Università degli Studi di Salerno
Dipartimento di Informatica

Tesi di Laurea Magistrale in
Informatica

Simulazioni ABM in Unity3D: l'architettura del client e la gestione dei messaggi

Relatore
Prof. Vittorio Scarano

Candidato
Lucio Grimaldi

Anno Accademico 2022-2023

Abstract

Il panorama delle simulazioni è in grado di dare un grande apporto allo studio dei fenomeni reali. I modelli computazionali studiati per simularli, sono in grado di aumentare la conoscenza che abbiamo del mondo che ci circonda. I software sviluppati ad hoc per questi obiettivi sono creati per riprodurre il fenomeno che deve essere modellato, osservato e analizzato con il fine di imparare qualcosa di nuovo. La natura articolata e complessa dei fenomeni e dei modelli, fa si che lo sforzo si sia concentrato sempre sulla tecnica di implementazione, sull'ottimizzazione e raramente sulla loro rappresentazione grafica e sulla capacità che essi hanno di trasmettere le informazioni a chi non è, per forza, un esperto del settore. Il progetto sviluppato è diviso in tre macro aree che corrispondono alla creazione di una metodologia di astrazione e prototipazione dei modelli di simulazione su motore di simulazione MASON per la prima parte, grazie al lavoro di Pietro Russo nella sua tesi "Simulazioni ABM in Unity3D: Un framework di integrazione per simulazioni MASON", ad un middleware in grado di far comunicare MASON e il motore grafico Unity 3D illustrato in questa tesi ed infine un'ultima parte riguardante l'implementazione grafica in Unity3D nella tesi di Gerardo Barone "ABM Simulations for Unity: rendering 3D and user interaction". Obiettivo di questo lavoro di tesi è quello di unire la conoscenza che proviene dall'uso delle simulazioni ad agenti, con la potenza dell'espressione resa possibile dalla grafica 3D, per rendere possibile la fruizione della conoscenza proveniente dall'uso delle simulazioni ad un pubblico più ampio.

Questa tesi è stata sviluppata in 

Indice

Capitolo 1

Introduzione

La possibilità di interagire con le ABM [1] è la principale caratteristica che è mancata nei motori di simulazione sviluppati fino ad oggi. Questo aspetto è in grado fornire un grande aiuto ai tipi di simulazioni esistenti in quanto si può massimizzare la personalizzazione della stessa anche durante la sua esecuzione, ad esempio aggiungendo o rimuovendo degli agenti, degli oggetti o semplicemente cambiando il punto di vista dell'osservatore. Il secondo aspetto molto importante al fine di aumentare la fruibilità delle ABM è l'integrazione con il comparto grafico 3D, in grado di semplificare la percercezione di ciò che il modello di simulazione vuole trasmettere, ad esempio, lo spazio di simulazione e i suoi agenti nel constesto grafico giusto, come uno stormo di uccelli che, negli attuali motori, MASON compreso, sono mostrati come semplici punti rossi in uno sterile sfondo nero. Il terzo aspetto è la collaborazione, fondamentale dal punto di vista didattico, permette a più utenti di osservare la simulazione in contemporanea, e allo stesso tempo modificarla. Questo lavoro di tesi si pone di mettere in pratica i tre aspetti citati, attraverso una struttura software in grado di controllare il server di simulazione MASON adeguatamente modificato e messo a punto dal mio collega Pietro Russo, tramite un client Unity3D nel quale sono stati implementati 2 moduli. Il primo modulo è quello responsabile dell'interazione tra Unity3D e MASON sviluppato da me e il secondo, dal collega Gerardo Barone, incentrato sulla personalizzazione grafica. Nel seguito di questo elaborato andrò ad esporre tutte le componenti del modulo da me creato, le sue caratteristiche, i problemi risolti e cosa apporta al mondo delle simulazione ad agenti.

[1] Abbreviazione di Agent Based Model, ovvero "modello basato su agenti"

Capitolo 2

Background e tecnologie

Un modello ad agenti (ABM) [26] è una simulazione nel quale un insieme di entità, chiamate agenti, che interagiscono tra loro, vengono osservate alla ricerca di fenomemi emergenti durante l'esecuzione. Le ABM sono strumenti importanti per la comprensione e l'esplorazione delle relazioni che ci sono tra gli elementi di sistemi interconnessi. Attualmente le ABM sono usate in molti campi scientifici e sociali per studiare sistemi complessi, ad esempio gli ecosistemi, i fenomeni atmosferici, le società di individui, i comportamenti in gruppo di animali e esseri umani. Le ABM sono una tipologia di modelli appertenenti all'insieme dei "microscale models" nei quali vengono studiate le azioni di una moltitudine di agenti, nel tentativo di ricreare e prevedere l'apparizione di fenomeni complessi, un esempio possono essere i modelli in grado di simulare la defluizione di grandi insiemi di persone da edifici su cui si sono abbattuti fenomeni atmosferici o naturali[20]. Un comportamento emergente è qualcosa che nasce dal livello microscopico a quello macroscopico e permette di sfruttare la natura simile che hanno gli agenti dello stesso tipo, per poter studiarne gli effetti su una scala più ampia. La nozione chiave è che semplici regole comportamentali generano comportamenti complessi e quindi avere uno strumento che è in grado di generare comportamenti complessi a partire da comportamenti semplici è essenziale per ampliare la conoscenza del fenomeno stesso. La scelta del motore di simulazione è ricaduta sulla libreria MASON [17] scritta in Java e sviluppata dalla George Mason University [9], per la comunicazione invece il protocollo MQTT [18] viene usato per lo scambio di messaggi tra server e client, fino ad arrivare al motore grafico Unity3D che visualizza le informazioni ricevute e svolge la funzione di client. Prima dell'inizio dello sviluppo di questo progetto nel settembre del 2019, nello stato dell'arte non erano presenti architetture che

avevano l'obiettivo di integrare un motore di simulazione ad agenti con un software in grado di rappresentare in modo esteticamente fedele o comunque riconducibile alla realtà, gli agenti e lo spazio in cui si muovono.

2.1 Stato dell'arte

Nello stato dell'arte attuale gli elaborati della letteratura scientifica, si sono posti l'obiettivo di migliorare le implementazioni delle ABM andando a lavorare sul singolo aspetto dei motori di simulazione, come può essere uno specifico algoritmo implementato o la loro resa grafica bidimensionale piuttosto che tridimensionale, oppure sulla trasposizione dei modelli su motori grafici preesistenti che fino a pochi anni fa non erano sufficientemente sviluppati per essere in grado di implementare facilmente i modelli più complessi. Possiamo raggruppare, dunque, gli approcci in "**Tight-Coopled**"[2] e "**Loose Cooupled**"[3].
Parlando di "**Tight-Coopled**", l'architettura del modello si incentra dunque su uno specifico software in grado di elaborare i comportamenti logici degli agenti delle ABM, in cui c'è, o è possibile implementare nativamente, anche una visualizzazione grafica in 2D oppure in 3D, senza la necessità di usare altri software esterni. Motori che hanno questo tipo di caratteristiche sono, per la maggioranza di volte, essenzialmente incentrati sullo sviluppo della parte logica, a discapito della parte grafica spesso vista come un qualcosa di "extra" rispetto all'implementazione del comportamento degli agenti, che quindi risulta in una visualizzaione molto semplice e priva di dettagli utili alla comprensione del constesto in cui gli agenti "agiscono", penalizzandone invece la fruibilità ai meno esperti. Come viene mostrato in Figura 2.1 la visualizzaione 2D del motore Netlogo[19] per il modello "prede-predatore" in cui il predatore è semplicemente un'immagine stilizzata di un lupo, la preda è un'immagine stilizzata di una pecora e il terreno su cui si muovo è rappresentato da celle di colore verde o marrone rende l'idea di come la visualizzazione sia poco dettagliata, un'altro esempio è la Figura 2.2 del motore di simulazione MASON[17] dove abbiamo uno stormo di volatili rappresentati questa volta da semplici frecce in uno sfondo bianco senza avere nemmeno un riferimento alla forma stessa dei volatili o al classico colore azzurro del cielo, rendendone la comprensione più difficile senza le

[2] "Strettamente legato", si intende una stretta correlazione tra le componenti logiche e grafiche

[3] "Debolmente legato", si intende una debole correlazione tra le componenti logiche e grafiche sviluppate

corrette informazioni su cosa il modello vuole rappresentare, peccandone dunque di intuitività.

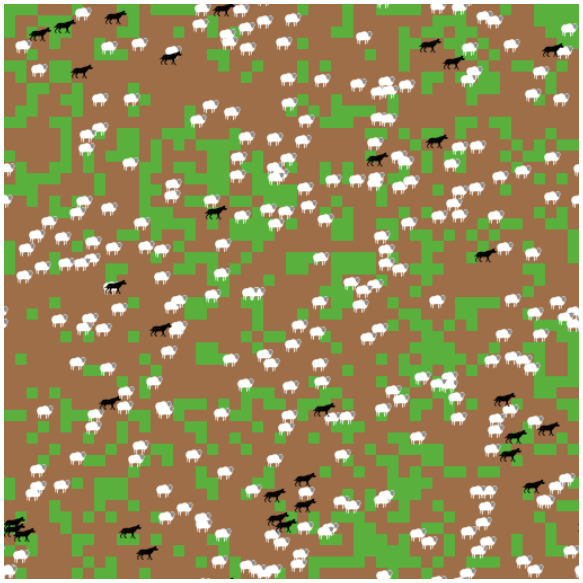

Figura 2.1: Visualizzazione 2D nativa in NetLogo del modello preda-predatore

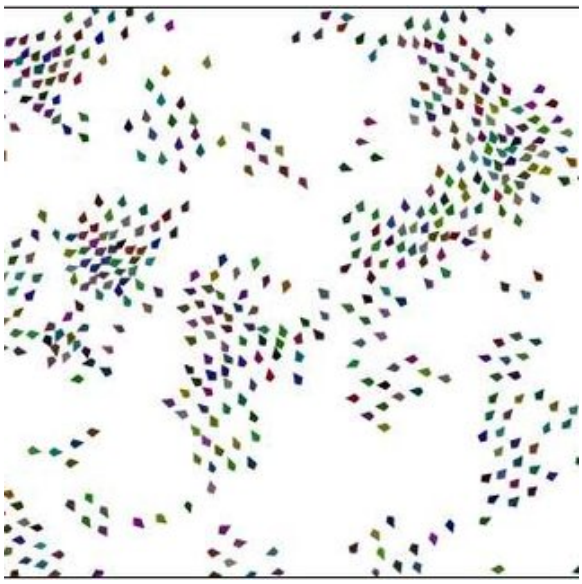

Figura 2.2: Visualizzazione 2D nativa in MASON del modello Flockers in cui viene rappresentato uno stormo di volatili

