato su tuple, in cui: (*i*) le tuple sono collocate in regioni dello spazio fisico del mondo reale e in grado di spostarsi all'interno di esso, (*ii*) il comportamento delle primitive LINDA standard di coordinamento è esteso in modo da dipendere dalle proprietà spaziali degli agenti, delle tuple e della topologia dello spazio, e (*iii*) lo spazio delle tuple può essere visto come uno strato virtuale che decora la realtà fisica.

Questo modello nasce con l'intento di supportare la coordinazione di agenti in uno scenario di computazione pervasiva, basato sullo spazio e sulla posizione fisica, in risposta alla sempre più crescente diffusione di tecnologie mobili, indossabili e pervasive (Figura 2.4).

La nozione di spazio viene considerata come astrazione di prima classe e permette di sviluppare e definire nuove strategie e pattern di coordinazione che si basano sulla posizione e sul movimento delle entità situate nel sistema. Due esempi molto espressivi sono la *comunicazione situata*, in cui il messaggio viene percepito solo dagli agenti situati in un determinato luogo, e la *sincronizzazione spaziale*, in cui l'ordine delle azioni delle entità è basato sulla loro posizione fisica.

Il modello *Spatial Tuples* introduce il concetto di **tupla spaziale**, pezzo di informazione che possiede una *posizione* ed una *estensione* nello spazio. Ne consegue che, in questo modello, una tupla è legata indissolubilmente in ogni momento ad una regione dello spazio, rappresentando, dunque, un arricchimento della realtà

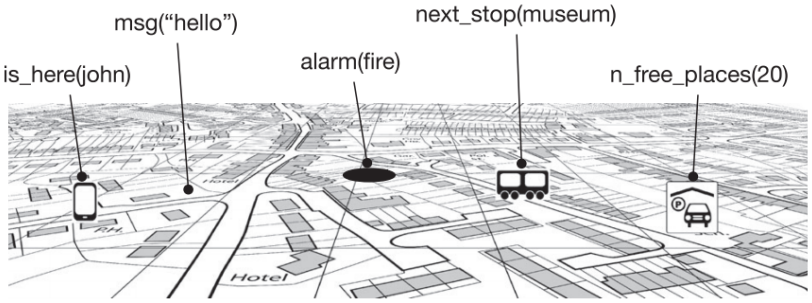


Figura 2.4: Tuple spaziali che aumentano la realtà fisica, tratto da [16]

fisica ed una informazione che vive nell'ambiente reale. Tale informazione spaziale può essere di qualsiasi tipo: posizione geografica (latitudine e longitudine), metrica (100m attorno alla mia posizione attuale), amministrativa (via Zamboni 33, Bologna, Italia).

Spatial Tuples risulta molto efficace in tutti gli scenari relativi alla *realtà aumentata*. In tale contesto, come definito da [1], (*i*) la realtà fisica è arricchita con informazioni digitali situate in una posizione del mondo reale e (*ii*) queste informazioni vengono percepite dall'utente umano per mezzo di specifici dispositivi (come occhiali smart, visori, o smartphone).

La tupla spaziale è ciò che decora lo spazio fisico e che aumenta la realtà con informazioni digitali. Una volta che una tupla spaziale è associata ad una regione, le sue informazioni sono attribuite ad una porzione dello spazio fisico, aggiungendo, in questo modo, proprietà osservabili a tali porzioni. Lo spazio è modellato come un insieme non vuoto di **regioni**. La posizione di una tupla è rappresentata da una regione, che può consistere anche in un singolo punto.

Il modello *Spatial Tuples* aggiunge un linguaggio spaziale, cioè che introduce specifiche informazioni relative allo spazio, al linguaggio di comunicazione delle primitive di base LINDA (Figura 2.5).

Le tuple spaziali vengono associate ad una regione `r` mediante una operazione di `out`:

`out(t@r)`: associa la tupla spaziale `t` alla regione `r`.

Le tuple spaziali vengono recuperate dalle regioni mediante operazioni di `in` e di `rd`:

`rd(tt@rt)` e `in(tt@rt)`: cercano una tupla `t` che corrisponde al template `tt`, collocata in qualunque regione `r` che corrisponda al template `rt`.

La corrispondenza fra regioni si ottiene osservando la loro intersezione: una regione `r` corrisponde al template di regione `rt` se la loro intersezione non è vuota.

Figura 2.5: Funzionamento delle primitive *Spatial Tuples*, tratto da [16]

Come in LINDA, l'operazione di `in` si distingue dall'operazione di `rd` in quanto rimuove la tupla dalla regione. Allo stesso modo, sono mantenute le proprietà di *sospensione* e *non-determinismo*.

Le tuple spaziali possono essere associate alle regioni in modo diretto o in modo *indiretto*. Le primitive viste fino ad ora associano direttamente la tupla `t` alla regione `r`. Tuttavia, una tupla spaziale può essere associata anche ad una entità del sistema collocata in uno spazio fisico, un **componente situato**.

Ne deriva, quindi, una associazione indiretta, scritta in questo modo:

`out(t@id)`: associa la tupla `t` al componente situato `id`. In modo indiretto, quindi, la tupla risiede nella regione `r` in cui si trova il componente situato.

Il punto focale è che questa associazione rimane anche quando la regione in cui è collocato `id` cambia nel tempo: `t` è quindi associato a qualunque posizione nello spazio in cui si trova `id` nel corso del suo ciclo.

Ci sono casi in cui l'agente che esegue un'operazione risulti esso stesso un componente situato. Per questo caso, sono introdotte due parole chiave, **here** e **me**. Pertanto, sono definite le seguenti operazioni:

- `out(t@here)`
- `out(t@me)`

Entrambe associano una tupla `t` alla regione `r` dove è attualmente collocato il componente situato. Nel caso di `here`, la tupla spaziale è associata in modo permanente alla regione `r`, anche se il componente dovesse spostarsi successivamente. Nel caso di `me`, la regione cambia non appena il componente si muove.

Allo stesso modo sono definite le operazioni

`rd(tt@here)` - `rd(tt@me)` - `in(tt@here)` - `in(tt@me)`

Un ulteriore scenario potrebbe essere quello in cui non si voglia considerare una regione definita, bensì una regione estesa, funzione di una regione esistente. Viene, quindi, definito il costrutto `region(id,s)`, dove `id` è l'identificativo del componente situato, ed `s` è una funzione che definisce la *forma* della regione, estendendo quindi la regione di appartenenza del componente situato.

2.3 Game Engine

2.3.1 Unity

Unity[1] è un motore di gioco multipiattaforma usato per la creazione di videogiochi e di simulazioni virtuali. Si tratta di uno dei maggiori contributori alla democratizzazione dello sviluppo di software videoludici, in quanto è accessibile in modo del tutto gratuito e offre molte funzionalità utili e interessanti, atte alla creazione rapida ed efficace di videogiochi e applicazioni.

Unity mette a disposizione degli sviluppatori numerosi strumenti semplici da utilizzare per creare giochi realistici e simulazioni immersive, come (*i*) un editor real-time facile e intuitivo, (*ii*) un sistema di fisica integrato, (*iii*) luci dinamiche e shader, (*iv*) creazione di modelli 2D e 3D direttamente nell'editor oppure la possibilità di importarli da strumenti esterni, (*v*) gestione delle texture, (*vi*) supporto a funzionalità basilari di intelligenza artificiale (ricerca del percorso, simulazione della folla, possibilità di evitare ostacoli, funzionalità di machine learning e altro ancora).

Questo motore offre un'architettura interna semplice e modulare, sfruttabile dagli sviluppatori per ideare e creare i propri progetti, utilizzando le astrazioni di Unity per gestire le complessità di un sistema videoludico.

Il concetto fondamentale è costituito dai **GameObjects**. Tutto ciò che è presente in una *scena* (collezione di assets e oggetti) è considerato un GameObject. Si tratta della classe base per ogni costrutto di Unity.

Ogni oggetto è costituito da diversi blocchi, chiamati **Components**. Ogni componente permette di aggiungere proprietà, caratteristiche e funzionalità diverse al GameObject. In questo modo, combinazioni differenti di componenti permettono di modellare il tipo di oggetto che lo sviluppatore intende creare. Alcuni esempi di componenti importanti e maggiormente usati sono:

- *Transform*: usato per memorizzare e manipolare la posizione, rotazione e dimensione di un oggetto. Ogni oggetto possiede di default questo componente;

- *Rigidbody*: Permette di applicare le leggi della fisica all'oggetto. Mediante esso, il GameObject è soggetto alle forze e agli spostamenti e, in generale, viene preso in considerazione dal motore di fisica;

- *Collider*: responsabile della gestione delle collisioni fra oggetti. Può avere forme e dimensioni variabili, e non per forza uguali al GameObject di cui è componente;

[1] `https://unity.com/`

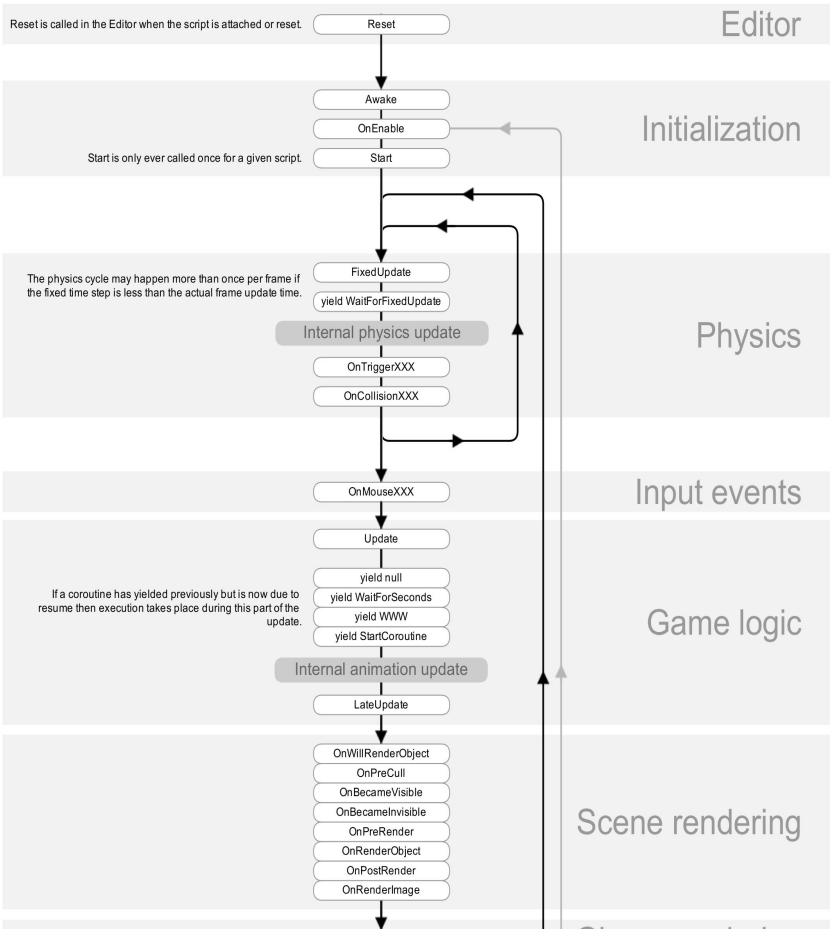


Figura 2.6: Parte dello schema di funzionamento di Unity

- *Animator*: abilita le animazioni sull'oggetto;
- *Script*: all'interno di esso si può definire in che modo deve comportarsi il GameObject. Di solito viene scritto in **C#**.

Unity ha un preciso schema di funzionamento (Figura 2.6).[2] Il controllo di esecuzione è gestito da un solo e *unico* thread, dedito a quello che viene chiamato *Game Loop*. Il thread esegue questo loop a ogni *frame*, chiamando tutti gli script che sono abilitati nella scena corrente.[3] Conseguenza di ciò, non esiste parallelismo.

[2]tratto dal Manuale di Unity, nella voce in cui viene spiegato l'ordine di esecuzione delle funzioni. `https://docs.unity3d.com/Manual/ExecutionOrder.html`

[3]Per far parte del *Game Loop*, uno script deve estendere la classe `UnityEngine.MonoBehaviour`. Se tale estensione non è presente, lo script verrà ignorato dal loop.

Ne deriva che, tutto il sistema deve essere altamente performante e ottimizzato, e gli script non possono fare calcoli onerosi, pena il blocco di tutto il sistema.

Una nota da sottolineare è che Unity permette la creazione di più thread ma l'accesso e la manipolazione degli oggetti (incluse tutte le operazioni relative alla fisica) è appannaggio esclusivamente del thread principale. Quindi, i thread secondari sono utili ad essere usati per calcoli onerosi, usualmente di tipo matematico.

Uno dei motivi principali per cui è stato scelto Unity per questa tesi è che possiede una fervida comunità, con molteplici forum di grande aiuto, una documentazione esplicativa e ben gestita, e viene largamente utilizzato in corsi accademici, ricerche scientifiche o da esperti programmatori di videogiochi, nel mondo commerciale e non.

2.3.2 Google Maps

Google Maps Platform gaming services è una piattaforma di sviluppo per creare giochi che presentano scene del mondo costruite a partire dai dati geospaziali di *Google Maps*, e renderizzati a run-time dal motore di gioco Unity. La piattaforma offre accesso semplice, espressivo e programmatico ai dati geografici.

Si possono creare ambienti di gioco immersivi che replicano gli scenari del mondo reale. In base alla *posizione* del giocatore, si ottiene una mappa di alta qualità, arricchita da strutture rappresentanti edifici e geometrie del mondo. Questa mappa può essere ulteriormente personalizzata modificando lo stile di rappresentazione, le texture degli edifici, gli effetti di luce e le proprietà fisiche (essendo oggetti di Unity, è possibile operare su di essi come con qualsiasi GameObject).

L'avatar di gioco caricato sulla mappa è in grado di navigare il mondo reale percorrendo le strade e di interagire con l'ambiente e il mondo di gioco.