Parlando invece di "**Loose-Coopled**" intendiamo delle architetture in cui la componente logica, quindi che simula il comportamento degli agenti, è slegata o debolmente legata alla parte grafica, ciò vuol dire che è possibile andare a sostituire una delle due componenti, grafica o logica, andando ad impattare in modo minimo il numero di cambiamenti della parte rimanente o all'architettura stessa, mantenendo però lo stesso tipo di funzionamento. Questo obiettivo può essere raggiunto tramite l'uso di tecnologie o protocolli di comunicazione intermedi standardizzati, quindi implementabili in un gran numero di software, e che ci permettono di usare o di sviluppare separatamente i motori di simulazione dai software di rappresentazione grafica 2D e 3D. Un esempio è il protocollo MQTT[18] che consente agilmente lo scambio di messaggi tra le componenti software di due o più dispositivi che vedremo nelle prossime pagine più nel dettaglio. L'obiettivo, dunque, di architetture "**Loose-Coupled**" è quello di aggiungere flessibilità alle ABM, facilità di sostituzione delle componenti e possibilità di ampliarne le funzionalità ed è il tipo di architettura su cui è stato basato questo progetto e ne verrà descritta la nostra implementazione anch'essa nelle prossime pagine.
Il primo aricolo da cui questa tesi ha preso spunto e dove sono introdotte le architetture di tipo **Tight-Coupled** e **Loose-Coupled** è *Advances and Techniques for Building 3D Agent-Based Models for Urban Systems* [2], dove

viene spiegato che i modelli basati su agenti si spostano sempre più nel dominio spaziale bi e tridimensionale come punto focale dei modelli stessi, e che quindi c'è bisogno di nuovi modi per esplorare, visualizzare e comunicare tali modelli, in particolare a coloro a cui cerchiamo di trasmettere conoscenza o laddove tali modelli sono al servizio degli esperti, che tramite le ABM, studiano le caratteristiche dei problemi che vengono simulati. Questa è già stata identificata come una delle principali sfide per le ABM e dunque non riguarda solo la nozione che i buoni modelli devono sembrare corretti, ma si riferisce anche a uno degli scopi principali dei modelli basati su agenti, che è quello di trasmettere visivamente il comportamento del modello in modo chiaro e rapido. Con queste premesse, come anticipato precedentemente, si capisce come la concentrazione si pone sul miglioramento grafico al fine di trasmettere più informazioni o semplicemente dare all'utente una comprensione migliore del contesto in cui le ABM sono utilizzate. Gli autori sostengono che la natura stessa della disciplina è orientata alla teoria piuttosto che alla divulgazione dell'utente finale diversamente da quella di videogiochi e film, e quindi per questo motivo lo sviluppo e l'implementazione grafica delle ABM è da sempre passata in secondo piano rispetto alla componente teorica e concettuale alla base dei modelli sviluppati.

Per creare quindi un ambiente paragonabile al mondo reale dove lasciar muovere gli agenti, c'è bisogno di sorgenti di dati digitali geografici esaustivamente ampie e dettagliate, come ad esempio terreni in cui gli agenti possono muoversi, edifici dove "vivere" o andare a "lavorare", strade dove camminare o viaggiare. L'ultima decade ha visto la proliferazione di sorgenti di dati geografiche a scala fine, le quali sono diventate sempre più grandi e dettagliate spesso legate alle scansioni satellitari 3D per la creazione dei modelli delle città.

Questi modelli sono il risultato dell'integrazione tra software per manipolazione 3D, come ad esempio CAD[4] [6], (Figura 2.3), GIS [5][8], (Figura 2.4), e dati provenienti da tecnologie di rilevamento aeree. Cambiamenti significativi sono stati fatti nell'incrementare l'accuratezza dei modelli 3D, ad esempio, delle città, ma molti rimangono degli involucri "vuoti" senza dati socio-economici correlati che danno un senso ai modelli 3D e che permettono di analizzare il ruolo che hanno all'interno della vita di tutti i giorni, e quindi l'assenza di questi dati aggiuntivi, risulterebbe un'ostacolo alla creazione di ABM 3D.

[4]Computer-Aided Design significa progettazione con l'aiuto del calcolatore

[5]Geographic Information System

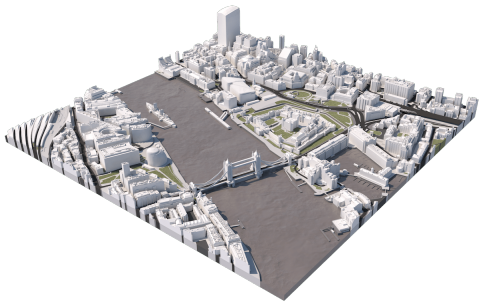

Figura 2.3: Esempio di mappa 3D creata con strumenti CAD

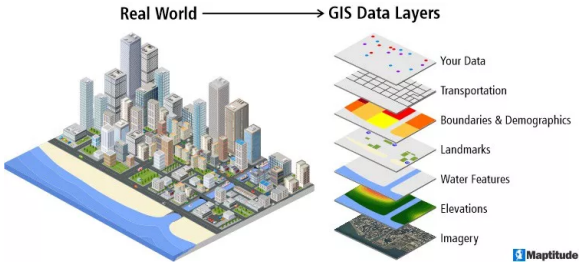

Figura 2.4: Esempio di strutture dati usate da GIS per creare mappe 3D

Sempre in questo articolo viene mostrato a pagina 55-58 un confronto di come può essere sviluppata una ABM **Tight-Coopled** interamente con 3DS Max[3] sia per la logica degli agenti sia per la parte di rappresentazione grafica, e una **Loose-Coupled** con l'utilizzo di NetLogo per la parte logica e sempre 3DS Max per la parte di visualizzazione tridimensionale. Pur non essendo di per sé un pacchetto di visualizzazione specializzato basato su agenti, 3DS Max dispone di una simulazione integrata di sistema conosciuto come "Crowd and Delegate", in Figura 2.5, che consente di animare gruppi di oggetti e personaggi 3D utilizzando un sistema di semplici regole, che rappresentano i comportamenti basilari degli agenti come ad esempio "evita", "segui", "ricerca" o insiemi di regole più complesse personalizzabili, facendo così si possono creare insiemi di agenti con comportamenti anche molto complessi. In questo caso una volta che viene definito il comportamento degli agenti e creata la superficie tridimensionale su cui farli muovere, è possibile avviare la simulazione ed osservarne i risultati nel tempo.

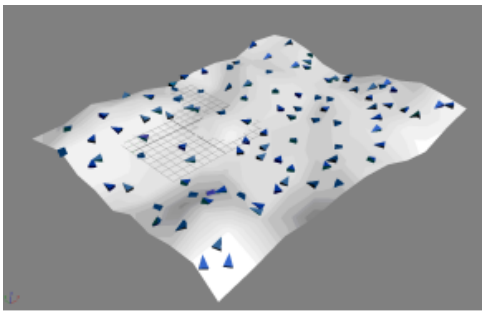

Figura 2.5: Esempio di implementazione Crowd and delegate di 3DS Max di agenti che si muovono seguendo la forma del terreno

L'esempio portato come paragone tra i due approcci è basato su una simulazione del comportamento di pedoni e veicoli. Da un lato abbiamo in Figura 2.8 la simulazione interamente su Netlogo dove si vedono tutti i parametri modificabili in quanto essendo Netlogo un motore di simulazione creato ad hoc, si hanno una maggiore personalizzazione e complessità dei modelli. Rispettivamente invece, in Figura 2.6, abbiamo la visualizzazione su 3DS Max in un ambiente tridimensionale e completamente personalizzabile seppur lasciato scarno dagli autori per replicarne le caratteristiche di quello su Netlogo, è possibile eventualmente arricchire la scena con ogni dettaglio che si voglia aggiungere.

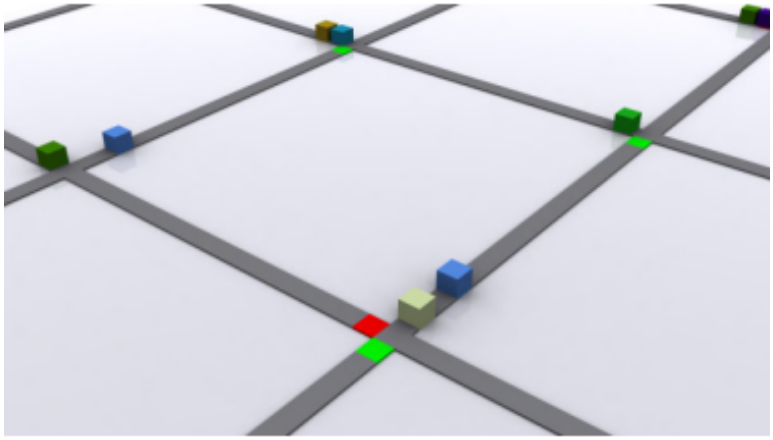

Figura 2.6: Esempio di implementazione basilare in 3DS Max di una città con veicoli e semafori

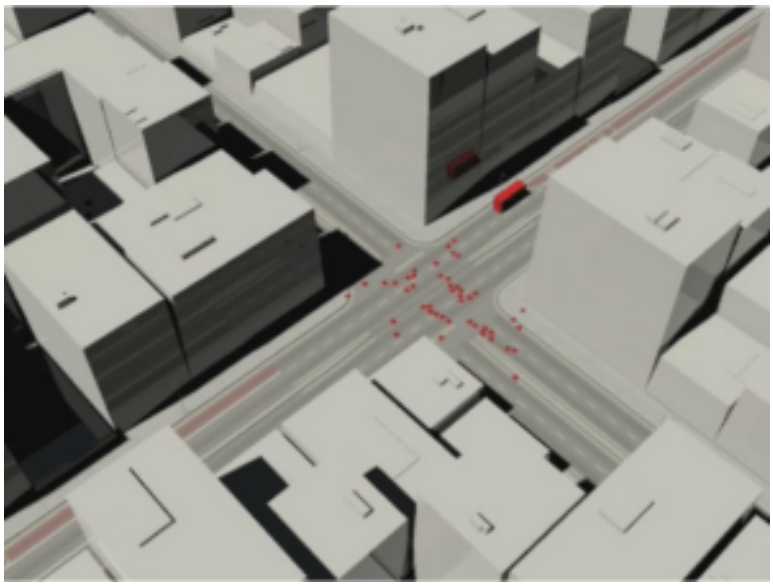

Figura 2.7: Esempio di implementazione più dettagliata in 3DS Max di una città

Contrariamente a quanto appena descritto, l'implementazione Loose-Coupled fornisce un'alternativa interessante, nel senso che possiamo creare un modello basato su agenti utilizzando una libreria specifica o utilizzare un motore di simulazione/modellazione progettato specificamente per ABM e poi visualizzare gli output del modello in un ambiente 3D (quindi la scena 3D è puramente a scopo di visualizzazione a patto che ci siano coordinate x, y e z incorporate direttamente nel modello simulato). In questo caso di esempio viene usato NetLogo, come sistema di simulazione e 3DS Max per la visualizzazione. Come esempio viene preso un semplice modello di traffico da NetLogo come mostrato in Figura 2.8, che modella il movimento delle auto su una rete stradale. Il movimento è limitato dai semafori, gli agenti si fermano al semaforo rosso e passano al verde. In modo da ottenere una rappresentazione fisica tridimensionale dell'ambiente, il movimento delle auto in NetLogo è tradotto e salvato in file di testo registrando il loro movimento ad ogni iterazione (step[6]) del modello, le coordinate delle auto, le coordinate dei tratti di strada e la posizione dei semafori vengono memorizzate per ogni step per un totale di 500 step. Questi dati vengono quindi importati in 3ds Max tramite uno script ed eseguiti i passaggi per animare e rendere la scena

[6]Uno step è un passo di simulazione in cui l'agoritmo dietro il modello di simulazione applica, su ogni agente, il comportamento previsto

come mostrato in Figura 2.6. Il processo di comunicazione tra NetLogo e 3ds Max è mostrato in Figura 2.9.
Questo approccio chiaramente ad un primo impatto può sembrare sfavorevole rispetto al Tight-Coupled, in quanto necessita di un coinvolgimento di un maggior numero di componenti software, eventuale trasfomazione dei dati in un formato compatibile ad entrambi i moduli dell'architettura e soprattutto tempo di implementazione. Se questo può esser vero è anche vero che è così possibile usufruire di tutti i benefici descritti nelle pagine precedenti come semplicità di sostituzione delle componenti, miglioramento della parte grafica e la conseguente semplicità divulgativa del modello.
Dettagli aggiuntivi sulle implementazioni Tight-Coupled e Loose-Coupled sono disponibili nell'articolo *Advances and Techniques for Building 3D Agent-Based Models for Urban Systems* [2].

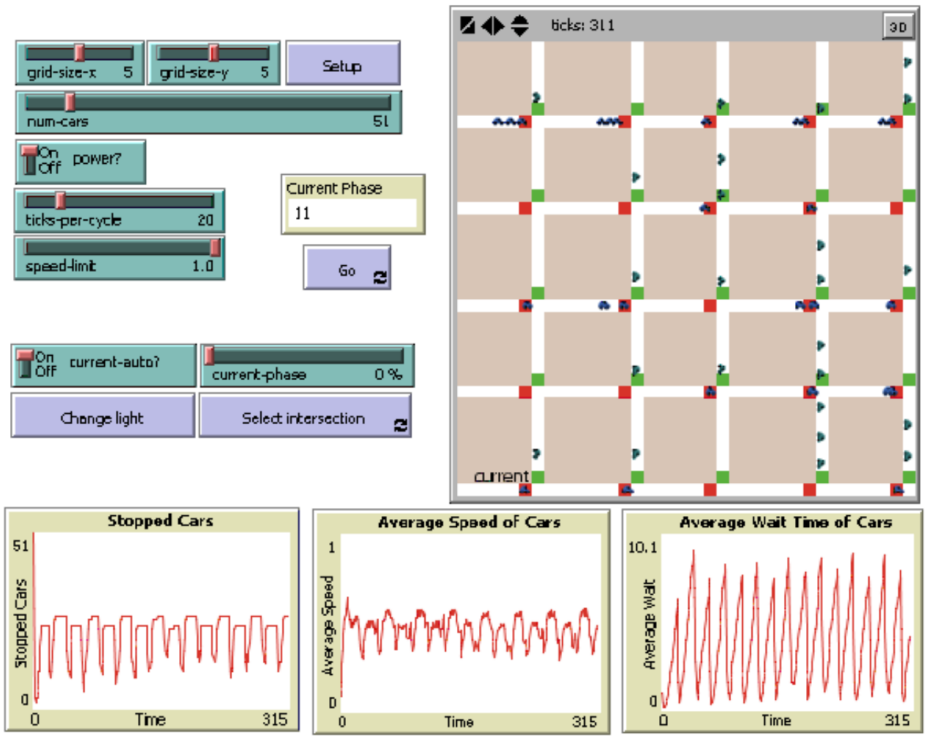

Figura 2.8: Esempio di implementazione di un modello in Netlogo per la visualizzazione del comportamento di pedoni e veicoli in una città

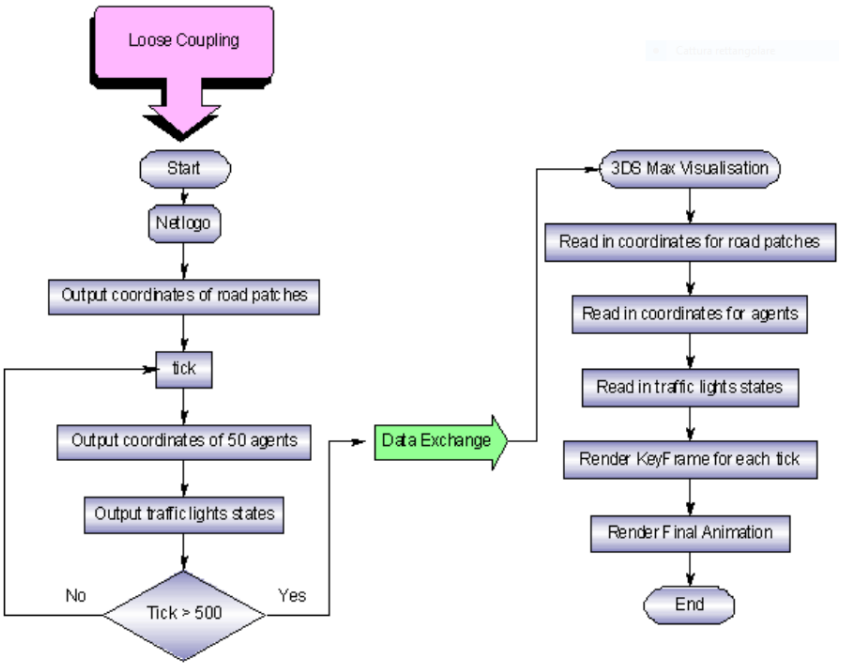

Figura 2.9: Dettaglio del passaggio dei dati della simulazione da Netlogo a 3DS Max

Mentre le piattaforme di sviluppo ABM nel loro complesso non hanno ancora incluso completamente le funzionalità 3D, un'altra serie di piattaforme per lo sviluppo dei modelli, che ha visto un aumento negli ultimi anni, che sembra essere in grado di supportare lo sviluppo di ABM 3D si trova nei "Game Engines"[7]. Essi offrono lo stato dell'arte della grafica 3D con un backend avanzato per la programmazione e l'implementazione dei modelli logici. Unity3D è stato ampiamente utilizzato come piattaforma di sviluppo in molteplici studi, dalla generazione di modelli animati di pesci [11], a studi sulla salute e sul benessere [10], studi sulla realtà virtuale [15], GIS comportamentale [16] e, ancora più rilevante, sistemi di agenti intelligenti [14]. Il secondo dei due articoli principali da cui questa tesi ha preso spunto è *ABMU: An Agent-Based Modelling Framework for Unity3D*[5], esso presenta un framework opensource basato sul motore grafico Unity3D creato per la modellazione grafica 3D dei modelli di simulazione ad agenti intelligenti e

[7]Motori di Gioco, ovvero pacchetti software avanzati per la produzione di videogiochi o contenuti digitali 3D ad alte prestazioni

fornisce metodi e classi per la facile implementazione delle funzionalità logiche alla base delle ABM. Oltre le capacità tecniche legate alla grafica a cui si può attingere usando Unity, è possibile qui usare tutte le librerie di terze parti sviluppate dalla grande community a supporto di questo motore di gioco. Gli autori illustrano la loro architettura sulla base del concetto di *Stepper*, ovvero uno schema di comportamento che deve essere ripetuto nel corso della simulazione e che contiene la logica che fa "agire" gli agenti del modello. L'esempio più importante di Stepper è il comportamento di un agente, infatti quando uno Stepper viene creato, viene registrato nell'oggetto *Scheduler* della simulazione. Come suggerisce il nome, lo Scheduler è responsabile di tenere traccia di tutti gli aspetti di scheduling della simulazione e, in particolare, contiene un elenco di tutti gli Stepper attualmente attivi nella simulazione e che vengono invocati per l'esecuzione al momento opportuno. Infine, una simulazione ABMU è gestita da un oggetto Controller, un modulo singleton che gestisce le varie fasi d una simulazione, come creazione e distruzione di oggetti e agenti, gestione della memoria e altro ancora. Questo framework non stravolge di per se il funzionamento di un ABM, infatti il concetto di comportamento di un agente (Stepper) eseguito in un certo ordine (Scheduler) è possibile ritrovarlo anche in altri motori di simulazione come ad esempio MASON che è già stato citato precedentemente. Uno schema riassuntivo dell'architettura di ABMU è presentato in è presentata nella Figura 2.10.