La piattaforma di sviluppo contiene due componenti:

- **Maps SDK per Unity**: SDK per i dati di Google Maps, e una collezione di assets di Unity (materiali, script, plug-in e scene di esempio). Mediante questo SDK, le proprietà della mappa come edifici, strade e specchi d'acqua, divengono GameObjects nativi di Unity;
- **API dei luoghi giocabili**: propone luoghi candidabili ad essere utilizzati dalle meccaniche del gioco, come ad esempio spawn points o punti in cui ottenere oggetti di gioco. (Queste API non verranno utilizzate dalla tesi)

Maps SDK per Unity

Maps SDK per Unity è un insieme di strumenti di sviluppo, servizi e assets pronti all'utilizzo, che estendono l'ambiente di sviluppo di Unity con caratteristiche che

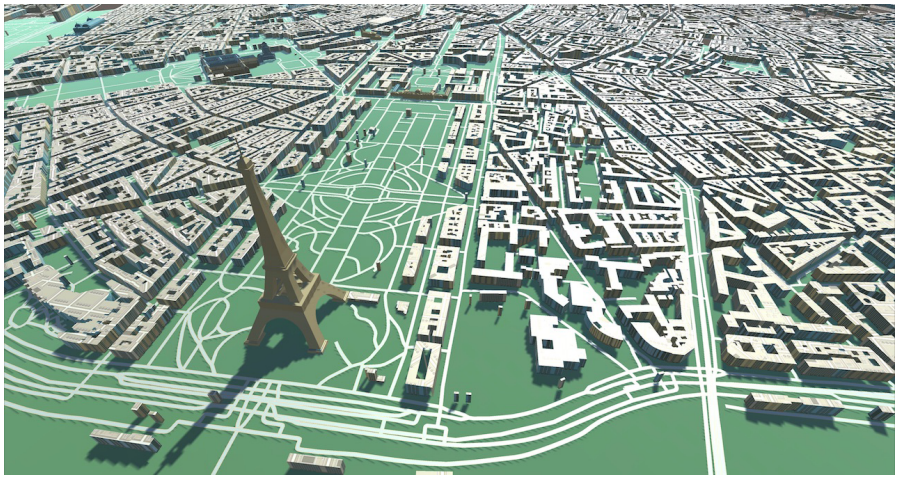


Figura 2.7: Mappa 3D di Google Maps raffigurante Parigi

permettono di creare facilmente giochi per mobile ambientati nel mondo reale (Figura 2.7).

Si ha accesso a dati geografici di alta qualità provenienti dal database di Google Maps. L'accesso avviene a run-time e genera il mondo di gioco in base alla *posizione fisica* del giocatore. Viene caricata la mappa dalla qualità più alta disponibile e, su di essa, vengono renderizzati elementi geografici come edifici, modelli 3D di monumenti, strade e superfici d'acqua. Tutti questi elementi sono accessibili come GameObject di Unity.

Ogni GameObject rappresentante elementi geografici possiede i seguenti componenti: *Transform*, *Mesh Renderer* e *Mesh Filter*. Questo offre un ottimo punto di partenza dal quale poter personalizzare gli elementi con componenti che ne cambiano lo stile o gli effetti, come *Rigidbody*, *Collider* e *Material*. Inoltre, ogni elemento geografico possiede dei *Metadata*, contenenti per esempio un *placeId* e un *nome*.

Per ulteriori dettagli si rimanda alla documentazione ben fornita.[4] Di seguito viene posta attenzione su alcune peculiarità che vengono particolarmente utilizzate nel lavoro svolto da questa tesi.

MapsService. La classe `MapsService` funge da punto di partenza per interagire con Maps SDK per Unity. Essa incapsula la *ApiKey*,[5] il *GameObjectManager*

[4] `https://developers.google.com/maps/documentation/gaming`

[5] Con API key ci si riferisce ad una stringa criptata di valori alfa-numerici, usata per accedere alle API rese disponibili dalla piattaforma Cloud di Google. Le API keys vengono utilizzate per tener conto delle richieste di API, in modo da aver traccia della quota di richiesta, ai fini di determinare la fatturazione da attribuire all'utilizzatore. Infatti, i costi della piattaforma Cloud di Google sono determinati da delle soglie di utilizzo. Entro una certa soglia, l'utilizzo è gratis.

(notifica il SDK ogniqualvolta un GameObject viene distrutto, in modo da poterlo ricreare nel momento delle successive chiamate di caricamento della mappa), la funzione `LoadMap`(carica una regione rettangolare dalle dimensioni specificate), e gli *Eventi* generati dalla creazione di GameObject o dal caricamento della mappa.

Per poter utilizzare Maps SDK per Unity, occorre aggiungere il componente *script* chiamato `MapsService` ad un GameObject vuoto nella scena di Unity. Lo script aggiunge automaticamente come figlio di questo oggetto ogni GameObject rappresentante elementi geografici della mappa;

Sistema di coordinate. Maps SDK per Unity utilizza un sistema di coordinate basato su latitudine e longitudine, mentre Unity crea un sistema cartesiano per orientarsi nel mondo virtuale generato – *Unity Worldspace* (`Vector3` che contiene il valore x, y e z).

Il centro da cui si carica la mappa di Google Maps e il centro del *Worldspace* di Unity sono gli stessi e, tipicamente, consistono nella posizione iniziale dell'utente. Tuttavia, è bene precisare che i valori di `Vector3` relativi all'origine fanno affidamento alla singola sessione di utilizzo. Quindi, se si vuole posizionare un oggetto in un modo più definitivo, è consigliabile usare i valori di latitudine e longitudine.

Unity Worldspace ha una scala 1:1, misurata in metri.

Road lattice. Ogni mappa caricata contiene un *reticolo stradale*, che rappresenta, attraverso un grafo direzionato, l'infrastruttura di strade della porzione di spazio caricata (Figura 2.8).

- La classe `RoadLattice` incapsula tutti i nodi relativi ad un reticolo. Viene aggiornata ogni volta che una regione della mappa viene caricata o rimossa;
- La classe `RoadLatticeNode` rappresenta il singolo nodo del reticolo. Ogni volta che viene caricata una regione della mappa, i nodi corrispondenti ai vertici della linea spezzata, che può raffigurare, ad esempio, una strada, sono inseriti nel reticolo stradale corrispondente alla regione considerata. Un nodo può essere condiviso da più linee (ad esempio, gli incroci di strada);
- La classe `RoadLatticeEdge` rappresenta il segmento che unisce i vari nodi. Contiene metadati utili per la ricerca di percorsi.

In questo modo, si paga solo in base all'effettivo quantitativo di utilizzo. Per questa tesi sono state attivate la Semantic Tile API e la Playable Locations API (indispensabili per poter utilizzare Google Maps) e la Geocoding API (per convertire gli indirizzi in coordinate geografiche).

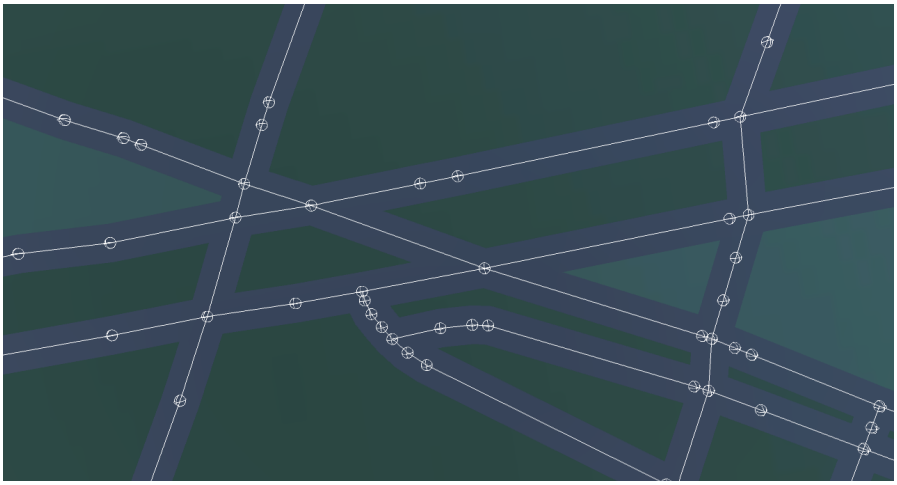


Figura 2.8: Reticolo stradale della mappa caricata

Ci sono quattro modi per accedere ad un nodo del reticolo:

1. Attraverso l'iterazione dei nodi in `RoadLattice.Nodes`;
2. Attraverso la posizione nel reticolo, con `RoadLattice.FindNodeAt`
3. Restituendo il nodo più vicino ad una posizione reale del mondo, con `RoadLattice.SnapToNode`;
4. attraverso i suoi nodi vicini, con `RoadLatticeNode.Neighbors`.

Unity offre un sistema di navigazione della scena nel mondo virtuale, attraverso il componente *NavMesh*. Maps SDK per Unity propone una funzionalità simile attraverso l'infrastruttura stradale contenuta nella regione di mappa caricata. Vengono offerti metodi per calcolare il percorso più breve fra due nodi del reticolo. L'avatar di gioco si troverà, quindi, a raggiungere un determinato punto seguendo il percorso delle strade, evitando di tagliare la mappa da un punto `A` ad un punto `B` in modo innaturale. Il metodo `RoadLattice.FindPath` calcola il percorso fra due nodi e restituisce la collezione di nodi da dover attraversare per raggiungere la destinazione.

2.4 Lavori precedenti

I lavori precedentemente svolti, che hanno contribuito alla definizione e agli obiettivi di questa trattazione, utilizzano due approcci differenti per integrare i sistemi multi-agente e i game engines:

1. Integrare all'interno del game engine un *framework esterno*, che abilita le funzionalità e le caratteristiche per sviluppare sistemi multi-agente;

2. Realizzare un *middleware* di qualche sorta, in grado di collegare e far comunicare l'ambiente relativo al sistema multi-agente e l'ambiente del game engine, che rimangono, quindi, due entità separate.

2.4.1 Integrazione di framework esterni

Si tratta del lavoro su cui si basano [14, 3, 2]: (*i*) [14] integra in Unity il modello **BDI**(*Beliefs*, *Desires*, *Intentions*) per la programmazione degli agenti, mentre (*ii*) [3, 2] integrano un modello di coordinazione, rispettivamente LINDA e *Spatial Tuples*. L'idea di base che sottostà a questi lavori è l'integrazione in Unity dell'*interprete Prolog* chiamato **UnityProlog**. Questo interprete dispone di funzionalità per estendere le operazioni Prolog ai GameObject di Unity. In questo modo si può accedere e manipolare i GameObject e i relativi componenti direttamente da Prolog.

Ne deriva che, è proprio dall'utilizzo dell'interprete UnityProlog che è possibile creare agenti BDI. Ed è sulla base dell'interprete che vengono sviluppate le primitive utilizzate per il coordinamento definito da LINDA e *Spatial Tuples*. Vengono fornite librerie **C#** e API in modo che lo sviluppatore in Unity possa accedere alle funzionalità Prolog senza accorgersene.

2.4.2 Utilizzo di middleware

Si tratta del lavoro su cui si basano [7, 13]: il mondo del game engine e il mondo del sistema multi-agente rimangono separati e viene realizzato un canale di comunicazione per far interagire i due ambienti.

I GameObject sono visti come il *corpo* e gli agenti sono visti come la *mente*. Un corpo deve eseguire azioni e, in seguito a determinati eventi, deve trasmettere le proprie percezioni alla mente. La mente deve elaborare le percezioni per decidere quali azioni deve far svolgere al corpo. Per rendere possibile questa comunicazione, è stato progettato un middleware che funge da mezzo di comunicazione. In aggiunta, vi è la necessità di un protocollo di comunicazione tra le entità, utilizzando messaggi strutturati in un modo ben preciso.

Le due soluzioni si differenziano per i modi in cui è stato implementato il middleware: (*i*) in [7] il middleware è separato in due parti, una inglobata in Unity e l'altra nel sistema multi-agente: la comunicazione avviene fra questi due middleware; (*ii*) in [13] il middleware risulta un unico componente separato che mette in comunicazione i due sistemi. La comunicazione con il middleware avviene

mediante due librerie poste all'interno, rispettivamente, di Unity e del sistema multi-agente.

2.4.3 Confronto con la tesi presentata

La trattazione presentata in questo documento è stata ispirata dai lavori illustrati in 2.4. La fondamentale differenza consta nel fatto che si è scelto di **riprodurre** un modello di coordinazione utilizzando *solamente* gli strumenti proposti da Unity. Non vi è alcuna integrazione con framework esterni e non vi è alcuna presenza di middleware e canali di comunicazione. Si ritiene che questa soluzione abbia vantaggi in termini di *performance* e di *stabilità* del sistema.

Inoltre, questo progetto è fortemente radicato con l'utilizzo di Google Maps, realizzando concretamente la visione totalmente incentrata sulla posizione nel mondo reale, alla base di *Spatial Tuples*. Un tentativo, in tal senso, è stato proposto anche da [2] (utilizzando un plug-in diverso da quello di Google Maps e mancando, quindi, fra le altre cose, di un sistema di navigazione basato sulla percorrenza delle strade), il cui operato, tuttavia, si appoggia sulle integrazioni con framework (il motore Prolog), compiute da lavori precedenti.

Pertanto, le primitive illustrate in 3.4 mantengono la stessa sintassi e le stesse specifiche di funzionamento delle primitive presentate da [2] (in quanto derivano da quelle illustrate in [16]. Quello che cambia è l'intera filosofia interna e la struttura di implementazione, in quanto viene a mancare tutta la base di appoggio dell'integrazione dei lavori precedenti.

Capitolo 3

Spatial Tuples in Unity

Questo capitolo presenta le soluzioni e i meccanismi adottati per concretizzare l'obiettivo della tesi, cioè riprodurre il modello di coordinazione *Spatial Tuples* all'interno di Unity. Le principali astrazioni introdotte per soddisfare tale obiettivo sono la definizione del concetto di (*i*) tupla, (*ii*) regione e (*iii*) componente situato. Infine, vengono definite le primitive per la manipolazione delle tuple, e le loro specifiche di funzionamento.

Il progetto è ispezionabile, utilizzabile e modificabile attraverso un repository Git su *GitLab*.[1]

3.1 Tupla spaziale

La tupla, nei modelli di coordinazione, è vista come un qualsiasi tipo di informazione e mediante la quale si può ottenere *comunicazione* e *coordinazione*. Una **tupla spaziale** aggiunge a questa astrazione di base il concetto di posizione: una tupla, in ogni momento della sua vita, è collocata nello *spazio* e risiede in una *regione*.

In questa prima versione del progetto, l'informazione contenuta in una tupla è rappresentata da una *stringa* di caratteri. Non si esclude in futuro la possibilità di articolare in modo più espressivo e potente l'informazione contenuta nella tupla, utilizzando strutture diverse dalla stringa.

Per uno scopo puramente di funzionalità pratica e visiva, si è scelto di dare una forma concreta alla tupla, visibile nella scena creata in Unity. È bene sottolineare che tale resa visiva non ha alcun importanza ai fini della coordinazione, che è basata interamente sul messaggio contenuto nella tupla. Si è scelto di rendere la

[1] `https://gitlab.com/pika-lab/theses/thesis-pastore-ay2021`

tupla un GameObject di tipo `PrimitiveType.Sphere` e considerarlo un *Prefab*.[2] Il Prefab contiene anche lo *script* descritto di seguito.