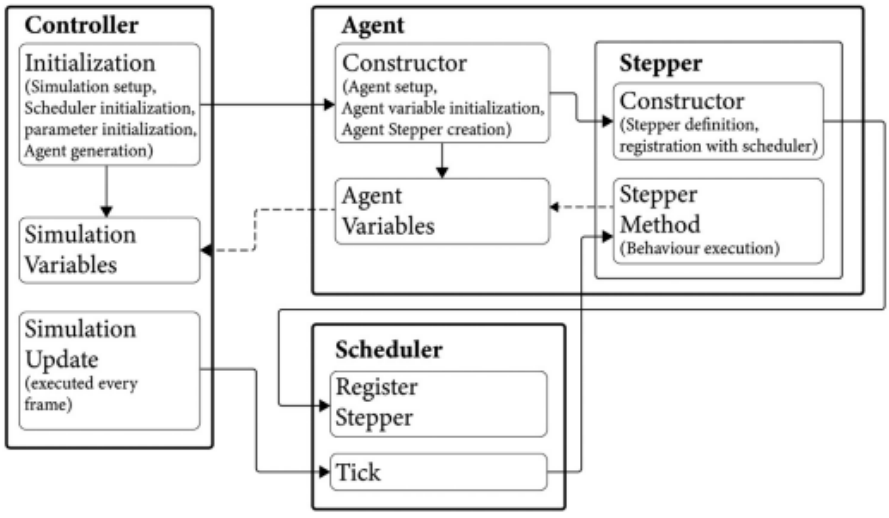

Figura 2.10: Schema architettura ABMU

Questo tipo di architettura rientra nel gruppo delle architetture, per lo sviluppo delle ABM, Tight-Coopled poichè sia la parte logica (in questo caso lo *Stepper*) sia la parte grafica (nativa in Unity) sono parte dello stesso applicativo e quindi non separabili uno dall'altro. A differenza dell'esempio precedente sull'architettura Loose-Coupled l'obiettivo in questo caso è di portare una nuova veste tecnologica alle ABM, un nuovo approccio per lo sviluppo e un nuovo bacino di librerie già integrate nei motori di gioco che grazie alla loro popolarità possono semplificare e arricchire ulteriormente lo sviluppo di nuove ABM. Nel framework sono stati implementaiti metodi per convertire le funzioni native di Unity in qualcosa di pronto all'uso per lo sviluppo delle ABM, e quindi la maggior parte degli strumenti e dei processi presenti in Unity possono essere utilizzati in ABMU. Questo ne aumenta significativamente il potenziale per lo sviluppo rapido di modelli complessi per diverse applicazioni, in quanto le librerie e le risorse Unity esistenti (ad esempio quelle presenti nell'Asset Store di Unity[23]) possono essere incorporati in un modello, senza dover reimplementare gli elementi di basso livello da zero, altri esempi possono essere l'uso di algoritmi di pathfinding, intelligenza artificiale o l'utilizzo del sistema di fisica integrato[24]. Lato grafico è possibile sfruttare le risorse di Unity anche per le funzionalità di di creazione di modelli 3D specifici, la generazione di visualizzazioni ad alto dettaglio (utilizzando uno qualsiasi dei numerosi modelli 3D, materiali, texture, shader, animazioni, ecc.). Per questo motivo, si prevede che ABMU essendo parte del più ampio ecosistema di Unity, fornisca significative capacità aggiuntive ai modelli 3D basati su agenti che possono essere sviluppati con esso. Per i dettagli implementativi dell'architettura appena descritta è possibile trovare i riferimenti sempre nell'articolo *ABMU: An Agent-Based Modelling Framework for Unity3D*[5].

2.2 Tecnologie

Partendo dalle nozioni viste in precedenza dei modelli basati su agenti e sui tipi di architetture, in questa sezione verranno descritte le componenti utilizzate e i motivi per cui sono state scelte. Partendo dall'architettura, ne è stata scelta una di tipo Loose-Coupled composta da un motore di simulazione con il compito di generare gli step di simulazione ed inviarli ad un client responsabile della visualizzazione, il tutto in tempo reale e con la possibilità di avere più client grafici che visualizzano in contemporanea la simulazione. Per quanto riguarda invece la parte di comunicazione tra la componente logica e grafica è stato introdotto un terzo componente, ovvero

un broker di messaggi in grado di ricevere le informazioni generate dal motore di simulazione e di veicolarle ai client connessi.
L'architettura creata, dunque, prende spunto da *Advances and Techniques for Building 3D Agent-Based Models for Urban Systems*[2] per quanto riguarda il concetto di disaccoppiamento tra logica e la parte grafica e da *ABMU: An Agent-Based Modelling Framework for Unity3D*[5] per quanto riguarda la parte di implementazione in Unity. Nelle prossime sezioni verrà descritto l'uso del motore di simulazione *MASON*[17] per la parte logica, il broker di messaggi *Moqsuitto*[7] per la parte di comunicazione tra le componenti grafiche e logiche e infine *Unity3D*[25] per la parte client responsabile della visualizzazione grafica.

2.2.1 MASON

Inizialmente la scelta per il motore di simulazione da utilizzare per l'architettura era tra un bacino limitato di soluzioni, per far si che la componente che eseguisse il modello di simulazione fosse il più semplice possibile da modificare, leggera dal punto di vista computazionale e che non fosse strettamente legata ad una componente grafica interna che ne impedisse l'utilizzo di una esterna. Questo lavoro è stato eseguito dal mio collega Pietro Russo nella sua tesi *Simulazioni ABM in Unity3D: Un framework di integrazione per simulazioni MASON*[21] in cui illustra nel dettaglio il panorama delle simulazioni basate su agenti e dei framework disponibili sul web per lo sviluppo dei modelli e di come il motore di simulazione selezionato è stato adattato per gli obiettivi del progetto. Tale scelta è ricaduta sul framework MASON, scritto in Java e sviluppato dalla George Mason University, progettato per supportare simulazioni ad elevato numero di agenti, leggero dal punto di vista computazionale e dotato di una componente logica denominata *Modello* non strettamente codificata con la componente grafica denominata *Visualizzazione* come illustrato in Figura 2.11. In MASON, dato il disaccoppiamento è possibile eseguire le ABM in modo indipendentemente alla grafica, e questo ha dato il via alla sostituzione del modulo Visualizzazione nativo con la nostra soluzione rendendo l'architettura Loose-Coupled integrando un broker di messaggi e un client esterno.

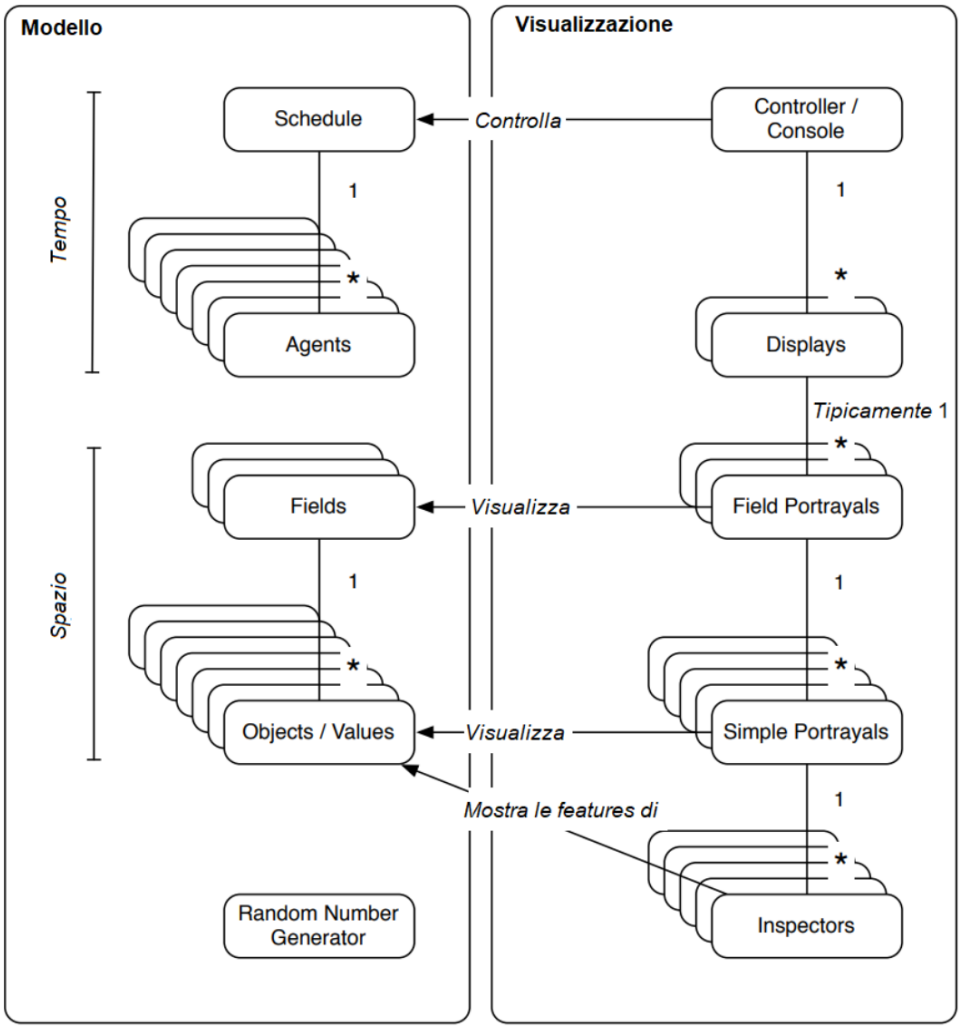

Figura 2.11: Schema di disaccoppiamento Modello-Visualizzazione di MASON

In questo specifico caso però l'interfaccia utente di MASON, visibile in Figura 2.12, fornisce anche funzionalità aggiuntive rispetto alla sola esecuzione dell'ABM senza visualizzazione, come la possibilità di mettere in pausa, riavviare, resettare, far avanzare uno step alla volta, salvare su disco e creare snapshot della simulazione, oppure modificarne i parametri prima del suo avvio senza dover alterare il codice manualmente. Queste possibilità sono state considerate cruciali per un ABM, tanto che sono diventate tra le funzionalità principali del client Unity sviluppate durante questo progetto.

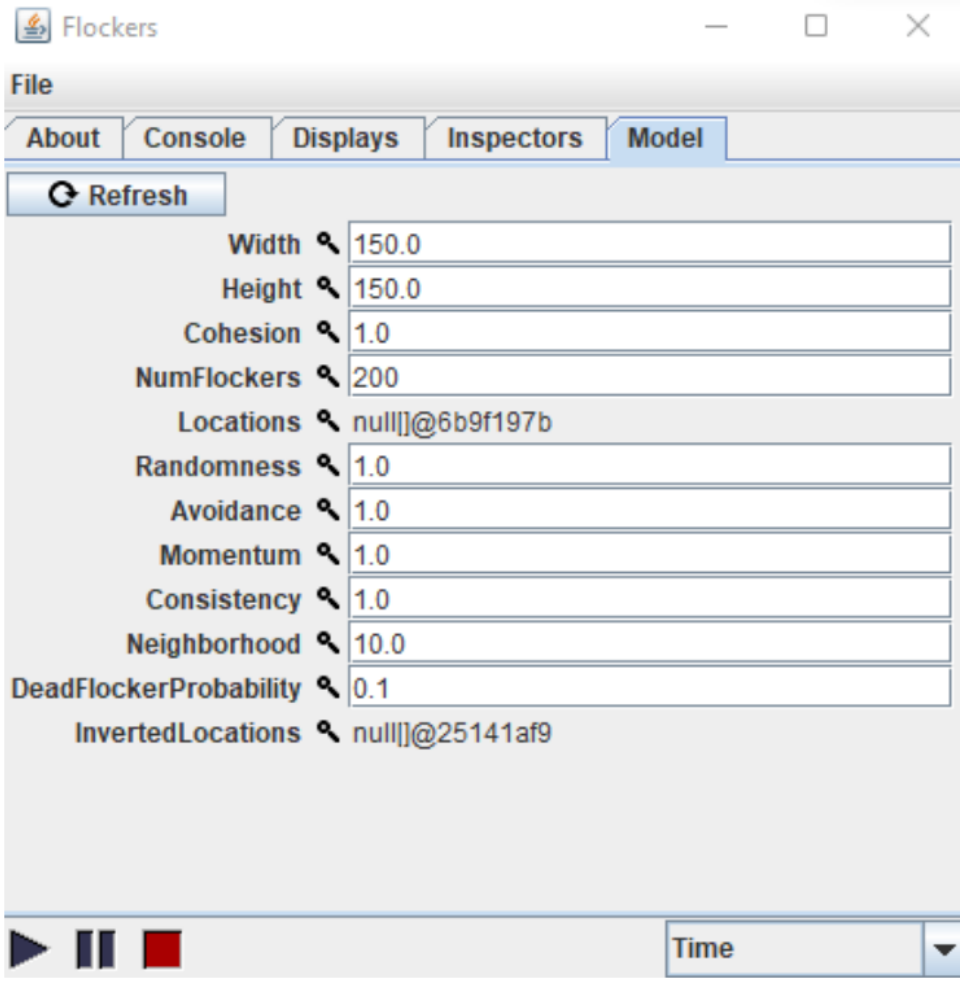

Figura 2.12: Interfaccia di personalizzazione di una simulazione in MASON

Le modifiche sostanziali apportare a MASON riguardano i seguenti punti:

Adattabilità delle simulazioni Nel panorama delle ABM non è mai stato definito uno standard a cui far riferimento per le implementazioni dei modelli. Questo porta ad una discrepanza in termini strutturali di come questi modelli vengono creati, andando ad aumentare la frammentazione e di conseguenza la difficoltà di chi entra in questo mondo per la creazioni di nuovi modelli. Ciò che è stato ritenuto necessario aggiungere è una struttura flessibile che dia il permesso di creare un'astrazione per qualsiasi tipo di problema si voglia rappresentare, per modellare

potenzialmente ogni tipo di problema che tramite le ABM vogliamo studiare. Al fine di raggiungere l'obiettivo è stato necessario realizzare dei wrapper[8] che astraggono i concetti fondamentali delle simulazioni (oggetti di simulazione e parametri), fungendo da entità fondamentali per l'integrazione delle simulazioni già presenti e delle future che si voglia implementare.

Controllo dello stato da remoto Dato che la simulazione viene eseguita su una macchina server, la possibilità di controllarla e modificarla deve passare per un protocollo di comunicazione client-server definito internamente. Il modo in cui il client interagisce con i parametri e gli oggetti di simulazione, varia in base al momento in cui si cerca di modificarli. Le due fasi sono al momento della creazione, dove si potrà inizializzare il modello e quindi poter modificare la maggior parte dei parametri a disposizione, e la fase dell'esecuzione dove è possibile modificare in tempo reale solo una parte di questi parametri.

Supporto a più dispositivi client Come accennato in fase di introduzione è stata prevista l'estensione del controllo della simulazione anche ai client Unity remoti in modo da avere un server di simulazione completamente usabile dall'esterno in modalità scatola nera prevedendo un'architettura di rete in grado di mettere in comunicazione i client con il server.

I dettagli implementativi possono essere trovati in *Simulazioni ABM in Unity3D: Un framework di integrazione per simulazioni MASON* [21]

2.2.2 MQTT

Per far si che un client e un server comunichino è necessario stabilire un protocollo di comunicazione su cui operare. In questo caso le condizioni necessarie, riguardo le prestazioni, per scegliere un protocollo sono:

- Leggero dal punto di vista computazionale
- Di semplice implementazione e utilizzo
- Che supporti un largo numero di linguaggi di programmazione

[8] *Wrapper*, oggetti che racchiudono funzionalità già esistenti e ne estendono delle caratteristiche

La scelta dunque è ricaduta sul protocollo MQTT[18], standard affermato per i dispositivi IoT[12][9]. Progettato secondo il concetto "publish/subscribe" in cui i dispositivi che si registrano su un topic[10] sono chiamati "*subscribers*", mentre i dispositivi che producono e dunque pubblicano i messaggi sul topic sono chiamati "*publishers*". Ideale per connettere dispositivi fisici di piccole dimensioni come sensori, telecamere, dispositivi di smart home ecc. con il minimo impatto sulla banda e sulle prestazioni. Nello specifico è stato utilizzato Mosquitto, un broker MQTT open-source il che significa che il codice sorgente è disponibile per chiunque e può essere utilizzato, modificato e distribuito liberamente. Questo rende possibile una facile integrazione con altre tecnologie e una personalizzazione delle funzionalità per soddisfare le esigenze specifiche del progetto, ma nel nostro caso l'implementazione attualmente disponibile è sufficiente ai nostri scopi infatti è stato utilizzato "out of the box"[11]. Inoltre, essendo un broker leggero e scalabile, Mosquitto è in grado di supportare grandi quantità di dati e di gestire un gran numero di dispositivi connessi contemporaneamente. Ciò lo rende una scelta ideale per la creazione di sistemi scalabili e affidabili.

2.2.3 Unity3D

Unity3D è una piattaforma di sviluppo per la creazione di giochi e applicazioni interattive. Offre un ambiente di sviluppo integrato (IDE) che permette agli sviluppatori di creare contenuti utilizzando una combinazione di asset grafici, codice, audio e altro ancora. La piattaforma supporta una vasta gamma di piattaforme, tra cui Windows, Mac, iOS, Android, WebGL e molte altre. Il motore di gioco di Unity3D rende facile la creazione di giochi e applicazioni interattive ad alte prestazioni infatti la piattaforma include anche una vasta gamma di strumenti e funzionalità per l'ottimizzazione delle prestazioni, la creazione di interfacce utente e la gestione dei dati. Gli sviluppatori possono utilizzare il linguaggio di programmazione C# e C++ per scrivere il proprio codice personalizzato e integrarlo con il motore di gioco. Inoltre, Unity3D offre una vasta gamma di opzioni di animazione, fisica, illuminazione e rendering, che consentono di creare giochi e applicazioni con un alto livello di realismo e dettaglio. La piattaforma include anche una libreria di asset pre-costruiti, tra cui modelli, texture, effetti speciali e altro ancora, che possono essere facilmente importati e utilizzati all'interno del gioco o dell'applicazione.

[9] "Internet of Things", o Internet delle cose

[10] Identificativo associato ai messaggi usato dal broker per instradare i messaggi ai dispositivi che si sono registrati al suddetto topic

[11] Senza modifiche, letteralmente "fuori dalla scatola"

Unity3D è anche molto popolare tra gli sviluppatori indipendenti, poiché offre una vasta gamma di opzioni di pubblicazione e distribuzione, tra cui la pubblicazione su app store, la distribuzione via Web e la creazione di contenuti per la realtà virtuale e aumentata. Inoltre, la piattaforma include una comunità attiva di sviluppatori e una vasta gamma di documentazione e tutorial che possono aiutare gli sviluppatori a imparare e utilizzare al meglio la piattaforma. Il successo avuto da Unity tra i piccoli sviluppatori è in parte dovuto alla sua possibilità di essere usato gratuitamente per progetti personali non commerciali, ma anche la caratteristica di avere come linguaggio principale di programmazione appunto C#, molto simile a Java anch'esso linguaggio molto comune e diffuso da molti anni.

Capitolo 3

Obiettivi del progetto

L'esistenza, dunque, di motori di simulazione già in grado di eseguire egregiamente i più svariati tipi di modelli ad agenti in combinazione con l'evoluzione dei motori grafici, hanno portato all'idea alla base di questo progetto di tesi di unire i due mondi in un'architettura in grado sfruttare i modelli già esistenti, inserirli in un contesto grafico coerente al modello stesso e, come punto principale, aggiungere la caratteristica della collaborazione e la condivisione dell'esperienza grafica tra gli utenti, aspetto che verrà illustrato nei prossimi capitoli.
Lo sviluppo ha richiesto l'utilizzo delle tecnologie precedentemente descritte in un'architettura client-server di tipo Loose-Coupled sviluppata da zero descritta in Figura 4.1, che vede come componenti principali:

- Un server per simulazioni ad agenti
- Un broker di messaggi che permette la comunicazione
- Un client basato su un motore grafico

Nello specifico la componente già esistente è il server di simulazione MASON, modificato ai fini di aggiungere la compatibilità con l'invio dei messaggi verso il broker di messaggi MQTT Mosquitto, responsabile dell'inoltro dei messaggi generati dal server verso i client, e infine i client Unity. Possiamo quindi riassumere gli obiettivi posti per il progetto con le seguenti parole chiave:

Visualizzazione Permettere a tutti i dispositivi di visualizzare le simulazioni in un contesto condiviso, al netto delle proprie performance per favorire il più possibile un'esperienza di utilizzo fluida, senza rinunciare alla collaboratività.

Condivisione Permettere l'interazione con la simulazione, dove prevista, all'interno dello spazio tridimensionale di Unity3D.

Adattabilità Conservare quanto più possibile inalterato il funzionamento base delle simulazioni.

3.1 Obiettivi personali

Dopo aver descritto gli obiettivi che ci siamo posti per l'intero progetto, da questo momento in poi, in questa tesi, verranno esposti gli obiettivi personali su cui ho lavorato. La tematica principale che ho affrontato è quella della **Condivisione**, ovvero permettere la corretta ricezione e interpretazione dei messaggi provenienti dalla componente server che esegue una simulazione ad agenti sui client Unity e che questi client possano visualizzare in contemporanea la simulazione e gestirla da remoto senza la necessità di un intervento diretto lato server. Nel dettaglio il mio compito è stato quello di sfruttare l'architettura che è stata progettata affinchè i client Unity possano gestire i messaggi di simulazione in arrivo dal server, regolare le proprie prestazioni in modo da riuscire a consumare i messaggi in arrivo rimanendo al passo con la simulazione ed essere in grado di produrre e ricevere i messaggi necessari per le gestione della stessa.