Il componente script chiamato `Tuple` è ciò che caratterizza un GameObject che viene considerato come una tupla. Esso definisce un campo pubblico di tipo stringa chiamato `Message`. In più, viene fornito un metodo statico attraverso il quale produrre una tupla:[3] prende in ingresso una stringa che rappresenta l'informazione, crea una nuova istanza del Prefab della tupla, ne prende lo script `Tuple` e gli assegna come messaggio la stringa passata in ingresso.

3.2 Regione

Una **regione** è definita da una *forma* e da una *estensione*, ed è collocata in uno spazio ben preciso del mondo. Deve essere in grado di percepire gli altri oggetti, incluse altre regioni, con cui entra in contatto e deve essere consapevole di ciò che risiede all'interno di essa.

In termini di astrazioni di Unity, una regione è vista come un *Collider* di una certa forma e dimensione. Tramite la percezione delle collisioni si è in grado di capire quando le regioni si intersecano nello spazio: regioni con spazi in comune hanno i rispettivi *Collider* che entrano in collisione.

Inoltre, Unity offre il metodo `Physics.OverlapSphere` che effettua una ricerca nell'area definita dai parametri di ingresso, e restituisce i *Collider* trovati nella zona di ricerca. In questo modo, si ha un metodo pratico per conoscere tutti gli oggetti, dotati di un *Collider* (incluse, quindi, le regioni), presenti in una determinata area. Se si vuole limitare questa ricerca alle sole regioni, si sfruttano le funzionalità di `tag` e `layer` offerte da Unity. Si tratta di due costrutti utili a identificare e catalogare oggetti di uno stesso gruppo che condividono medesime proprietà. Si è scelto, quindi, di attribuire a tutte le regioni un `tag` e un `layer` specifico (`tag`: region, `layer`: 8).

È bene specificare che il *Collider* della regione deve essere impostato come `trigger`, tramite il campo pubblico `Collider.isTrigger`. In questo modo, il *Collider* non è soggetto alle forze della fisica e quindi non si sposta quando entra in contatto con qualcosa, risultando, di conseguenza, attraversabile. Ad ogni collisione, un *Collider* impostato come `trigger` segnala l'evento attraverso i metodi `OnTriggerEnter`, `OnTriggerExit` e `OnTriggerStay`. Due *Collider* `trigger` che

[2] Asset riutilizzabile che immagazzina tutte le proprietà e componenti assegnati al GameObject, in modo da evitare di dover ridefinire ogni volta lo stesso oggetto. Agisce da template da cui creare le istanze dell'oggetto.

[3] I Componenti non sono mai creati direttamente via codice. Quindi, le classi che derivano da `UnityEngine.MonoBehaviour` non posso essere inizializzate attraverso costruttori. Ciò che si fa, invece, è scrivere codice nello script e attaccare tale script al GameObject attraverso il metodo `GameObject.AddComponent`.

entrano in collisione non segnalano alcun evento. Per rimediare a ciò, è necessario aggiungere al GameObject della regione un componente *Rigidbody* di tipo `Kinematic`.

Anche se ai fini della coordinazione è sufficiente solamente il componente *Collider*, per ragioni visive e di praticità, è stato scelto di considerare la regione come un GameObject vero e proprio e di dotarlo di un materiale semi-trasparente.[4]

Lo script chiamato `Region` è ciò che definisce le proprietà e le funzionalità di una regione. Lo script contiene i campi `Region.Shape` e `Region.Extension`, che definiscono rispettivamente la forma e l'estensione della regione. In questa tesi, la forma è definita da `PrimitiveType.Sphere` o da `PrimitiveType.Cube`. Ciò è dovuto al fatto che, a livello computazionale, i *Collider* offrono prestazioni migliori se aventi la forma di un `PrimitiveType` (che sono *Sfera*, *Cubo*, *Cilindro*, *Capsula*, *Piano*, *Quadrilatero*). Tuttavia un *Collider* può avere qualsiasi forma arbitraria, anche complessa, e quindi una regione può potenzialmente avere qualsiasi conformazione. L'estensione è data da un `float` che dichiara la lunghezza del lato del cubo o il raggio della sfera.

Lo script `Region` contiene anche una lista di tuple e metodi per ottenerla o modificarla, aggiungendo o rimuovendo una tupla.

Per far fronte all'implementazione della semantica sospensiva (si veda il Paragrafo 3.4.1), vi è un campo che funge da *collezione di chiave-valore*, che tiene conto delle eventuali sospensioni delle primitive di `in` e di `rd` su una particolare tupla. Vi sono, poi, i metodi per manipolare e gestire questa collezione. Risulta di particolare importanza il metodo `UnpendPrimitive`, che prende in ingresso la stringa rappresentante il messaggio della tupla e si occupa di controllare e *risvegliare* le primitive sospese su tale messaggio. Ciò può avvenire al momento del rilascio di una tupla tramite operazione di `out`, oppure quando si entra in contatto con una nuova regione.

Infine, è definito un metodo statico per creare una regione, avente come parametri di ingresso il nome da attribuire alla regione, la forma espressa con `Primitive Type`, l'estensione e la posizione (esprimibile sia attraverso `Vector3` di Unity, sia attraverso latitudine e longitudine di Google Maps). Tale metodo crea un GameObject dalla forma e dimensione specificata (in questo modo viene creato automaticamente anche il corrispettivo *Collider*), lo colloca nella posizione indicata e ne imposta nome, `tag`, `layer` e materiale semi-trasparente.

[4]Non è possibile dotare di materiali i *Collider* e, quindi, nelle dimostrazioni pratiche risulterebbe difficile accorgersi della presenza di una regione. Per ovviare a ciò, si è scelto di creare un GameObject dalla forma ed estensione della regione, e di dotarlo dei componenti *Collider* e *Material* (definito con un grado di trasparenza).

3.3 Componente Situato

Con **componente situato**, in questa tesi, ci si riferisce a qualunque elemento posizionato nella scena di Unity, e quindi collocato in una ben precisa posizione sulla mappa di Google Maps caricata. Possiede capacità che lo rendono in grado di percepire e di essere consapevole se risiede in una regione della scena. Inoltre, può essere anche dotato di funzionalità in grado di abilitare la possibilità di navigare la scena percorrendo il reticolo stradale contenuto nella mappa.

In tal senso, sono stati creati due *script*:

- Script `AgentController`: abilita la percezione delle regioni. Il GameObject che possiede questo componente è colui che ha l'intenzione di manipolare le tuple spaziali contenute in una regione, attraverso le primitive. Lo script contiene una lista di regioni, che indica le regioni in cui risiede attualmente il componente situato (il GameObject a cui è stato aggiunto questo script come componente). La lista è aggiornata dai metodi `OnTriggerEnter` e `OnTriggerExit`. Si tratta di due metodi che vengono chiamati quando avviene la collisione con un *Collider* `trigger`. Tramite essi, se il *Collider* appartiene ad una regione, quest'ultima viene aggiunta alla lista delle regioni dello script `AgentController`.

  Infine, lo script possiede anche una lista di regioni che si spostano insieme al componente situato. In questo modo si abilitano le funzionalità della parola chiave `Me` (da sottosezione 3.4.3). Tale lista può essere modificata dai metodi `AddRegion` e `RemoveRegion`;

- Script `MyRoadLatticeCharacterController`: abilita la funzionalità di navigazione del reticolo stradale. In realtà tale compito è soddisfatto dalla classe da cui eredita, chiamata `PathingAgent` e fornita da Google Maps. Quest'ultima classe possiede, tra le altre cose, un metodo astratto chiamato `CheckPath`. Compito di `MyRoadLatticeCharacterController` è implementare tale metodo, che ha la funzione di indicare la posizione della destinazione da raggiungere. In questo script, la posizione è determinata dal *click del mouse*: viene rilevato il punto sulla mappa in cui è avvenuto il click e se ne calcola la posizione mediante `Vector3` di Unity. `PathingAgent` si occupa, poi, di trovare il nodo del reticolo più vicino a tale posizione e inizializzare la procedura di navigazione, dalla posizione corrente del componente che possiede questo script alla posizione della destinazione.

  La scelta di determinare la destinazione mediante click del mouse, quindi, non è vincolante. Si può optare per qualsiasi altro metodo desiderato. Infatti, in alcuni pattern presentati nel Capitolo 4, vengono utilizzate altre modalità.

In questa tesi si è utilizzato una sorta di avatar di gioco, che rappresenta l'agente intenzionato a manipolare le tuple spaziali. Si tratta di un *Prefab* dalla forma di un `PrimitiveType`, avente un proprio *Collider* e un proprio *Rigidbody* (indispensabili per poter utilizzare `OnTriggerEnter` e `OnTriggerExit`), uno script `AgentController` e uno script del tipo `RoadLatticeController` (lo specifico `RoadLatticeController` utilizzato varia a seconda del pattern di coordinazione considerato).

3.4 Primitive Spatial Tuples

In questa sezione si illustrano le primitive che intendono riprodurre le specifiche illustrate nel modello *Spatial Tuples*. In [16] vengono definite tre operazioni:

- `out(t@r)`: emette la tupla `t` nella regione `r`;
- `in(t@r)`: recupera, cancellando, la tupla `t` nella regione `r`;
- `rd(t@r)`: legge, senza cancellare, la tupla `t` nella regione `r`.

Oltre a queste dichiarazioni dirette, vengono fornite anche operazioni *indirette*, mediante le parole chiave `here` e `me`, o attraverso il nome di una regione o di un componente situato.

La sintassi e le specifiche di funzionamento delle primitive proposte in questa tesi hanno l'obiettivo di mantenere il più possibile la struttura presentata in [16].

Viene fornita una implementazione delle primitive sia con *semantica sospensiva*, sia con *semantica predicativa*. Le primitive sospensive si mettono in attesa del ritrovamento della tupla nell'area di ricerca definita. Pertanto, non prevedono fallimento. Le primitive predicative non sono soggette alla sospensione e quindi è previsto anche uno scenario di fallimento. Se la tupla richiesta non è presente nell'area di ricerca, la primitiva fallisce.

Le primitive sono contenute nella classe statica chiamata `Primitives`, che si presenta quindi come una collezione di metodi statici.

3.4.1 Semantica di sospensione in Unity

Sia l'operazione di `in` che l'operazione di `rd` operano mediante una semantica di sospensione: ricercano una certa tupla in una regione e, se la tupla non è disponibile, si sospendono finché la ricerca ha successo.

In Unity non c'è parallelismo, tutto il *Game Loop* principale viene svolto da un unico thread, che ha il compito di eseguire le funzionalità del motore di gioco, e di chiamare tutti gli script che estendono `MonoBehaviour`. Questo loop avviene

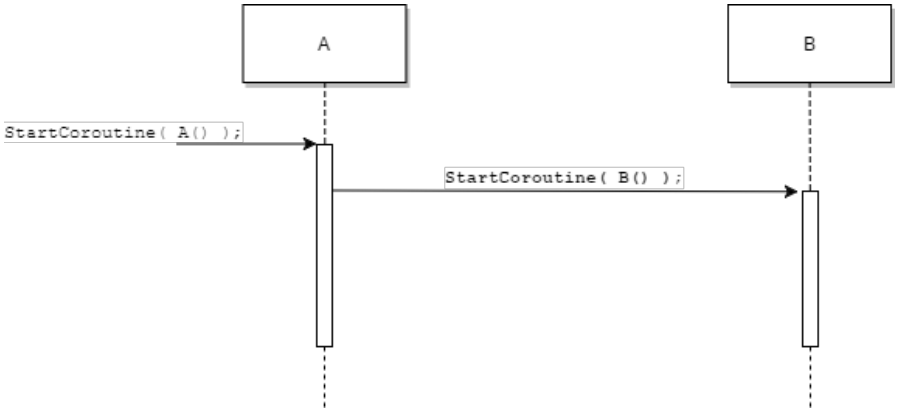


Figura 3.1: Coroutine asincrone e concorrenti

ad ogni *frame* del gioco. Tutte le operazioni e gli script, quindi, devono accadere e concludere entro ogni singolo aggiornamento del frame.

Per svolgere operazioni computazionalmente onerose o che si debbano svolgere per più frame, si utilizza il meccanismo delle *Coroutine*. L'esecuzione del codice viene sospesa in precisi punti, il controllo viene restituito a Unity e l'esecuzione viene ripresa successivamente, da dove si era interrotta. Di default, ciò accade al frame successivo, ma è possibile riprenderla in situazioni diverse mediante l'uso di istruzioni dedicate, che derivano dalla classe `YieldInstruction`.

Le Coroutine rappresentano essenzialmente un utilizzo furbo degli *iteratori*. Hanno come tipo di ritorno `IEnumerator` e includono da qualche parte nel loro corpo una dichiarazione `yield return`. Quest'ultimo è il punto in cui l'iterazione si ferma e riprende al frame successivo (se viene usato `yield return null`), chiamando `MoveNext` dell'iteratore. Esistono anche altri tipi di ritorno possibile, in modo da far ripartire l'esecuzione in momenti diversi. Ad esempio, `yield return WaitForSeconds` riprende l'iterazione dopo aver aspettato il numero di secondi dato in ingresso alla funzione `WaitForSeconds`. Una Coroutine viene fatta partire attraverso la funzione `StartCoroutine`.

Il meccanismo delle Coroutine è ben radicato in Unity e rappresenta un costrutto molto espressivo. Ad esempio, è possibile introdurre una sorta di sincronizzazione. È bene sottolineare ancora una volta come non ci sia parallelismo. Il thread principale di Unity si occupa di bloccare e riprendere l'esecuzione delle varie Coroutine, che, però, vivono tutte sullo stesso thread.

Coroutine Asincrone. Facendo partire una Coroutine all'interno dell'altra, si ottiene la loro esecuzione in modo concorrente. Le due Coroutine sono del tutto indipendenti fra di loro (Figura 3.1).