Capitolo 4

Architettura

Per soddisfare i requisiti posti come obiettivo, è stato pensato di inserire all'interno dell'architettura diversi topic su cui i client e il server avrebbero scambiato i messaggi. La struttura di comunicazione è mostrata in Figura 4.1 e possiamo dividere i topic in due gruppi principali:

Topic di comunicazione Gestiti dalla componente interna al server chiamata **Comm_client**[12] per la gestione della simulazione, come avvio, pausa, e configurazione iniziale.

- **All_to_mason**: Topic usato dai client per comunicare verso il server
- **Mason_to_all**: Topic usato dal server per comunicare in broadcast verso i client
- **Topic privato**: Topic usato dal server per comunicare in modo unidirezionale verso uno specifico client

Topic di simulazione Gestiti dalla componente interna al server chiamata **Sim_client**[13]

- **0 a N topic**: A partire dal topic *0* vengono inviati gli step di simulazione in ordine crescente fino al topic N[14] per poi ricominciare dal topic 0 e così via

[12] Abbreviazione *Communication Client*, oggetto interno al server MASON

[13] Abbreviazione di *Simulation Client*, in riferimento all'oggetto interno al server MASON che gestisce i messaggi di simulazione

[14] N è un numero arbitrario definito a priori in base alle esigenze dei client

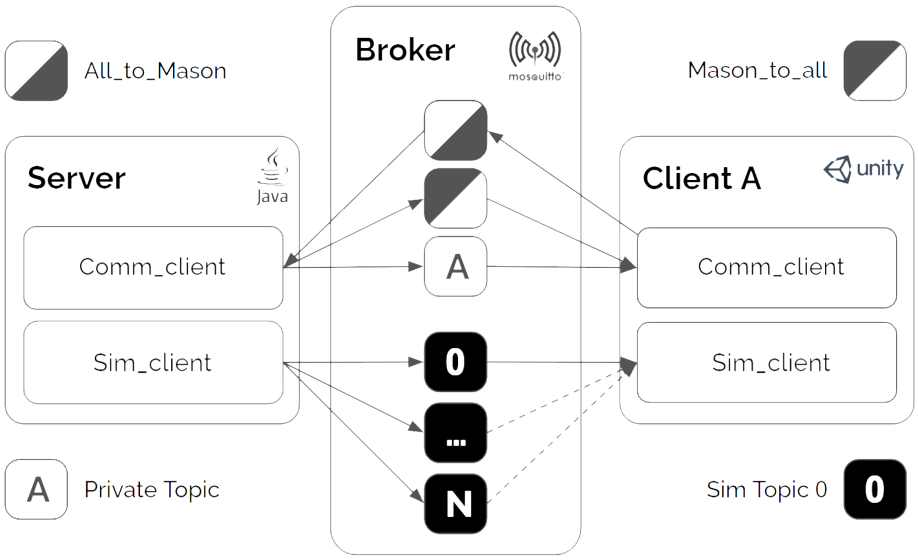

Figura 4.1: Architettura client-broker-server implementata

Sono presenti dunque due "strade" che possono prendere i messaggi in base alla loro tipologia. In Figura 4.1 i messaggi generati dalla simulazione e inviati dal **Sim_client** del server, vengono inviati agli N topic presenti sul broker e dall'altro lato i client dovranno essere in grado di ricevere questi messaggi gestiti sempre dal Sim_client implementato in lato Unity. L'altra "strada" viene percorsa dai messaggi di comunicazione, che comprendono tutto ciò che riguarda la gestione della simulazione, il canale di comunicazione è bidirezionale in quanto seppur i messaggi sono creati dalla componente **Comm_client** dei client, corrisponde sempre una risposta di conferma da parte del server sempre dalla componente Comm_client.

4.1 Struttura dei messaggi

4.1.1 Messaggio di simulazione

Il messaggio di simulazione contiene tutte le informazioni necessarie per poter visualizzare gli agenti e gli oggetti nello spazio tridimensionale. Definito anche come "step" in quanto è la "fotografia" dello stato di avanzamento della simulazione e per questo motivo si presta molto ad essere racchiusa in una struttura dati per essere trasmessa. La struttura principale è composta da:

- **Header**: prima parte del messaggio ad essere processata, contiene una quantità fissata di informazioni in una struttura definita a priori che servono per poter leggere la successiva parte del messaggio, il **Payload**
- **Payload**: seconda parte del messaggio, dinamica a differenza dell'header

Figura 4.2: Struttura dello step di simulazione

4.1.2 Messaggio di controllo

La seconda tipologia di messaggio riguarda i messaggi di controllo, ovvero tutti quelli scambiati tra client e server che servono per la gestione della simulazione, sia prima dell'avvio, sia durante la fase di personalizzazione che durante l'esecuzione per la modifica in tempo reale. In sintesi le tipologie di messaggio supportate lato server sono:

- CHECK_STATUS
- CONNECTION
- DISCONNECTION
- SIM_LIST_REQUEST
- SIM_INITIALIZE
- SIM_UPDATE
- SIM_COMMAND
- CLIENT_RESPONSE

- CLIENT_ERROR

Nei prossimi capitoli verranno approfondite le funzionalità e il contesto di utilizzo dei singoli messaggi.

4.2 Architettura del client Unity

Anche il client Unity presenta una sua architettura specifica, principalmente è composto da due moduli:

- Logico.
- Grafico.

Il modulo grafico si occupa delle gestione dell'interfaccia e della messa su schermo delle informazioni ricevute dal server e processate dal modulo logico. Facendo riferimento dunque alla Figura 4.3, la componente da me implementata è il *Modulo Logico*, modulo in cui sono presenti 3 principali componenti software:

- Communication controller.
- Simulation controller.
- Performance manager.

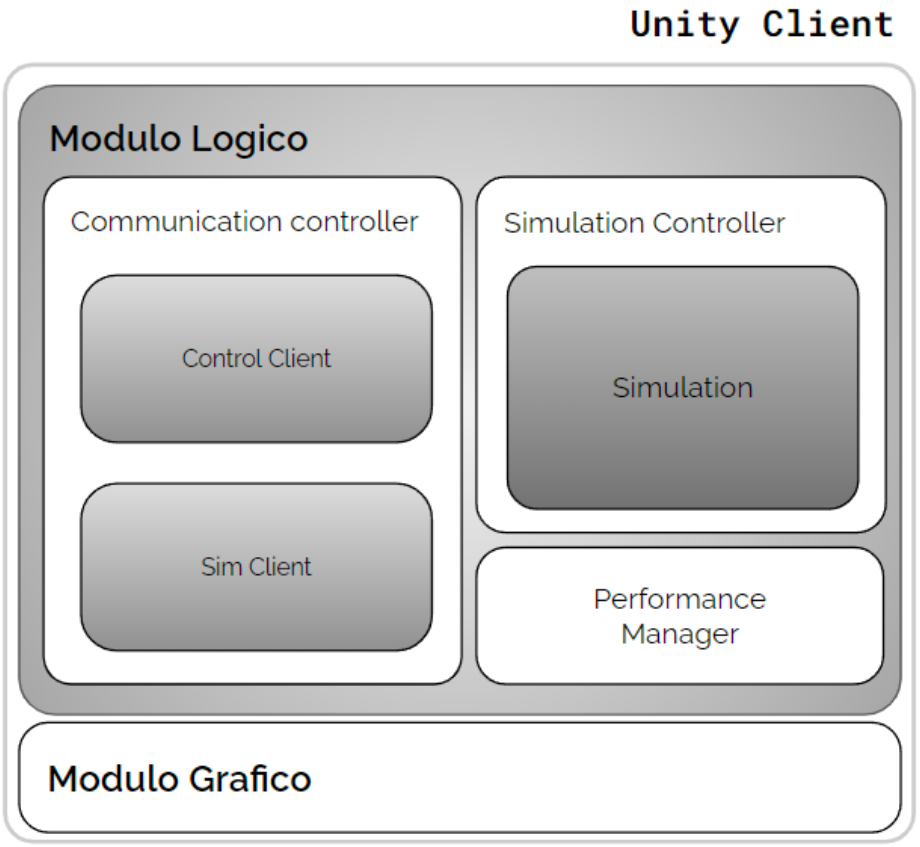

Figura 4.3: Architettura della componente client basata su Unity3D con dettaglio sul modulo logico

Queste tre componenti software sono state pensate per massimizzare il disaccoppiamento tra la parte logica che processa il messaggio in arrivo e la parte grafica che lo visualizza e anche tra le le componenti stesse del modulo, in questo modo nessuna delle due può rallentare l'altra, questa evenienza potrebbe accadere nel caso in cui sia necessario che il modulo grafico aspetti il termine dell'elaborazione di uno step di simulazione per poterlo visualizzare e di conseguenza si andrebbe a perdere molto tempo causando un notevole calo degli FPS[15] prodotti dall'intero client. Descrivendo ad alto livello queste tre componenti abbiamo:

[15]FPS sta per *Frames per seconds* ovvero immagini al secondo, più alto questo numero più è evidente una sensazione di fluidità delle immagini dal punto di vista dell'utente, lo standard è 60

Il Communication controller : contiene a sua volta due componenti client per la comunicazione verso il broker di messaggi. Queste due componenti, **Communication_client** e **Control_client** gestiscono in parallelo i messaggi in entrata ed uscita verso i rispettivi topic di competenza.

Il Simulation controller : ha il ruolo centrale nel modulo ovvero quello di gestore della simulazione dal punto di vista delle strutture dati e componenti del client Unity. L'oggetto principale di cui ha responsabilità è l'oggetto **Simulation** che non è nient'altro che la definizione sotto forma di oggetto di tutta la simulazione con i relativi parametri.

Il Performance manager : è l'unica componente non considerata un vero e proprio controller ma appunto un manager in quanto agisce in parallelo a tutte le componenti in modo completamente disaccoppiato analizzando in tempo reale le prestazioni che del client basandosi su diversi aspetti e parametri per garantire la migliore esperienza possibile per l'utente permettendogli di visualizzare la simulazione in tempo reale.

Capitolo 5

Communication controller

Il **Communication controller** gestisce i 2 client MQTT responsabili della comunicazione da e verso il server. La classe che lo implementa è mostrata nel listato 5.1.

```
1  public class CommunicationController
2  {
3      /// Communication Events ///
4      public static event EventHandler<EventArgs>
       OnControlClientConnectedHandler;
5      public static event EventHandler<EventArgs>
       OnSimClientConnectedHandler;
6      public static event EventHandler<EventArgs>
       OnControlClientDisconnectedHandler;
7      public static event EventHandler<EventArgs>
       OnSimClientDisconnectedHandler;
8
9      /// QUEUES ///
10     public ConcurrentQueue<MqttMsgPublishEventArgs>
       messageQueue = new ConcurrentQueue<
       MqttMsgPublishEventArgs>();
11     public ConcurrentQueue<MqttMsgPublishEventArgs>
       simMessageQueue = new ConcurrentQueue<
       MqttMsgPublishEventArgs>();
12     private SortedList<long, byte[]> secondaryQueue = new
       SortedList<long, byte[]>();
13
14     /// MQTT CLIENTS ///
15     private MQTTControlClient controlClient = new
       MQTTControlClient();
16     private MQTTSimClient simClient = new MQTTSimClient();
17     private static bool sim_client_ready = false,
       control_client_ready = false;
18
```

```
19      /// THREADS ///
20      private Thread controlClientThread;
21      private Thread simClientThread;
22  }
```

Listing 5.1: Definizione delle proprietà del Communication Controller

Nella classe sono stati definiti degli "eventi di comunicazione" tramite la classe **EventHandler** presente in C#. Gli eventi in .NET[1] sono basati sul modello del **delegato**. Il modello delegato segue l'"observer design pattern", che consente a un subscriber[16] di registrarsi e ricevere notifiche da un provider. Un mittente di eventi invia una notifica che si è verificato un evento e un destinatario di eventi riceve tale notifica e definisce una logica da applicare alla sua ricezione. In questa classe sono stati definiti i seguenti eventi:

OnControlClientConnectedHandler Evento in risposta alla connessione al broker MQTT del client che gestisce i messaggi di controllo verso il server.

OnSimClientConnectedHandler Evento in risposta alla connessione al broker MQTT del client che gestisce i messaggi di simulazione verso il server.

OnControlClientDisconnectedHandler Evento in risposta alla disconnessione al broker MQTT del client che gestisce i messaggi di controllo verso il server.

OnSimClientDisonnectedHandler Evento in risposta alla disconnessione al broker MQTT del client che gestisce i messaggi di simulazione verso il server.

Altra componente principale del Communication Controller sono le code. Divise in code usate per i messaggi di simulazione e per i messaggi di controllo abbiamo:

simMessageQueue Definita con la classe **ConcurrentQueue** è una struttura dati FIFO (First In First Out) thread-safe.

messageQueue Definita anche'essa con la classe **ConcurrentQueue** è una struttura dati FIFO (First In First Out) thread-safe.

secondaryQueue Definita come **SortedList** è una coda ordinata utile per inserire in ordine cronologico gli step arrivati.

[16] Chi si iscrive alla ricezione delle notifiche su determinati eventi

Infine ci sono le definizioni dei client MQTT:

- **simClient**
- **controlCLient**

e i thread **simclientThread** e **controlClientThread** che rispettivamente eseguono su thread separati da quelli usati da Unity, i client MQTT. L'avvio dei thread per il **controClient** e per il **simClient** avviene nei metodi **StartSimulationClient()** e **StartControlClient()** nel Listato 5.2

```
1      /// <summary>
2      /// Start Simulation MQTT Client
3      /// </summary>
4      public void StartSimulationClient()
5      {
6          simClient.ConnectionSucceeded +=
       OnSimClientConnected;
7          //simClient.ConnectionFailed +=
8          simClientThread = new Thread(() => simClient.Connect
       (ref simMessageQueue));
9          simClientThread.Start();
10     }
11
12     /// <summary>
13     /// Start Control MQTT Client
14     /// </summary>
15     public void StartControlClient()
16     {
17         controlClient.ConnectionSucceeded +=
       OnControlClientConnected;
18         //controlClient.ConnectionFailed +=
19         controlClientThread = new Thread(() => controlClient
       .Connect(ref messageQueue));
20         controlClientThread.Start();
21     }
```

Listing 5.2: Avvio dei thread nel Communication Controller

La lista di metodi implementati è mostrata nel Listato 5.3

```
1  /// METHODS ///
2
3      /// MQTT Clients
4
5      /// <summary>
6      /// Start Simulation MQTT Client
7      /// </summary>
8      public void StartSimulationClient()
```

```
9
10
11      /// <summary>
12      /// Start Control MQTT Client
13      /// </summary>
14      public void StartControlClient()
15
16      /// <summary>
17      /// Disconnect and aborts Simulation MQTT Client
18      /// </summary>
19      public void DisconnectSimulationClient()
20
21      /// <summary>
22      /// Disconnect and aborts Control MQTT Client
23      /// </summary>
24      public void DisconnectControlClient()
25
26      // Sim Client
27
28      /// <summary>
29      /// Sim Client SubscribeOnly Wrapper
30      /// </summary>
31      public void SubscribeOnly(int[] topics)
32
33      /// <summary>
34      /// Sim Client SubscribeAll Wrapper
35      /// </summary>
36      public void SubscribeAll()
37
38      /// <summary>
39      /// Sim Client SubscribeTopics Wrapper
40      /// </summary>
41      public void SubscribeTopics(int[] topics)
42
43
44      /// <summary>
45      /// Sim Client UnsubscribeTopics Wrapper
46      /// </summary>
47      public void UnsubscribeTopics(int[] topics)
48
49
50      // Control Client
51
52      /// <summary>
53      /// Control Client SubscribeTopic Wrapper
54      /// </summary>
55      public void SubscribeTopic(string nickname)
56
57      /// <summary>
```

```
58      /// Control Client UnsubscribeTopic Wrapper
59      /// </summary>
60      public void UnsubscribeTopic(string nickname)
61
62
63      /// OnEvent Methods ///
64      private void OnSimClientConnected()
65
66      private void OnControlClientConnected()
67
68      private void OnSimClientDisconnected()
69
70      private void OnControlClientDisconnected()
71
72
73      /// Utilities ///
74
75      /// <summary>
76      /// Send message to MASON
77      /// </summary>
78      public void SendMessage(string sender, string op,
        JSONObject payload)
79
80      /// <summary>
81      /// Send keepAlive message to MQTT Broker
82      /// </summary>
83      public void KeepAliveSimClient()
84
85      /// <summary>
86      /// Empty queues
87      /// </summary>
88      public void EmptyQueues()
89
90      public ref int GetStepsReceived()
91
92      public void Quit()
93
94  }
```

Listing 5.3: Metodi del Communication Controller

Particolare menzione vafatta per i seguenti metodi:

SubscribeOnly()/SubscribeAll() Sono metodi che servono per sottoscriversi ai topic del broker MQTT al fine di ricevere i messaggi. Il metodo **SubscribeAll()** permette di iscriversi a tutti i topic (60 nella nostra implementazione) mentre, **SubscribeOnly()**, viene usato per iscriversi

solo ad uno specifico numero di topic, richiamato tramite evento dal **Performance Manager**.

SendMessage() Metodo per inviare i messaggi di controllo.

5.1 Control Client

Il messaggi di controllo della simulazioni usati lato client sono:

- CHECK_STATUS
- CONNECTION
- DISCONNECTION
- SIM_LIST_REQUEST
- SIM_LIST_INITIALIZE
- SIM_UPDATE
- SIM_COMMAND

mentre quelli implementati lato server ma che non sono stati usati sono:

- CLIENT_RESPONSE
- CLIENT_ERROR

Tutti queste tipologie di messaggi sono in formato JSON e hanno le seguenti funzionalità:

CHECK_STATUS messaggio di heartbeat per mantenere attiva la connessione con il broker MQTT e per far capire al server che quel determinato client è ancora connesso e attivo.

CONNECTION Messaggio per stabilire la connessione.

DISCONNECTION Messaggio per comunicare al server la volontà da parte del client di disconnettersi.

SIM_LIST_REQUEST Il server supporta solo determinate simulazioni, quindi tramite questo messaggio il client può aggiornarsi con la lista più recente delle simulazioni disponibili per poi successivamente scegliere quale usare. La lista in riposta conterrà tutto il necessario per poter inizializzare le risorse lato Unity per la visualizzazione e il corretto processamento degli step di simulazione.

SIM_LIST_INIZIALIZE A partire dalla lista delle simulazioni disponibili ricevuta con la SIM_LIST_REQUESTS è possibile scegliere il prototipo della simulazione a cui si è interessati e personalizzare:

- Dimensioni dello spazio di simulazione
- Parametri di simulazione
- Numero di agenti e i loro parametri

Con questo messaggio, il prototipo della simulazione viene alterato e inviato al server con i nuovi valori.

SIM_UPDATE Una volta avviata la simulazione è possibile alterarla aggiungendo/modificando/rimuovendo oggetti di simulazione e/o alterandone i parametri.

SIM_COMMAND A simulazione in corso è possibile controllarla con i seguenti comandi:

- Play
- Pausa
- Stop
- Cambio velocità: sono supportate le velocità:
  - **0.25x** Un quarto della velocità standard
  - **0.5x** Metà della velocità standard
  - **1x** Velocità standard
  - **2x** Doppio della velocità standard
  - **MAX** Velocità massima di produzione supportata dal server

Queste velocità sono relative alla velocità di produzione degli step da parte del server, per convenzione è stata assunta una velocità di produzione di 60 step al secondo come velocità standard.

5.2 Sim Client

Il **Sim_Client** o **Simulation_Client** è la componente che si occupa della ricezione degli step di simulazione e della sottoscrizione ai topic del broker MQTT. Il ruolo di questo client è quello di ricevere e mettere in primo luogo i messaggi ricevuti all'interno della coda thread-safe **simMessageQueue**. L'essere thread-safe in questo caso non è solo una precauzione ma una vera

e propria necessità sorta durante lo sviluppo per far si che anche altri thread possano accedere alla coda senza generare inconsistenze. Essenzialmente l'unico altro thread che vi accede è il **Simulation Controller** in quanto è l'unico ad avere conoscenza dello stato di avanzamento della simulazione e quindi avrà anche il compito di estrarre dalla prima locazione di memoria della coda lo step e inserirlo in modo ordinato all'interno della seconda coda presente all'interno del Communication Controller, ovvero la lista ordinata **secondaryQueue**.
Seconda funzione del **Sim_Client** è quella di iscriversi ai topic del broker per poter ricevere gli step di simulazione. Come spiegato nel Capitolo 4 il topic da cui riceve gli step non è solo uno ma bensì un numero variabile che va da un minimo di 1 ad un massimo di 60. La decisione su quanti e a quali topic bisogna iscriversi è delegata al **Performance Manager** che in seguito al calcolo di alcune metriche darà il comando al Sim_Client di iscriversi o disiscriversi a determinati topic.