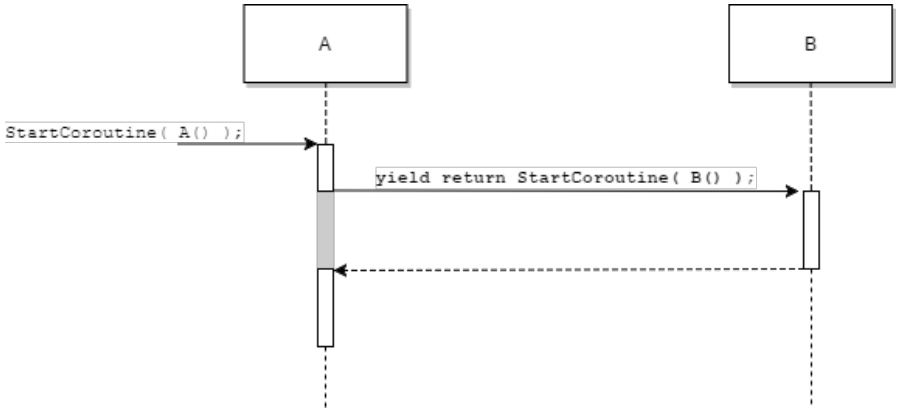


Figura 3.2: Coroutine sincrone

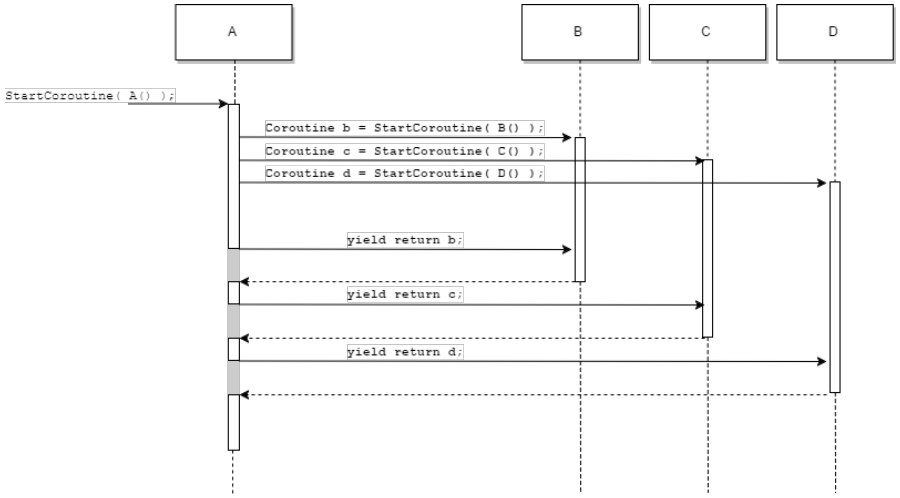


Figura 3.3: Sincronizzazione di Coroutine

Coroutine Sincrone. All'interno di una Coroutine è possibile attendere la fine dell'esecuzione di un'altra Coroutine. Si ottiene questo usando `yield return StartCoroutine` (Figura 3.2).

Sincronizzazione di Coroutine. Mettendo insieme chiamate sincrone e asincrone, è possibile ottenere operazioni di sincronizzazione di Coroutine. Si fanno partire Coroutine in modo concorrente mediante `StartCoroutine` e si attende la fine dell'esecuzione mediante `yield return` (Figura 3.3).

Implementazione del meccanismo di sospensione

Per simulare la semantica sospensiva delle primitive *Spatial Tuples*, è stato utilizzato il meccanismo delle *Coroutine*.

Se la tupla desiderata non è presente nell'area di ricerca, l'operazione viene sospesa, in attesa del suo completamento. La sospensione viene fatta attraverso la classe **C#** chiamata **TaskCompletionSource<TResult>**.[5] Essa incarna, allo stesso tempo, la creazione di un `Task` (si veda Paragrafo 3.4.1) – senza tuttavia affidargli un compito da svolgere, cioè senza un *delegato* – e la definizione del suo completamento. `TaskCompletionSource` ha al suo interno un campo rappresentate il `Task` e dei metodi per definirne lo *stato* (completato, cancellato o fallito). Si tratta quindi della rappresentazione di un valore ancora non definito, una *promessa* di un risultato futuro, che verrà compiuta attraverso il metodo `SetResult(TResult)`.

La sospensione avviene proprio in base allo stato del `TaskCompletionSource`: finché non è completato, la Coroutine esegue una `yield return`. Nel momento in cui lo stato risulterà completato, l'operazione della primitiva verrà ripresa (Listato 3.1).

Listato 3.1: Metodo che rappresenta l'operazione `in` con semantica sospensiva

```
1  public static IEnumerator SuspensiveIn(string tupleMessage,
       Vector3 center, PrimitiveType shape, float dimension, System.
       Action<Region> result = null)
2  {
3      TaskCompletionSource<Region> task = new TaskCompletionSource<
           Region>();
4      TaskCompletionSource<Region> nextTask = null;
5      Region suspensiveInReg = Region.CreateRegion("InSuspensive",
           center, dimension, shape);
6      Region found = Primitives.In(tupleMessage, suspensiveInReg,
           task);
7      while (found == null){
8          while (!task.Task.IsCompleted){
9              yield return null;
10         }
11         nextTask = new TaskCompletionSource<Region>();
12         found = Primitives.In(tupleMessage, suspensiveInReg,
               nextTask, task);
13     }
14     GameObject.Destroy(suspensiveInReg.gameObject);
15     if (result != null)
16         result(found);
17 }
```

[5] https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcompletionsource-1?view=net-5.0

La segnalazione del completamento dello stato di `TaskCompletionSource`, e quindi la chiamata del metodo `SetResult`, avviene quando viene effettuata una operazione di `out` della tupla interessata. A tal proposito, nello script `Region` è stato introdotto un campo che tiene traccia delle operazioni che sono sospese in attesa di una determinata tupla all'interno della regione a cui lo script è attaccato.

La classe **C#** chiamata **Dictionary<TKey,TValue>**[6] funge da collezione di chiave-valore. Si tratta di una *tabella hash* in cui ogni *chiave* è associata a un determinato *valore*. Il `Dictionary` dello script `Region` prende come chiave il messaggio di una tupla e associa ad essa, come valore della chiave, la *lista* dei `TaskCompletionSource` che sono in attesa della tupla in questione.

Lo script `Region` offre un metodo, chiamato `UnpendPrimitive`, che prende in ingresso la stringa rappresentante il messaggio della tupla. Questo metodo si occupa di controllare se nel `Dictionary`, suo e di ogni altra regione con cui interseca nello spazio, è presente la chiave relativa al messaggio. Se sì, controlla la lista di `TaskCompletionSource` ad essa associata, ne estrae uno e ne setta il risultato. Il risultato contiene la regione in cui è presente la tupla e quindi il `Tresult` ottenuto attraverso `SetResult(TResult)` rappresenta l'istanza a cui si riferisce lo script `Region`.

Il metodo `UnpendPrimitive` (Listato 3.2) viene chiamato in due momenti distinti:

1. quando viene effettuata una operazione di `out` nella regione in questione;

2. quando la regione entra in contatto con un'altra regione, e quindi risulta intersecare nello spazio con essa. Ciò può avvenire quando viene creata una nuova regione oppure quando una regione si sposta (operazioni `Me` delle primitive). Il contatto è segnalato dalla chiamata del metodo `OnTriggerEnter`.

Quando le operazioni di `in` e di `out` non hanno successo, viene creata una regione, corrispondente all'area di ricerca della primitiva, e viene inserito, all'interno del suo `Dictionary`, un `TaskCompletionSource`, che quindi è la rappresentazione dell'operazione sospesa e del suo risultato *futuro*.

Quando una primitiva si risveglia, in seguito alla chiamata `SetResult` che avviene nel metodo `UnpendPrimitive` della regione, non c'è garanzia che la tupla sulla quale avveniva l'attesa sia ancora presente nella regione. Infatti, fra il tempo del risveglio e la ricerca della tupla, potrebbe esserci l'esecuzione di una `in` da parte di un differente componente situato. Al risveglio, quindi, occorre effettuare nuovamente la ricerca della tupla, per controllare che sia effettivamente disponibile. Nel caso non lo fosse, si procede nuovamente all'inserimento del

[6] `https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-5.0`

Listato 3.2: Metodo che gestisce il risveglio delle primitive in attesa, tramite `SetResult`

```
public void UnpendPrimitive(string tupleMessage) {
    List<TaskCompletionSource<Region>> tasks;
    foreach (Region reg in this.overlappingRegions) {
            if (reg.pendingPrimitives.TryGetValue(tupleMessage,
                out tasks)){
                if(!tasks[0].Task.IsCompleted)
                    tasks[0].SetResult(this);
                tasks.RemoveAt(0);
                if (tasks.Count == 0)
                    reg.pendingPrimitives.Remove(tupleMessage);
            }
    }
}
```

`TaskCompletionSource` all'interno del `Dictionary` della regione relativa all'area di ricerca.

È bene sottolineare che in questo scenario di re-inserimento in seguito al risveglio, deve essere creato un nuovo `TaskCompletionSource`. Non si può riutilizzare quello precedente perché è già stato segnalato completato in seguito alla chiamata `SetResult`. Gli stati di completamento non si possono annullare o resettare. Di conseguenza, all'inserimento di `TaskCompletionSource` nuovi, quelli vecchi vengono eliminati dalla lista del `Dictionary` (vedi operazione di `in` nella Sezione 3.4.4).

La classe `TaskCompletionSource` è *thread-safe*. La classe `Dictionary` ha una versione *thread-safe*: `ConcurrentDictionary`. L'implementazione adottata da questa versione del progetto è, quindi, adatta anche ad una eventuale estensione futura in ottica *multi-threaded*.

Come ultimo passo, quando l'operazione di `in` e di `rd` hanno successo, vengono eliminate le regioni relative all'area di ricerca (che erano state create in conseguenza della sospensione), in quanto non servono più.

Quando si vuole utilizzare una primitiva con semantica sospensiva, il chiamante dell'operazione attende la fine dell'esecuzione mediante una `yield return StartCoroutine`.

Alternative future

Non si esclude la possibilità in versioni future del progetto di provare altre modalità per implementare la semantica sospensiva. A tal proposito, si vede la necessità di introdurre altri due concetti, illustrati qui di seguito: TAP e DOTS.

TAP. Nella versione 5 di **C#** è stato introdotto un nuovo approccio semplificato alla *programmazione asincrona*, chiamato **Task Asynchronous Programming** (TAP), portando il supporto asincrono alle versioni 4.5 o successive del framework .NET (al momento attuale, Unity supporta la versione 4.5). La struttura logica di tale approccio è molto simile al codice scritto in modo sincrono, facilitando quindi la scrittura da parte dello sviluppatore e lasciando tutto il lavoro al compilatore. Per utilizzare TAP è necessario far uso delle parole chiave `async` e `await`. Il passaggio da codice *sincrono* a codice *asincrono* è, quindi, molto semplice: si aggiunge `async` prima del tipo di ritorno di un metodo, e si mette `await` nei punti del corpo del codice in cui si vuole attendere l'operazione (Listato 3.3 e Listato 3.4).

Listato 3.3: Codice sincrono

```
1  private void btnClk
2  (object sender, EventArgs
       args)
3  {
4      textBox.Text =
5          "Running..";
6      var result =
7          foo.getValue();
8      textBox.Text =
9          "Result is " +
               result;
10 }
```

⇒

Listato 3.4: Codice asincrono

```
11 private async void
       btnClkAsync
12 (object sender, EventArgs
       args)
13 {
14     textBox.Text =
15         "Running..";
16     var result =
17         await foo.getValue
               ();
18     textBox.Text =
19         "Result is " +
               result;
20 }
```

Tramite metodi asincroni si evita di bloccare il thread principale di Unity in operazioni onerose in termini di tempo, permettendo di continuare il *Game Loop* senza intoppi.

Il tipo di ritorno di un metodo asincrono (un metodo contrassegnato da `async`) può essere: (*i*) `Task<TResult>` se il metodo ritorna un operando di tipo `TResult`, (*ii*) `Task` se ritorna senza operando, (*iii*) `void` se si sta scrivendo un event-handler asincrono (un cosiddetto *fire-and-forget*: non ritornando un Task, non si può conoscere lo stato dell'operazione e non si può arrestarlo o attenderlo), (*iv*) qualsiasi altro tipo che possiede il metodo `GetAwaiter`(da **C#** 7 in poi).

Nei punti in cui il codice è sospeso in attesa del completamento di una operazione asincrona, il controllo di esecuzione ritorna al chiamante. Il metodo sospeso ritorna un `Task`, che rappresenta il lavoro in corso, con la promessa di produrre un effettivo valore di ritorno nel momento in cui il lavoro è

completato. Al completamento, il `Task` viene marcato come completato e il risultato viene immagazzinato in esso. Un `Task` quindi incapsula lo stato del processo asincrono e il risultato finale di tale processo (o l'eccezione lanciata se l'operazione non ha avuto successo).[7]

Ne consegue che, i metodi asincroni sono operazioni non-bloccanti. Un'espressione `await` in un metodo asincrono non blocca il thread corrente mentre il `Task` che si sta aspettando è in esecuzione. L'espressione marca il resto del metodo come una *continuazione* e il controllo ritorna al chiamante.

Internamente, nel compilatore, avviene la creazione di una **macchina a stati**. Il codice viene suddiviso in blocchi, uno per ogni presenza della parola `await`. Ogni `Task` di cui si vuole attendere il completamento diventa un campo della macchina a stati, e il metodo `Start` si iscrive come continuazione di ciascuno di questi `Task` (metodo `Task.ContinueWith`). Quando raggiunge il completamento, si cambia il valore del campo rappresentante lo stato della macchina, e poi si chiama `Start`, che ha il compito di controllare in quale stato della macchina si è attualmente. La macchina a stati compie il passo successivo ed esegue il corrispondente blocco di codice. Quando tutto si conclude, la macchina imposta il risultato nel valore di ritorno del `Task`, e cambia lo stato di quest'ultimo in *completato*.

Le parole chiave `async` e `await` non causano la creazione di thread addizionali. Metodi asincroni non richiedono multi-threading perché questi metodi non eseguono su un loro thread specifico. Il metodo esegue nel *contesto di sincronizzazione corrente* e usa il tempo sul thread solo quando è attivo. Se si vuole spostare il lavoro computazionalmente oneroso su un thread secondario, si può usare `Task.Run`.

Allo stato attuale, Unity sconsiglia ancora di usare `async` e `await`, pur supportandoli, per un discorso di *thread safety* e contesto di sincronizzazione.[8]

Ai fini di questo progetto, non dovrebbero esserci problematiche. Tuttavia non si ritiene ci siano grandi vantaggi in più per operare questa soluzione, dato che il controllo interno per le riprese delle operazioni dovrebbe avvenire comunque a ogni frame (come per una `yield return`). In ogni caso, la sostituzione Coroutine-TAP è molto facile ed immediata: si sostituiscono i punti del codice in cui è presente una `yield return` con la parola `await`, e

[7] Differenza fra `Task` e `TaskCompletionSource` (utilizzato in questa versione del progetto): `Task` gode di vita propria e svolge l'attività che si dichiara debba svolgere, attraverso un *delegato*; `TaskCompletionSource` rappresenta solo la creazione di un `Task` – senza avviarlo e senza *delegato* – e il suo completamento. Non gode di vita propria, non svolge nessun compito. L'unica cosa che si può fare è settarne il risultato nel momento che si desidera.

[8] `https://docs.unity3d.com/2021.2/Documentation/Manual/overview-of-dot-net-in-unity.html`

al posto del tipo di ritorno `IEnumerator` si usa `Task`. Il resto della logica rimane invariata.

DOTS. Nel 2018, Unity ha presentato una roadmap che porterà a rivoluzionare dalle fondamenta l'intero motore di gioco. **Unity Data-Oriented Tech Stack** (DOTS) segna, infatti, un cambio fondamentale nella direzione dell'architettura di Unity. Conseguenza di ciò è un differente approccio nel considerare il codice e i dati, non più *object-oriented* ma **data-oriented**. Inoltre, DOTS permetterà di incrementare le performance dei progetti Unity sfruttando il vantaggio dei processori multi-core per *parallelizzare* il processo dei dati.

DOTS è formato dai seguenti elementi:

1. **Entity Component System** (ECS): è il framework che consiste di produrre codice usando un approccio *data-oriented*;
2. **C# Job System**: offre un modo semplice per generare codice multi-thread ed eliminando *race conditions*;
3. **Compilatore Burst**: compilatore ottimizzato per questa nuova struttura, in grado di generare codice nativo rapido e ottimizzato;
4. **Native Containers**: strutture dati di ECS che offrono controllo sulla memoria.

La porta di accesso fondamentale per usare DOTS è rappresentato dall'*Entity Component System*. ECS è un modo per strutturare e scrivere codice che permette di separare l'informazione (ad esempio, il comportamento) dai dati. Offre modalità per definire come organizzare i dati in memoria e in che modo debbano essere recuperati dalla CPU, migliorando di conseguenza la velocità e l'efficienza di accesso ai dati dalla memoria. L'architettura ECS è strutturata in: (*i*) **entità**, rappresenta l'identificativo che punta ai dati, (*ii*) **componente**, sono i contenitori di dati (sotto forma di strutture dati), (*iii*) **sistema**, racchiude il comportamento e la logica, mediante manipolazione e trasformazione dei dati (Figura 3.4).[9]

Nell'approccio standard object-oriented di Unity il processo di sviluppo consiste in: (*i*) creare un GameObject, (*ii*) aggiungere ad esso dei componenti, (*iii*) realizzare script `MonoBehaviour` per cambiare proprietà e comportamenti di questi componenti. A run-time, il GameObject dipende dalle referenze dei componenti. Quando uno script `MonoBehaviour` cerca i dati di un componente, questi sono sparpagliati nella memoria e, di conseguenza,

[9] tratto da `https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/ecs_core.html`

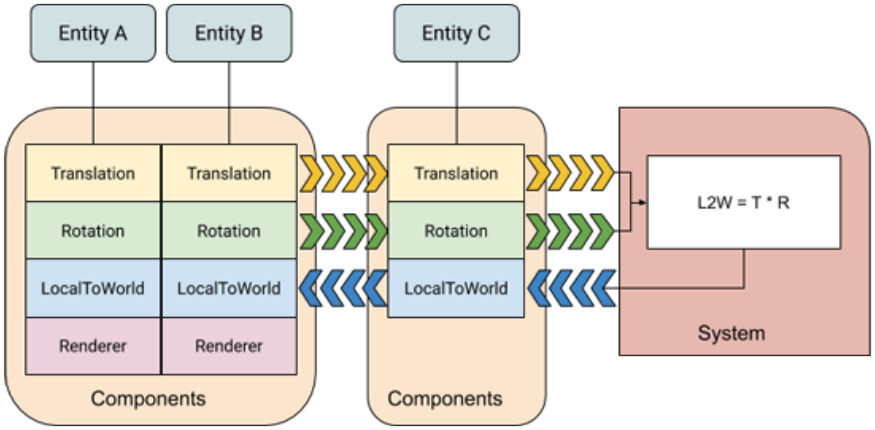


Figura 3.4: Diagramma che illustra in che modo operano le parti costituenti un'architettura ECS

viene richiesto diverso tempo per accedervi. Nell'approccio object-oriented, il codice è strutturato secondo la costruzione di entità (oggetti, costruiti come classi), la loro struttura (i dati che un oggetto possiede) e le definizioni di cosa fanno e di come interagiscono con altre entità. Questo riflette abbastanza naturalmente il modo in cui costruiamo le cose nel mondo reale. Tuttavia, dal punto di vista della strutturazione delle informazioni che gli oggetti contengono, questo approccio aggiunge livelli di astrazione ai dati. In questo modo, l'organizzazione dei dati può risultare frammentata e l'accesso può risultare rallentato.

In contrapposizione a ciò, in un approccio *data-oriented*, bisogna considerare tutto come **dati**, e non come oggetti. In questo modo si assicura che ogni dato è facilmente accessibile e senza alcuna restrizione dovuta alle classi degli oggetti. Il processo di sviluppo data-oriented consiste nell'identificare e organizzare i dati che coinvolgono i compiti più comuni che si vogliono implementare. Organizzare il codice attorno ai dati e al flusso di dati, significa che l'accesso ad essi a run-time può risultare molto più efficiente rispetto a recuperarli tramite molteplici classi di oggetti (Figura 3.5).[10]

Al momento, DOTS è ancora in fase di preview. I cambiamenti alla struttura e alle API sono frequenti e il processo di raffinamento è in corso. Allo stato attuale, quindi, Unity non garantisce ancora un prodotto stabile e pronto ad un utilizzo professionale. Inoltre, non tutte le caratteristiche di Unity sono ancora compatibili. Ciò non toglie che DOTS rappresenta il futuro del motore di gioco.

[10] tratto da `https://learn.unity.com/tutorial/what-is-dots-and-why-is-it-important`

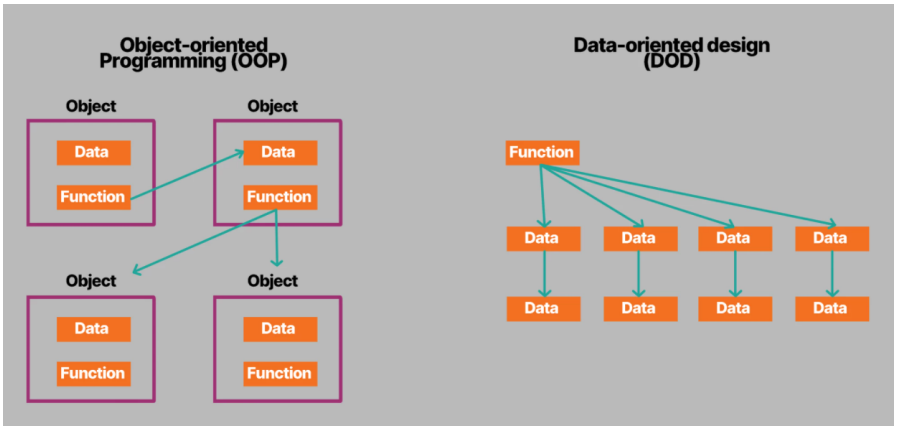


Figura 3.5: Differenza fra objected-oriented programming e data-oriented design

È bene sottolineare che al momento è possibile anche un approccio ibrido, con parti con `MonoBehaviour` e parti con DOTS. Inoltre, il Job System risulta in una fase avanzata del rilascio. Potrebbe essere interessante tentare di riprodurre il lavoro fatto in questa tesi utilizzando il Job System e scoprire la fattibilità del processo e gli eventuali vantaggi/svantaggi.

3.4.2 Geocoding API

Con *Geocoding* si intende il processo di convertire indirizzi in coordinate geografiche (espresse mediante latitudine e longitudine).

Le API fornite da Google Maps Platform offrono un modo diretto di accedere a questa funzionalità attraverso una richiesta HTTP, strutturata in questo modo

```
https://maps.googleapis.com/maps/api/geocode/outputFormat?parameters
```

dove `outputFormat` può avere uno dei seguenti valori: (*i*) `json` o (*ii*) `xml`.

Alcuni parametri sono opzionali, altri sono obbligatori, e vengono separati dal carattere "`&`".

Tra i parametri obbligatori figurano: (*i*) indirizzo e (*ii*) API key. Se si vuole inserire gli spazi all'interno della formattazione dell'indirizzo, questi vanno scritti con il relativo codice URL: "`%20`".

La risposta in formato JSON contiene due elementi alla radice: (*i*) status, che definisce i metadati della richiesta, (*ii*) results, che è formato da un array di informazioni relativi all'indirizzo geocodificato.

I risultati dell'operazione di *Geocoding* vengono posizionati nell'array JSON chiamato `results`. Esso contiene i seguenti campi:

- `types[]`: array che indica il tipo del risultato ritornato. Contiene un insieme di zero o più tag identificativi del tipo di caratteristica ritornato. Per esempio, può indicare una località, una strada, un'entità politica, un paese, un quartiere, e così via.

- `formatted_address`: stringa contenente l'indirizzo in forma leggibile da un umano. Spesso questo indirizzo coincide con l'*indirizzo postale*. L'indirizzo formattato è composto da più `address_components`. Per esempio, "via Zamboni 33, Bologna, Italia" è composto da: (*i*) "via Zamboni"(la strada), (*ii*) "33"(il numero civico), (*iii*) "Bologna"(la città) e (*iv*) "Italia"(lo Stato).

- `adress_components[]`: array contente i distinti componenti applicabili all'indirizzo. Ogni componente contiene tipicamente i seguenti campi:

  - `types[]`: array indicante il tipo dell'`address_component`(sono gli stessi tipi illustrati precedentemente);
  - `long_name`: il nome completo del componente, così come ritornato dal *Geocoder*;
  - `short_name`: forma testuale abbreviata del nome del componente. Per esempio, "Italia" potrebbe avere la forma abbreviata "IT".

- `postcode_localities[]`: array che denota tutte le località contenute in un codice postale, nel caso quest'ultimo contenesse molteplici località.

- `geometry`: contiene le seguenti informazioni:

  - `location`: contiene il valore di longitudine e latitudine. Per la maggior parte delle operazioni, questo è il campo più importante;
  - `location_type`: memorizza dati aggiuntivi riguardo allo specifico luogo (ad esempio, se il luogo trovato rappresenta precisamente quello cercato oppure è solamente una approssimazione o una interpolazione);
  - `viewport`: contiene i valori raccomandati per visualizzare i risultati sulla mappa tramite una certa visuale. Contiene due valori di latitudine e longitudine che definiscono i punti `southwest` e `northeast` del rettangolo di visualizzazione;

- `plus_code`: si tratta di un modo alternativo per rappresentare un luogo nello spazio.[11]

[11] per maggiori informazioni consultare la pagina `https://maps.google.com/pluscodes/`

Listato 3.5: Classe che svolge l'operazione di *Geocoding* in Unity

```
public static IEnumerator GetLatlngFromAddress(string _address,
    System.Action<LatLng> result){
     string URL = "https://maps.googleapis.com/maps/api/geocode/
         json?address=" + _address + "&key=" + apiKey;
     UnityWebRequest www = UnityWebRequest.Get(URL);
     yield return www.SendWebRequest();
     if (www.isNetworkError || www.isHttpError)
          Debug.Log(www.error);
     if (string.IsNullOrEmpty(www.error)){
          MyAddressResult testResult = new MyAddressResult();
          testResult = JsonUtility.FromJson<MyAddressResult>(www.
              downloadHandler.text);
          if (testResult.status == "OK"){
               if (testResult.results.Length > 0){
                 ..........
                 //accesso ai campi
                 .........
               }
          }
     }  else
          Debug.Log("Error: " + www.error);
}
```

- `partial_match`: indica che il *Geocoder* non è riuscito a trovare una corrispondenza esatta con la richiesta formulata, ma solo una parziale. Ad esempio, questo può succedere se la strada non esiste nella località in cui si sta cercando, oppure esistono due o più corrispondenze. Sono inclusi in questo campo anche richieste contenenti errori ortografici.

- `place_id`: identificativo univoco che può essere utilizzato in altre API Google.

In Unity, tutto questo si traspone in una richiesta Web tramite la classe `UnityEngine.Networking.UnityWebRequest`, dando come URL la richiesta HTTPS vista in 3.4.2. La richiesta avviene sotto forma di `yield return` per non bloccare il thread nell'attesa. Ottenuta la risposta in formato JSON, si procede al parsing utilizzando il metodo `UnityEngine.JsonUtility.FromJson`, che fornisce i campi del *Geocoding* alla classe `MyAddressResult`. Fatto ciò, si può accedere ai campi di interesse (Listato 3.5). È bene notare che le Coroutine non offrono la possibilità di avere un risultato di ritorno. Per ovviare a questo, in ingresso al metodo c'è una callback (`Action<LatLng>`) che accetta come ingresso i valori che si vogliono ritornare dalla chiamata di *Geocoding* (in questo caso, latitudine e longitudine).

3.4.3 Operazione di out

La primitiva **out** ha lo scopo di collocare una tupla in una regione dello spazio. Ne consegue che, *tupla* e *regione* fungono da parametri in ingresso dell'operazione.

- **modalità esplicita**: viene esplicitata la *forma* e l'*estensione* della regione in cui effettuare l'operazione.
- **modalità implicita**: non viene specificata la forma o l'estensione della regione. La posizione del rilascio della tupla viene definita solamente attraverso l'uso di `here`, `me` o specificando il nome di un componente situato.

La posizione in cui collocare la tupla può essere ricavata nei seguenti modi:

1. tramite *latitudine* e *longitudine*;
2. tramite `here`, indicante la posizione corrente del chiamante dell'operazione;
3. tramite `me`, indicante la posizione corrente del chiamante dell'operazione, aggiornata ad ogni suo spostamento nello spazio;
4. tramite il *nome* di un altro componente situato o di una regione;
5. tramite *indirizzo stradale*.

Operazione di base

```
out(t@r)
```

Si tratta della forma base da cui derivano tutte le altre operazioni di `out` (Listato 3.6). Prende in ingresso una tupla `t` e una regione `r` e si occupa di posizionare `t` in `r`. Dal punto di vista di Unity, la tupla diventa un GameObject figlio della regione. Infine, chiama il metodo `UnpendPrimitive` della regione, in modo da risvegliare eventuali primitive in attesa della tupla appena rilasciata.

Latitudine e Longitudine

```
out(t@region(coord(lat,long),name,shape(dimension)))
```

Crea una `region` di nome `name`, con il centro posizionato nelle coordinate geografiche date da `lat,long`, di forma `shape` e avente un'estensione pari a `dimension`. Dopodiché, viene chiamata l'operazione base di `out` in tale regione.

Listato 3.6: Operazione base di `out` in Unity

```
public static bool Out(Tuple tuple, Region region)
{
    region.AddTuple(tuple);
    tuple.gameObject.transform.position = region.transform.
       position;
    tuple.gameObject.transform.SetParent(region.transform);
    tuple.gameObject.SetActive(true);
    region.UnpendPrimitive(tuple.Message);
    Debug.Log($"Operation out in {region.name}: successful!");
    return true;
}
```

Posizione Worldspace di Unity

`out(t@region(Vector3,name,shape(dimension)))`

In questo caso il centro della regione non è dato dalle coordinate geografiche bensì dalla posizione espressa in valori cartesiani rappresentanti il mondo di Unity.

Here

`out(t@region(here,shape(dimension)))`

Si posiziona la tupla nella posizione corrente del componente situato che ha chiamato l'operazione di `out`, con le seguenti specifiche:

- Se il componente è già collocato in una regione, e l'estensione della regione combacia con quella di `dimension`, allora viene effettuata l'operazione base di `out` su tale regione;
- Se il componente situato è già collocato in una regione, ma la sua estensione non corrisponde a `dimension`, ne viene creata una nuova, avente come centro la posizione del componente, come forma `shape` e come estensione `dimension`. Dopodiché, viene effettuata l'operazione base di `out` su questa regione;
- Se il componente situato non è collocato in alcuna regione, ne viene creata una nuova, con le medesime specifiche dichiarate nel punto precedente.

È possibile effettuare l'operazione di `here` anche attraverso la forma implicita

`out(t@here)`

In questo modo: (*i*) se l'agente non è collocato in alcuna regione, ne viene creata una di default, avente il centro nella posizione corrente del componente situato, forma `PrimitiveType.Cube` ed estensione 5 metri; (*ii*) se il componente situato è già collocato in una regione, viene effettuata l'operazione base di `out` su tale regione, senza alcun controllo di estensione.

Me

```
out(t@region(me,shape(dimension)))
```

Si posiziona la tupla nella posizione corrente del componente situato che ha chiamato l'operazione di `out`. In questo caso la tupla si sposta insieme al componente. Le specifiche di funzionamento sono le seguenti:

- Se il componente situato ha già una regione che lo segue, e l'estensione della regione è pari a `dimension`, la tupla viene collocata in tale regione;
- Se l'estensione non combacia con `dimension`, oppure il componente non possiede alcuna regione che lo segue, allora viene creata una nuova regione, avente come centro la posizione attuale del componente, di forma `shape` e di estensione `dimension`. La regione creata viene inserita nella lista, contenuta nello script `AgentController`, delle regioni che seguono il componente, mediante il metodo `AddRegion`.

Dal punto di vista di Unity, per far in modo che la regione segua il componente, bisogna renderlo un GameObject figlio del componente.

È possibile effettuare l'operazione di `me` anche attraverso la forma implicita

```
out(t@me)
```

In questo modo: (*i*) se il componente situato possiede già una regione che lo segue, la tupla viene collocata in tale regione; (*ii*) altrimenti, viene creata una nuova regione di default, di forma `PrimitiveType.Cube` ed estensione 5 metri e la si aggiunge alla lista delle regioni che seguono il componente. Non viene fatto alcun controllo di estensione.

Nome della regione o del componente situato

```
out(t@region(name,shape(dimension)))
```

Si ricerca all'interno della scena di Unity il GameObject di nome `name`:

- Se viene trovato ed è una regione (ha il `tag` region), si effettua un controllo dell'estensione: (*i*) se è uguale a `dimension`, allora si colloca la tupla in tale regione; (*ii*) altrimenti, si crea una nuova regione, avente il centro nella stessa posizione, forma `shape` ed estensione `dimension` e si inserisce la tuple al suo interno;

- Se viene trovato ma non è una regione, viene creata una nuova regione, avente il centro nella posizione del GameObject, forma `shape` ed estensione `dimension`, e la si rende un figlio del GameObject(in modo da farla spostare insieme all'oggetto `name`). Infine, si colloca la tupla in tale regione;

- Se non viene trovato nessun GameObject, la tupla viene eliminata e persa definitivamente.

È possibile effettuare l'operazione anche attraverso la forma implicita

```
out(t@name)
```

In questo modo, si cerca il GameObject di nome `name` e:

- Se viene trovato ed è una regione, la tupla viene collocata al suo interno, senza controlli aggiuntivi sull'estensione;

- Se viene trovato ma non è una regione, ne viene creata una di default, di forma `PrimitiveType.Cube` ed estensione 5 metri, e la si rende figlio dell'oggetto trovato;

- Se non viene trovato, la tupla viene persa definitivamente.

Indirizzo

```
out(t@region(address,shape(dimension)))
```

Il metodo è scritto sotto forma di *Coroutine*. Si sfruttano le Geocoding API fornite da Google Maps e abilitate dalla API key, per convertire l'indirizzo in latitudine e longitudine. Si attende la fine dell'operazione mediante una `yield return`. Se esiste una regione avente il nome che corrisponde all'indirizzo, si colloca la tupla in tale regione. Altrimenti, si crea una nuova regione, avente centro nella posizione determinata dalla latitudine e longitudine dell'indirizzo, forma `shape` ed estensione `dimension`.

3.4.4 Operazione di in

La primitiva **in** ha lo scopo di recuperare una tupla `t` in una regione `r`. L'operazione si sospende fino al ritrovamento della tupla. Tale sospensione è implementata in Unity tramite *Coroutine* (come spiegato in 3.4.1).

Esattamente come per l'operazione di inserimento, la ricerca di una tupla può essere compiuta in posizioni geografiche definite nei seguenti modi:

1. tramite punto definito da *latitudine* e *longitudine*;
2. tramite `here`, indicante la posizione corrente del chiamante dell'operazione;
3. tramite `me`, indicante la posizione corrente del chiamante dell'operazione, aggiornata ad ogni suo spostamento nello spazio;
4. tramite il *nome* di un altro componente situato o di una regione;
5. tramite *indirizzo stradale*.

Analogamente, l'operazione può essere definita attraverso due modalità:

- **modalità esplicita**: viene esplicitata la *forma* e l'*estensione* della regione in cui effettuare l'operazione.
- **modalità implicita**: non viene specificata la forma o l'estensione della regione. La posizione dell'area di ricerca della tupla viene definita solamente attraverso l'uso di `here`, `me` o specificando il nome di un componente situato.

Per completezza, le operazioni di `in` e di `rd` sono state implementate con una *versione sospensiva* e una *versione non-sospensiva*. La prima, nel caso la tupla non sia stata trovata, crea una regione corrispondente all'area di ricerca e si sospende con il `TaskCompletionSource`. La seconda si limita a cercare la tupla nell'area di ricerca, senza creare una regione.

Operazione di base

```
in(t@area)
```

Si tratta della forma base da cui derivano tutte le altre operazioni di `in`. Prende in ingresso una stringa rappresentante il messaggio della tupla, un'area di ricerca data dalla regione, e il `TaskCompletionSource`, nel caso in cui la `in` non abbia successo. Il metodo compie una ricerca della tupla desiderata. La ricerca viene effettuata nell'area data in ingresso e in tutte le regioni con cui interseca nello spazio. Se la tupla viene trovata, si restituisce la regione di ritrovamento. Altrimenti, si inserisce il `TaskCompletionSource` nella lista delle primitive sospese della regione in questione, avendo cura di rimuovere eventualmente quelli vecchi (Listato 3.7).

Listato 3.7: Operazione base di `in` in Unity

```
public static Region In(string tupleMessage, Region region,
    TaskCompletionSource<Region> task, TaskCompletionSource<Region>
     oldTask)
{
     Region result = null;
     Region r;
     for (int i = 0; i < region.overlappingRegions.Count; i++){
          r = region.overlappingRegions[i];
          if (InSearch(tupleMessage, r)){
               Debug.Log("In() operation successful!");
               result = region.overlappingRegions[i];
               break;
          }
     }
     if (result == null){
          region.AddPendingPrimitive(tupleMessage, task);
          if (oldTask != null)
               region.RemovePendingPrimitive(tupleMessage, oldTask);
     }
     return result;
}
```

Latitudine e Longitudine

`in(t@area(coord(lat,long),shape(dimension)))`

Si utilizza il metodo `Coords.FromLatLngToVector3` fornito da Google Maps per convertire la coordinata geografica data in ingresso, rappresentata dalla latitudine e longitudine, in un punto del *Worldspace* di Unity. Dopodiché, viene chiamata l'operazione base di `in` nell'area avente come centro il punto trovato, forma `shape` ed estensione `dimension`.

Here

`in(t@area(here,shape(dimension)))`

Si ricerca la tupla nella posizione corrente del componente situato che ha chiamato l'operazione di `in`, controllando se esso è già collocato all'interno di una regione. Viene chiamato il metodo base dando come valori in ingresso i relativi parametri, come spiegato di seguito:

- Se il chiamante si trova già all'interno di una regione, la tupla viene ricercata in un'area di forma `shape`, avente il centro coincidente con il centro

della regione, ed estensione pari alla *somma* dell'estensione della regione e `dimension`;

- Se il chiamante non è collocato in alcuna regione, viene considerato come centro dell'area di ricerca la posizione corrente del chiamante, la forma è `shape` e l'estensione è pari a `dimension`.

È possibile effettuare l'operazione di `here` anche attraverso la forma implicita

```
in(t@here)
```

In questo modo: (*i*) se l'agente non è collocato in alcuna regione, la ricerca avviene in un'area di default, avente il centro nella posizione corrente del componente situato, forma `PrimitiveType.Sphere` ed estensione 15 metri; (*ii*) se il componente situato è già collocato in una regione, viene effettuata l'operazione base di `in` nell'area di ricerca avente come centro, forma ed estensione quelli della regione.

Me

```
in(t@area(me,shape(dimension)))
```

Si ricerca la tupla nella posizione corrente del componente situato che ha chiamato l'operazione di `in`, aggiornata di volta in volta ad ogni spostamento del componente. Si ottiene ciò rendendo la regione un GameObject figlio del componente.

Viene chiamato il metodo base dando in ingresso il messaggio della tupla e l'area di ricerca con centro la posizione corrente del componente, la forma `shape` e l'estensione `dimension`, avendo cura di controllare che l'area non esista già.

È possibile effettuare l'operazione di `me` anche attraverso la forma implicita

```
in(t@me)
```

In questo modo, viene chiamato il metodo di base dando come parametri di ingresso il messaggio della tupla e l'area di ricerca avente come centro la posizione attuale del chiamante, e una forma e una estensione di default, rispettivamente `PrimitiveType.Sphere` e 15 metri, dopo aver controllato se tale tipo di area non esista già fra le regioni che seguono il componente.

Nome della regione o del componente situato

```
in(t@area(name,shape(dimension)))
```

Si ricerca all'interno della scena di Unity il GameObject di nome `name`:

- Se non esiste, l'operazione è interrotta, segnalando l'assenza dell'oggetto;

- Se esiste, si chiama il metodo base dando in ingresso il messaggio della tupla e l'area di ricerca avente come centro la posizione corrente dell'oggetto, la forma `shape`, e l'estensione data dalla *somma* della dimensione dell'oggetto e `dimension`.

È possibile effettuare l'operazione anche attraverso la forma implicita

```
in(t@name)
```

In questo modo, si cerca il GameObject di nome `name` e:

- Se non viene trovato, l'operazione è interrotta, segnalando l'assenza dell'oggetto;
- Se viene trovato, si chiama il metodo base dando in ingresso il messaggio della tupla e l'area di ricerca avente come centro la posizione corrente dell'oggetto, la forma di default `PrimitiveType.Sphere` e l'estensione data dalla dimensione dell'oggetto.

Indirizzo

```
in(t@area(address,shape(dimension)))
```

Il metodo è scritto sotto forma di *Coroutine*. Si sfruttano le Geocoding API fornite da Google Maps e abilitate dalla API key, per convertire l'indirizzo in latitudine e longitudine. Si attende la fine dell'operazione mediante una `yield return`. Dopodiché, si chiama l'operazione definita in 3.4.4.

3.4.5 Operazione di rd

Le operazioni di `rd` hanno sintassi e comportamento analoghi a quelli definiti per le operazioni di `in`. L'unica concreta differenza è che non avviene nessuna rimozione e distruzione quando si trova una corrispondenza fra la tupla da ricercare e le tuple contenute in una regione.

Data la ridondanza, si ritiene superfluo riportare nuovamente tutte le dichiarazioni e le specifiche di funzionamento.

Capitolo 4

Pattern di Coordinazione Spatial Tuples in Unity

In questo capito viene mostrata l'efficacia del modello *Spatial Tuples* nel supportare la coordinazione di entità computazionali le cui attività coinvolgono aspetti legati alla spazialità e alla *posizione* e *movimento* nello spazio.

In [16] viene illustrata una collezione semplice, ma allo stesso tempo rilevante ed esaustiva, di **pattern di coordinazione**, volti a risolvere problemi tipici e generali.

Di seguito, vengono illustrati i pattern e i modi in cui sono stati implementati in Unity, sfruttando le primitive descritte in 3.4.

È bene specificare che gli script di setup della scena relativa ai pattern sono componenti da aggiungere al GameObject vuoto che contiene lo script *Maps Service*, come dichiarato in 2.3.2.

I componenti situati che agiscono all'interno delle scene sono *Prefabs* a cui sono stati aggiunti e abilitati gli script relativi (*i*) al comportamento da avere nel pattern preso in considerazione, (*ii*) al modo in cui devono navigare il reticolo stradale della mappa caricata.

4.1 Situated Communication

Con **Situated Communication**, comunicazione situata, si vuole intendere la possibilità per un componente situato di lasciare un messaggio in uno specifico punto dello spazio o in una regione. Tale messaggio può essere ricevuto da qualsiasi componente (anche *sconosciuto a priori*, o creato successivamente) che è interessato a messaggi posizionati nel luogo considerato. Si tratta di comunicazione in cui c'è *disaccoppiamento* fra messaggio, destinatario e mittente.

Il mittente rilascia il messaggio tramite `out(msg@r)` e il destinatario ricerca il messaggio tramite `in(msg@r')`, con `r` e `r'` che rappresentano regioni con spazi in comune.

La comunicazione potrebbe avvenire anche non appena un componente situato entra nella regione in cui è collocato il destinatario, tramite una `out(msg@situated Component)` – o `out(msg@me)` – e una `in(msg@here)`.

In Unity, si è implementato questo pattern attraverso tre script (si veda la Figura 4.1 per la resa visiva):

- **SituatedCommunicationSetup**: è lo script che prepara la scena di Unity. Dopo aver atteso il caricamento della mappa, ha il compito di caricare su di essa i componenti situati e abilitarne gli script necessari, posizionare la telecamera per visualizzare la scena, caricare la regione in cui avviene la comunicazione situata;

- **SituatedCommunicationAgentMe**: è lo script che effettua l'operazione di `out@regioneComunicazione` e l'operazione di `out@me`;

- **SituatedCommunicationAgentOther**: è lo script che effettua l'operazione di `in@regioneComunicazione` e l'operazione di `in@here`.

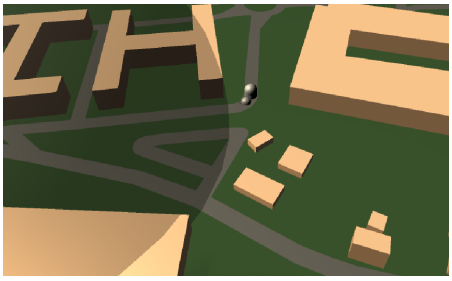
4.2 Situated Knowledge Sharing

Con **Situated Knowledge Sharing**, condivisione di informazione situata, si intende una generalizzazione della comunicazione situata, in cui le tuple spaziali sono utilizzate per mandare *indirettamente* un messaggio a tutti gli agenti collocati in una certa regione. Il messaggio viene rappresentato attraverso una tupla spaziale associata a tale regione.

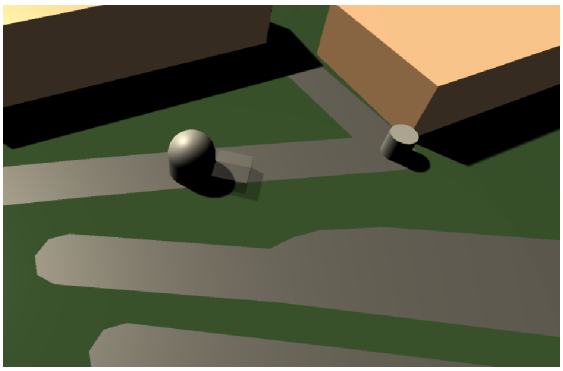
Per esempio, se, in una operazione di soccorso, l'agente `control_room` deve avvertire tutti i soccorritori riguardo la presenza di feriti in qualche regione, è sufficiente compiere una operazione di `out(warning(injured)@region)`. In questo modo, i soccorritori collocati in tale regione, che sono in attesa di messaggi di avvertimento attraverso `rd(warning(injured)@me)`, riceverebbero immediatamente la notizia.

Questo pattern illustra come le tuple spaziali possono rappresentare informazioni e conoscenze *condivise* riguardo il mondo fisico, nella regione dello spazio in cui tale conoscenza è emersa.

(a) Operazione di `out@region` e `out@me` (notare le due tuple)



(b) Operazione di `in@region` (notare la distruzione della tupla nella regione)



(c) Operazione di `in@here` (notare la distruzione della tupla e la regione vuota che segue il componente)

Figura 4.1: Resa visiva del pattern *Situated Communication*

(a) Regione di soccorso

(b) Log in cui si mostra come le operazioni di `rd` abbiano successo dopo la `out`

Figura 4.2: Resa visiva del pattern *Situated Knowledge Sharing*

In Unity, si è implementato questo pattern attraverso tre script (si veda la Figura 4.2 per la resa visiva):

- **SituatedKnowledgeSharingSetup**: è lo script che prepara la scena di Unity. Dopo aver atteso il caricamento della mappa, ha il compito di caricare su di essa i componenti situati e abilitarne gli script necessari, posizionare la telecamera per visualizzare la scena, caricare la regione in cui avviene la condivisione di informazione situata;

- **SituatedKnowledgeSharingRoom**: è lo script che effettua l'operazione di `out` nella regione di soccorso. L'oggetto a cui è attaccato questo script è collocato in un punto estremo della mappa, al di fuori della regione di controllo, per simulare la stanza di controllo delle operazioni di soccorso;

- **SituatedKnowledgeSharingRescuer**: è lo script che effettua l'operazione di `rd` nella regione di soccorso, in attesa di messaggi di allerta.

4.3 Awareness

Con **Awareness**, consapevolezza, si indica un caso specifico di condivisione di conoscenza, prendendo in considerazione la posizione di un componente situato che può muoversi nello spazio. In questo scenario, un componente potrebbe avere

la necessità di essere consapevole se e quando altri componenti arrivano in una regione.

Per esempio, si consideri l'ipotesi in cui viene stabilita una certa regione d'incontro nella quale un numero di componenti mobili devono incontrarsi. Un ulteriore componente, invece, è preposto al controllo dell'arrivo dei componenti mobili. L'arrivo è marcato dall'operazione `out(hereIAm@here` e il controllo avviene mediante una operazione `in(hereIAm@meetingPlace)`, per ogni componente atteso.

In Unity, si è implementato questo pattern attraverso tre script (si veda la Figura 4.3 per la resa visiva):

- **AwarenessSetup**: è lo script che prepara la scena di Unity. Dopo aver atteso il caricamento della mappa, ha il compito di caricare su di essa i componenti situati e abilitarne gli script necessari, posizionare la telecamera per visualizzare la scena, caricare la regione in cui avviene l'incontro dei componenti;

- **AwarenessMeetingControl**: è lo script relativo al componente che controlla l'arrivo nella regione di incontro. Compie una `in(hereIAm@meeting Place)` per ogni componente atteso. L'oggetto a cui è attaccato questo script è collocato in un punto estremo della mappa, al di fuori della regione di incontro, per simulare la stanza di controllo delle operazioni di incontro;

- **AwarenessMeeters**: è lo script che effettua una `out(hereIAm@here)`. L'operazione viene effettuata nel momento in cui il componente a cui è attaccato questo script entra nella regione di incontro, in seguito alla chiamata del metodo `OnTriggerEnter`.

Variante con Sincronizzazione Spaziale. Questa variante considera la possibilità di far iniziare un compito ad un componente situato `A` non appena un altro componente `B` arriva in uno specifico posto. In questo caso, è stato creato uno script aggiuntivo, `SynchronizationAwarenessMeeter`, che si sposta verso la regione di incontro solo alla lettura di una particolare tupla, mediante una operazione di `rd`. Lo script `AwarenessMeetingControl` compie il rilascio di questa tupla particolare nella regione `meetingPlace`, nel momento in cui giunge nella regione di incontro un certo numero di componenti mobili.

(a) Arrivo nella regione e operazione di `out` (cerchiato in rosso la tupla)

(b) Arrivo del SynchronizationAwarenessMeeter (cerchiato in rosso) dopo che l'operazione di `rd` ha successo

Figura 4.3: Resa visiva del pattern *Awareness*

4.4 Breadcrumbs

La condivisione delle informazioni potrebbe coinvolgere anche il *percorso* di un componente situato che si sposta nello spazio. Mentre si muove, il componente lascia una traccia di tuple spaziali (**breadcrumbs**, le briciole), ognuna delle quali contenente informazioni sull'ordine di rilascio, per esempio un *contatore* che si incrementa.

Si supponga lo scenario in cui un componente avente un `Contatore C` rilascia periodicamente una tupla spaziale attraverso `out(wasHere(C)@here)`, incrementando ogni volta `C`. In questo modo, il percorso del componente può essere tracciato e ricostruito osservando la distribuzione spaziale delle tuple contenenti il messaggio "wasHere", mediante continue operazioni di `in`.

In Unity, si è implementato questo pattern attraverso tre script (si veda la Figura 4.4 per la resa visiva):

- **BreadcrumbsSetup**: è lo script che prepara la scena di Unity. Dopo aver atteso il caricamento della mappa, ha il compito di caricare su di essa i componenti situati e abilitarne gli script necessari e posizionare la telecamera per visualizzare la scena;
- **BreadcrumbsMover**: è lo script relativo al componente che rilascia le briciole. Possiede un contatore che viene incrementato ad ogni operazione di `out(wasHere(C)@here)`;
- **BreadcrumbsChaser**: è lo script che effettua la ricerca delle briciole. Viene effettuata di volta in volta una `in(wasHere(C)@me)`, incrementando la zona

(a) Spargimento delle briciole nello spazio

(b) Raccoglimento delle briciole e inseguimento

Figura 4.4: Resa visiva del pattern *Breadcrumbs*

di ricerca se non si trova niente. Al ritrovamento della tupla, l'inseguitore raggiunge il punto in cui era posizionata e procede alla prossima ricerca, incrementando il `Contatore C`.

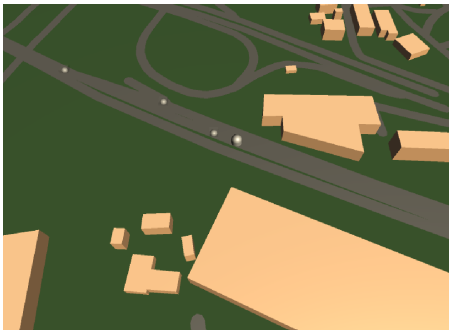
4.5 Sincronizzazione Spaziale

Con **Sincronizzazione Spaziale** si intende l'atto di sincronizzare le azioni delle entità in base alle informazioni spaziali contenute nel mondo fisico arricchito con tuple spaziali. In tal senso, un agente `A` compie un'azione `T` non appena un'azione differente `T'` avvenuta in una regione `r` è stata completata da un altro agente. L'agente `A` non ha bisogno di conoscere chi sta compiendo l'azione `T'` nella regione `r`, e chiunque stia compiendo l'azione `T'` non ha bisogno di conoscere chi è interessato al risultato di `T'`.

Si supponga lo scenario in cui un componente situato ha il compito di consegnare dei pacchi in qualche luogo all'interno della sua area di consegna. Non appena compie l'azione di consegna, rilascia una tupla spaziale nell'indirizzo del luogo, tramite `out(package_delievered@address)`.

Un altro componente, invece, ha il compito di raccogliere i pacchi che sono stati consegnati, non appena essi siano disponibili nella sua area di raccolta. Per segnalare il suo interesse a raccogliere i pacchi consegnati nella sua area di assegnazione, effettua una `in(package_delivered@areaDiRaccolta)`.

In Unity, si è implementato questo pattern attraverso tre script (si veda la Figura 4.5 per la resa visiva):



(a) Consegna dei pacchi

(b) Raccolta dei pacchi all'interno dell'area di assegnazione

Figura 4.5: Resa visiva del pattern *Sincronizzazione Spaziale*

- **PackageSetup**: è lo script che prepara la scena di Unity. Dopo aver atteso il caricamento della mappa, ha il compito di caricare su di essa i componenti situati e abilitarne gli script necessari e posizionare la telecamera per visualizzare la scena;

- **PackageDelivery**: è lo script relativo al rilascio della tupla spaziale indicante la consegna del pacco. Il rilascio avviene mediante una operazione di `out(packageDelivery@here)`;

- **PackagePicker**: è lo script che crea la regione che funge da area di raccolta dei pacchi consegnati al suo interno. Dopodiché, si sospende in attesa della consegna di pacchi, tramite `in(packageDelivery@areaDiRaccolta)`. Una volta trovata la tupla corrispondente, si sposta verso di essa in modo da effettuare la raccolta del pacco.

4.6 Mutua Esclusione Spaziale

Tramite il modello di coordinazione *Spatial Tuples*, è possibile implementare una **Mutua Esclusione Spaziale**, in modo da regolare l'accesso degli agenti in una regione fisica dello spazio.

Per esempio, per poter consentire l'accesso di un solo agente alla volta in una regione chiamata `mutex region`, viene utilizzata una tupla spaziale, che funge da `lock`. In questo modo, per poter accedere alla regione, si ha bisogno di possedere il `lock`, che viene rilasciato una volta che si è al di fuori della regione. Il `lock` si ottiene mediante `in(lock@mutexRegion)` e si rilascia mediante `out(lock@mutexRegion)`.

(a) Un componente alla volta può entrare (cerchio blu), gli altri attendono (cerchi rossi)

(b) Una volta che un componente è uscito (cerchio blu), l'altro può entrare (cerchio rosso)

Figura 4.6: Resa visiva del pattern *Mutua Esclusione Spaziale*

In Unity, si è implementato questo pattern attraverso due script (si veda la Figura 4.6 per la resa visiva):

- **MutualExclusionSetup**: è lo script che prepara la scena di Unity. Dopo aver atteso il caricamento della mappa, ha il compito di caricare su di essa i componenti situati e abilitarne gli script necessari, posizionare la telecamera per visualizzare la scena, creare la regione `mutex region` e, infine, collocare all'interno della regione la tupla spaziale che funge da `lock`, mediante `out(lock@mutexRegion)`;
- **MutualExclusionAgent**: è lo script relativo al componente che cerca di entrare nella `mutex region`, attraverso una `in(lock@mutexRegion)`. Una volta ottenuto il `lock`, entra nella regione, vi rimane all'interno per un certo quantitativo di tempo, dopodiché esce dalla regione e rilascia il `lock` mediante una `out(lock@mutexRegion)`.

4.7 Spatial Dining Philosophers

Si tratta di una variante del problema dei **Dining Philosophers**, adattato per mostrare la coordinazione spaziale abilitata da *Spatial Tuples*. Dato che non ci si vuole concentrare sul problema del *deadlock*, si utilizza una soluzione basata sul rilascio di *ticket*, biglietti di ingresso. In una tavola di `N` filosofi, ci sono `N-1` biglietti.

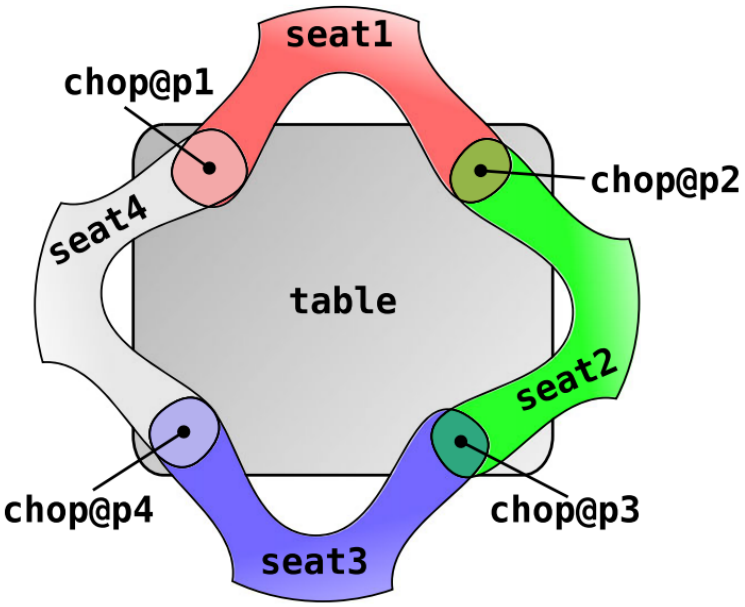


Figura 4.7: La tavola del problema *Spatial Dining Philosopher*, tratta da [16]

Si supponga un problema costituito da `N=4` filosofi. I `4` filosofi vogliono sedersi a tavola e mangiare. Per poter accedere alla tavola occorre avere uno dei `3` biglietti di ingresso disponibili. I `ticket` vengono posizionati nella tavola mediante `out(ticket@table)`. In aggiunta, ci sono `4` bacchette, rappresentate da tuple spaziali, chiamate `chop`, collocate ciascuna in un punto dello spazio (p_1, p_2, p_3, p_4). Questi punti appartengono alla regione rappresentante la tavola e ciascuno di essi è condiviso da due posti a sedere adiacenti, definiti dalle regioni chiamate `seat` (Figura 4.7).

Se un filosofo vuole mangiare nel posto $seat_1$, per prima cosa deve ottenere un ticket attraverso `in(ticket@table)`, poi deve ottenere le bacchette, effettuando due volte $in(chop@seat_1)$. In questo modo, il filosofo otterrebbe le due tuple spaziali $chop@p_1$ e $chop@p_2$, in quanto p_1 e p_2 hanno punti in comune con la regione $seat_1$, e quindi c'è corrispondenza spaziale.

Una volta ottenute le due bacchette, il filosofo può mangiare. Dopodiché, rilascia le bacchette mediante $out(chop@p_1)$ e $out(chop@p_2)$. Per ultima cosa, rilascia il ticket mediante `out(ticket@table)`.

In Unity, si è implementato questo pattern attraverso tre script (si veda la Figura 4.8 per la resa visiva):

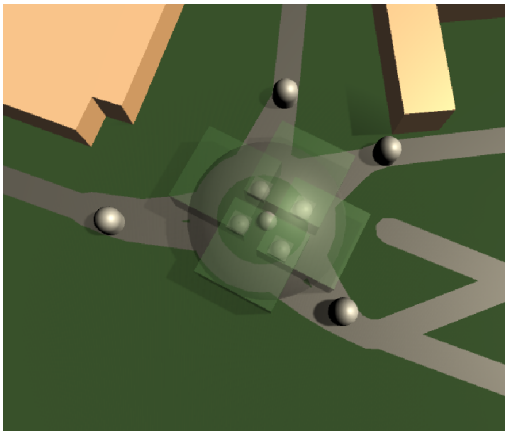
- **SpatialDiningSetup**: è lo script che prepara la scena di Unity. Crea la regione rappresentante la *tavola*. Crea le regioni che fungono da *posti a sedere*. Crea i *punti* in cui collocare le bacchette. Crea e posiziona i *filosofi*. Posiziona le bacchette nei punti, mediante `out(chop@p`$_n$`)`. Posiziona `3` ticket nella tavola mediante `out(ticket@table)`. Infine, posiziona la telecamera e abilita i vari script dei componenti creati;

- **SpatialDiningSeat**: è lo script relativo al *posto a sedere*. Si occupa di segnalare se un posto è libero oppure occupato. Se il GameObject a cui è assegnato questo script è di colore verde, il posto è libero. Se è di colore rosso, il posto è occupato;

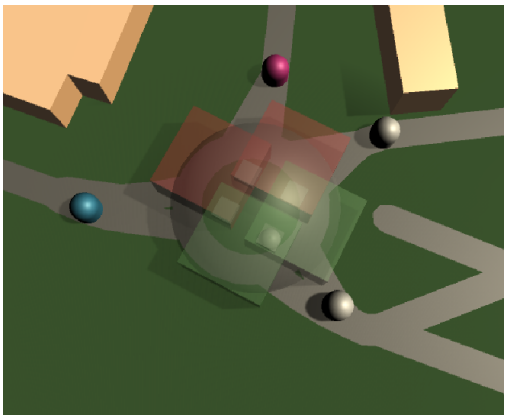
- **SpatialDiningPhilosopher**: è lo script relativo ai *filosofi*. Cerca il ticket mediante `in(ticket@table)`. Dopodiché, cerca le regioni contenute nella regione della tavola, in modo da ottenere le regioni relative ai posti a sedere. Controlla se c'è un posto a sedere libero. Se sì, lo occupa e cerca le bacchette mediante due `in(chop@seat`$_n$`)`.[1] Ottenute le bacchette, inizia la fase del mangiare. Una volta finita questa fase, si procede al rilascio delle bacchette nei punti p_n e $p_{(n\%4)+1}$ contenuti nella regione $seat_n$ occupata, mediante `out(chop@p`$_n$`)` e `out(chop@p`$_{(n\%4)+1}$`)`. Per ultimo, si rilascia il ticket mediante `out(ticket@table)` e si libera il posto occupato. Se il GameObject a cui è assegnato questo script è di colore viola, il filosofo ha acquisito il ticket ed è entrato nella tavola. Se il colore è turchese, sta mangiando.

Variante Filosofi Moventi. Per mostrare i vantaggi offerti dalla navigazione del reticolo stradale messo a disposizione da Google Maps, si è scelto di implementare anche una variante dei Dining Philosophers, in cui i filosofi si *muovono* fisicamente verso il posto a sedere che hanno occupato. In tal senso, lo script **SpatialMovingDiningPhilosopher** ha un comportamento analogo allo script visto in precedenza, con la differenza che una volta trovata la regione rappresentante il posto a sedere libero, il filosofo occupa tale posto muovendosi fisicamente verso la posizione della regione. Visivamente, dunque, si può osservare un continuo spostamento dei filosofi desiderosi di mangiare (Figura 4.9).

[1] L'area di ricerca $seat_n$ trova al suo interno le due regioni in cui risiedono le bacchette (nell'esempio con 4 filosofi, p_n e $p_{(n\%4)+1}$).

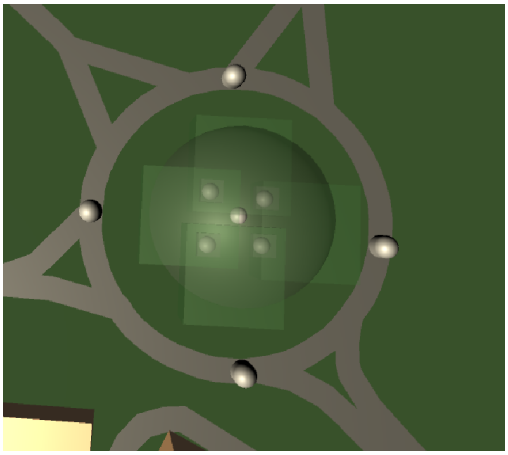


(a) Scenario iniziale della tavola



(b) Filosofi in azione

Figura 4.8: Resa visiva del pattern *Spatial Dining Philosophers*



(a) Scenario iniziale della tavola

(b) Filosofi in azione. Se non hanno il ticket, i filosofi attendono fuori dalla tavola

Figura 4.9: Resa visiva del pattern *Spatial Dining Philosophers*, variante *Filosofi Moventi*

Capitolo 5

Conclusioni e Lavoro Futuro

Il progetto sviluppato in questa tesi rappresenta una prima versione di riproduzione del modello *Spatial Tuples* all'interno di Unity, utilizzando solamente gli strumenti messi a disposizione dal motore di gioco. Sono stati forniti script e primitive facilmente utilizzabili sotto forma di API o librerie. Usando solo gli strumenti offerti dall'engine, la creazione di un plug-in di Unity risulta molto semplice e immediata, così da facilitare ancora di più l'utilizzo e la diffusione degli strumenti espressivi messi a disposizione dal progetto e donando supporto tecnologico alla visione di *Spatial Tuples*.

Si ritiene di aver soddisfatto l'obiettivo posto dalla tesi, in quanto si è riusciti a riprodurre completamente il modello di coordinazione *Spatial Tuples*. Si è data una definizione e una implementazione in Unity delle astrazioni relative alla *tupla spaziale*, che arricchisce la realtà con informazioni digitali, alla *regione*, che si estende in uno spazio fisico del mondo reale, e al *componente situato*, che svolge la manipolazione delle tuple e si muove nello spazio. Si sono definite le specifiche di funzionamento delle *primitive* dichiarate in [16], la sintassi e la loro implementazione. Infine, si è validato tutto il lavoro svolto, concretizzando i *pattern di coordinazione*.

Si è riusciti a dare forma alla visione del modello fortemente basata sulla spazialità e sul movimento dei componenti situati, sfruttando le funzionalità offerte dal plug-in di Google Maps. L'utilizzo di mappe 3D raffiguranti il mondo reale e lo sfruttamento della navigazione del reticolo stradale hanno dato opportunità di poter visualizzare in modo effettivo tutte le astrazioni definite da *Spatial Tuples*.

In quanto prima versione di riproduzione del modello di coordinazione, rimangono aperti diversi scenari per sviluppi futuri.

Si potrebbe andare oltre la rappresentazione delle tuple come semplici stringhe e cercar di riprodurre l'espressività delle variabili logiche, simulandone la costru-

zione in **C#**. In questo modo si potrebbe abilitare un pattern matching più efficace ed espressivo.

Altro aspetto relativo alle fondamenta del modello *Spatial Tuples*, è quello riguardo alla semantica sospensiva. Come spiegato nella sezione 3.4.1, si è scelto il meccanismo delle *Coroutine*. Può risultare un lavoro utile quello di pensare a modi alternativi per implementare la sospensione. Ciò comporta la ridefinizione delle specifiche di funzionamento e l'implementazione delle primitive o, eventualmente, l'intera struttura della tesi. In 3.4.1 sono stati ipotizzati scenari di esplorazioni future. In particolare, risulta interessante approfondire la fattibilità della trasposizione del progetto utilizzando il *Job System* messo a disposizione da Unity.

Proprio la natura a singolo thread di Unity, ci porta a considerare il processo di cambio radicale, fin dalle fondamenta, che sta avvenendo nel motore di gioco, migrando verso l'architettura DOTS, come illustrato in 3.4.1. Può risultare un lavoro degno di attenzione trasportare il progetto svolto in questa tesi verso i nuovi meccanismi che Unity sta introducendo alla sua architettura, volta a diventare *data-oriented* e *multi-threaded*.

Unity offre supporto nativo ai giochi *multiplayer*. Si ritiene molto interessante testare il sistema attuale in uno scenario multiplayer, in cui le regioni create vengono condivise con tutti i dispositivi appartenenti alla scena. Non si ritiene di dover affrontare grossi cambiamenti alla struttura del sistema per poter abilitare uno scenario multi-giocatore, in quanto è un aspetto ben radicalizzato in Unity e svolto in maniera semplice e ben documentata dal motore di gioco.

Questa tesi è stata testata su PC ma è facilmente esportabile per tutte le piattaforme supportate da Unity, senza dover cambiare niente alla struttura del codice. In tal senso, si può sostituire tutto il meccanismo della navigazione del reticolo stradale, che fissa il punto di destinazione mediante click del mouse, con un meccanismo che si basa sulla posizione dell'utente che sta utilizzando il progetto. Il SDK offerto da Google Maps supporta già questo scenario, è ben documentato e contiene un esempio che mostra come sfruttare tale possibilità. In questo modo, si ottiene la definitiva realizzazione di componenti collocati nello spazio e in grado di spostarsi e muoversi in esso, concetto fondante del modello presentato in *Spatial Tuples*.

Infine, possono essere considerati come ulteriori sviluppi futuri il miglioramento della piacevolezza di utilizzo e dell'usabilità, rendendo il progetto efficacemente accessibile in larga scala da utenti anche non esperti.

Bibliografia

[1] Ronald T. Azuma. A survey of augmented reality. *Presence Teleoperators Virtual Environ.*, 6(4):355–385, 1997.

[2] Alessandro Bagnoli. *Game Engines e MAS: Spatial Tuples in Unity3D.* PhD thesis.

[3] Mattia Cerbara. *Game engines and MAS: tuplespace-based interaction in Unity3D.* PhD thesis.

[4] Paolo Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys*, 28(2):300–302, June 1996.

[5] Paolo Ciancarini, Andrea Omicini, and Franco Zambonelli. Multiagent system engineering: The coordination viewpoint. In Nicholas R. Jennings and Yves Lespérance, editors, *Intelligent Agents VI. Agent Theories, Architectures, and Languages*, volume 1757 of *LNAI*, pages 250–259. Springer, 2000. 6th International Workshop (ATAL'99), Orlando, FL, USA, 15–17 July 1999. Proceedings.

[6] Giovanni Ciatto, Giovanna Di Marzo Serugendo, Maxime Louvel, Stefano Mariani, Andrea Omicini, and Franco Zambonelli. Twenty years of coordination technologies: COORDINATION contribution to the state of art. *Journal of Logical and Algebraic Methods in Programming*, 113:1–25, June 2020.

[7] Marco Fuschini. *Tecnologie ad Agenti per Piattaforme di Gaming: un caso di studio basato su JaCaMo e Unity.* PhD thesis.

[8] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[9] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.

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

[11] Andrea Omicini and Franco Zambonelli. MAS as complex systems: A view on the role of declarative approaches. In João Alexandre Leite, Andrea Omicini, Leon Sterling, and Paolo Torroni, editors, *Declarative Agent Languages and Technologies*, volume 2990 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, May 2004. 1st International Workshop (DALT 2003), Melbourne, Australia, 15 July 2003. Revised Selected and Invited Papers.

[12] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. In Marvin V. Zelkowitz, editor, *The Engineering of Large Systems*, volume 46 of *Advances in Computers*, pages 329–400. Academic Press, 1998.

[13] Luca Pascucci. *Synapsis - Middleware per l'integrazione di Game Engine e Sistemi Multi-Agente*. PhD thesis.

[14] Nicola Poli. *Game Engines and MAS: BDI & Artifacts in Unity*. PhD thesis.

[15] Alessandro Ricci, Andrea Omicini, Mirko Viroli, Luca Gardelli, and Enrico Oliva. Cognitive stigmergy: Towards a framework based on agents and artifacts. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *Environments for MultiAgent Systems III*, volume 4389 of *Lecture Notes in Computer Science*, chapter 7, pages 124–140. Springer Berlin Heidelberg, May 2007. 3rd International Workshop (E4MAS 2006), Hakodate, Japan, 8 May 2006. Selected Revised and Invited Papers.

[16] Alessandro Ricci, Mirko Viroli, Andrea Omicini, Stefano Mariani, Angelo Croatti, and Danilo Pianini. Spatial Tuples: Augmenting physical reality with tuple spaces. In Costin Badica, Amal El Fallah Seghrouchni, Aurélie Beynier, David Camacho, Cédric Herpson, Koen Hindriks, and Paulo Novais, editors, *Intelligent Distributed Computing X. Proceedings of the 10th International Symposium on Intelligent Distributed Computing – IDC 2016, Paris, France, October 10-12 2016*, volume 678 of *Studies in Computational Intelligence*, pages 121–130. Springer, 2017.

[17] Davide Rossi, Giacomo Cabri, and Enrico Denti. Tuple-based technologies for coordination. In Andrea Omicini, Franco Zambonelli, Matthias Klusch, and Robert Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies, and Applications*, chapter 4, pages 83–109. Springer, January 2001.

[18] Peter Wegner. Coordination as constrained interaction. In *International Conference on Coordination Languages and Models*, pages 28–33. Springer, 1996.

[19] Danny Weyns, Andrea Omicini, and James J. Odell. Environment as a first class abstraction in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, February 2007. Special Issue on Environments for Multi-agent Systems.

[20] Franco Zambonelli and Andrea Omicini. Challenges and research directions in agent-oriented software engineering. *Auton. Agents Multi Agent Syst.*, 9(3):253–283, 2004.