Capitolo 6

Simulation Controller

6.1 Struttura del *Simulation Controller*

Il **Simulation Controller** è il centro dell'architettura client, gestisce tutte le altre componenti e consiste nella classe che ha tutte le informazioni riguardanti la simulazione. Nel Listato 6.1 sono mostrate tutte le variabili, strutture dati e event handler[17] gestite dal Simulation Controller. Nello specifico le più importanti:

EventHandler : Divisi per categorie:

- Queues: event handler innescati alla ricezione di messaggi sulle code del Communication Controller
- Messages: event handler innescato al momento della disconnessione dell'utente admin e della conseguente assegnazione del ruolo di admin ad un altro utente ancora connesso da parte del server
- Responses: event handler usati per le risposte da parte del server ad azioni richieste dal client (es. alla conferma della ricezione del messaggio di **CHECK_STATUS** da parte del server viene innescato l'evento **OnCheckStatusSuccessEventHandler** oppure **OnCheckStatusUnsuccessEventHandler**)

Managers : l'unica classe Manager è il **Performance Manager** che si occupa di bilanciare il numero di topic a cui iscriversi in caso di scarse prestazioni.

[17] Event Handler = Gestore di Eventi, design pattern per la gestione di eventi generati da altre componenti software

Controllers : Controllers i quali gestiscono una parte dell'architettura client, divisi in controllers del modulo logico:

- Communication_Controller

e del modulo grafico:

- UIController
- MenuController
- SceneController

Variabili di stato : variabili e strutture dati per la gestione della simulazione:

- sim_id: id lato client per identificare le simulazioni
- clientState: stato attuale del client
- StateEnum: elenco di stati in cui può trovarsi il client
- serverSide_simId: id che identifica la simulazione lato server
- serverside_simState: stato della simulazione lato server
- Command: elenco di comando richiedibili al server tramite il Communication Controller e il relativo control client
- admin: indica se il client è admin o meno della simulazione, l'admin è l'unico utente durante una simulazione che può inviare i messaggi di modifica della simulazione (es. aggingengere/rimuovere/modificare gli oggetti di simulazione e inviare i messaggi di controllo)

```
1  public class SimulationController : MonoBehaviour
2  {
3      // Player Preferences
4      [SerializeField] private PlayerPreferencesSO
       playerPreferencesSO;
5      [SerializeField] public SimObject defaultSimObject;
6
7      /// EVENT HANDLERS ///
8
9      /// Queues
10     public event EventHandler<ReceivedMessageEventArgs>
       MessageEventHandler;
11     public event EventHandler<StepMessageEventArgs>
       StepMessageEventHandler;
12
```

```
13      /// Messages
14      public static event EventHandler<
        ReceivedMessageEventArgs> OnNewAdminEventHandler;
15
16      /// Responses
17      public static event EventHandler<
        ReceivedMessageEventArgs>
        OnCheckStatusSuccessEventHandler;
18      public static event EventHandler<
        ReceivedMessageEventArgs>
        OnCheckStatusUnsuccessEventHandler;
19      public static event EventHandler<
        ReceivedMessageEventArgs>
        OnConnectionSuccessEventHandler;
20      public static event EventHandler<
        ReceivedMessageEventArgs>
        OnConnectionUnsuccessEventHandler;
21      public static event EventHandler<
        ReceivedMessageEventArgs>
        OnDisonnectionSuccessEventHandler;
22      public static event EventHandler<
        ReceivedMessageEventArgs>
        OnDisconnectionUnsuccessEventHandler;
23      public static event EventHandler<
        ReceivedMessageEventArgs> OnSimListSuccessEventHandler;
24      public static event EventHandler<
        ReceivedMessageEventArgs> OnSimListUnsuccessEventHandler
        ;
25      public static event EventHandler<
        ReceivedMessageEventArgs> OnSimInitSuccessEventHandler;
26      public static event EventHandler<
        ReceivedMessageEventArgs> OnSimInitUnsuccessEventHandler
        ;
27      public static event EventHandler<
        ReceivedMessageEventArgs> OnSimUpdateSuccessEventHandler
        ;
28      public static event EventHandler<
        ReceivedMessageEventArgs>
        OnSimUpdateUnsuccessEventHandler;
29      public static event EventHandler<
        ReceivedMessageEventArgs>
        OnSimCommandSuccessEventHandler;
30      public static event EventHandler<
        ReceivedMessageEventArgs>
        OnSimCommandUnsuccessEventHandler;
31      public static event EventHandler<
        ReceivedMessageEventArgs>
        OnClientErrorSuccessEventHandler;
32      public static event EventHandler<
```

```
    ReceivedMessageEventArgs>
    OnClientErrorUnsuccessEventHandler;
33
34      /// MANAGERS ///
35      public PerformanceManger PerfManager;
36
37      /// CONTROLLERS ///
38      private UIController UIController;
39      private MenuController MenuController;
40      private SceneController SceneController;
41      public CommunicationController CommController;
42
43      /// SIM-RELATED VARIABLES ///
44
45      /// State
46      public static int sim_id = 0;
47      private StateEnum clientState = StateEnum.NOT_READY;
48      private Simulation simulation;
49
50      public static int serverSide_simId = 0;
51      public static Simulation.StateEnum serverSide_simState =
        Simulation.StateEnum.NOT_READY;
52      public enum StateEnum
53      {
54          CONN_ERROR = -2,     // Error in connection
55          NOT_READY = -1,      // Client is not connected
56          CONNECTED_MQTT = 0, // Client is connected to
57                               // MQTT Broker
58          LOGGED_IN = 1,       // Client is connected to
59                               // Simulation Server
60          READY = 2,           // Client is ready to
61                               // join a Simulation
62          INIT = 3,            // Client is Initializing
63                               // a Simulation
64          IN_GAME = 4          // Client has created/joined
65                               // a Simulation
66      }
67      public enum Command
68      {
69          STEP,
70          PLAY,
71          PAUSE,
72          STOP,
73          SPEED
74      }
75      public static bool admin = true;
76      public static string admin_name;
77      public static JSONArray sim_list_editable;
78      public static JSONArray sim_prototypes_list = new
```

```
        JSONArray();
79      private JSONObject uncommitted_updatesJSON = new
        JSONObject();
80      public ConcurrentDictionary<(string op, (SimObject.
        SimObjectType type, string class_name, int id) obj),
        SimObject> uncommitted_updates = new
        ConcurrentDictionary<(string, (SimObject.SimObjectType,
        string, int)), SimObject>();
81
82      /// Support variables
83
84      long start_millis = DateTime.Now.Ticks / TimeSpan.
        TicksPerMillisecond;
85
86      private static System.Random random = new System.Random
        ();
87      private string nickname = RandomString(10);
88      private long latestSimStepArrived = 0;
89      private int steps_to_consume = 0;
90      public int stepsConsumed = 0;
91
92      /// Threads
93      private Thread StepQueueHandlerThread;
94      private Thread MessageQueueHandlerThread;
95      private Thread PerformanceManagerThread;
96
97      // SimController Action Queue
98      public static readonly ConcurrentQueue<Action>
        SimControllerThreadQueue = new ConcurrentQueue<Action>()
        ;
99
100     public long LatestSimStepArrived { get =>
        latestSimStepArrived; set => latestSimStepArrived =
        value; }
101     public int Steps_to_consume { get => steps_to_consume;
        set => steps_to_consume = value; }
102     }
```

Listing 6.1: Strutture dati della classe Simulation Controller

6.2 Flusso dei messaggi nel client

Il Simulation Controller essendo il cuore dell'architettura è anche quella componente che viene usata più spesso in tutte le sezioni dell'applicazione. Ciò vuol dire che fin dal primo avvio dell'applicazione fino alla sua chiusura il Simulation Controller è coinvolto in qualche funzionalità.
Per descrivere al meglio tutte le sue funzionalità immerse nel modulo logico è necessario partire dall'avvio dell'applicazione e seguire il flusso dei messaggi per poter capire al meglio l'elaborazione apportata ad ogni tipologia di messaggio che fa si che infine tutto sia pronto per essere visualizzato a schermo dal modulo grafico.
All'avvio il Simulation Controller ha il compito di inizializzare tutti altri controller e manager e gestire l'avvio delle componenti che vanno eseguite su thread separati, come il Performance Manager, il consumo dei messaggi dalle code e i client MQTT come mostrato nel Listato 6.2

```
1  ...
2
3   /// Threads
4      private Thread StepQueueHandlerThread;
5      private Thread MessageQueueHandlerThread;
6      private Thread PerformanceManagerThread;
7
8  ...
9
10 /// UNITY LOOP METHODS ///
11
12     /// <summary>
13     /// We use Awake to bootstrap App
14     /// </summary>
15     private void Awake()
16     {
17         ...
18
19         // Instantiate Simulation
20         simulation = new Simulation(defaultSimObject);
21
22         // Retrieve Controllers/Managers
23         CommController = new CommunicationController();
24         MenuController = GameObject.Find("MenuController").
       GetComponent<MenuController>();
25         PerfManager = new PerformanceManger();
26
27         // Bootstrack background tasks
28         BootstrapControlTasks();
29
```

```
30          ...
31      }
32
33  ...
34      /// <summary>
35      /// Bootstrap background tasks
36      /// </summary>
37      private void BootstrapControlTasks()
38      {
39          CommController.StartControlClient();
40          StartMessageQueueHandlerThread();
41      }
42
43  ...
44
45  /// Queue Handling
46
47      /// <summary>
48      /// Start steps message handler
49      /// </summary>
50      public void StartStepQueueHandlerThread()
51      {
52          UnityEngine.Debug.Log(this.GetType().Name + " | " +
        System.Reflection.MethodBase.GetCurrentMethod().Name + "
         | Step Management Thread started..");
53          StepQueueHandlerThread = new Thread(delegate () {
        StepQueueHandler(ref Simulation.state, ref
        PerformanceManger.SORTING_THRESHOLD); });
54          StepQueueHandlerThread.Start();
55      }
56      /// <summary>
57      /// Start messages handler
58      /// </summary>
59      public void StartMessageQueueHandlerThread()
60      {
61          UnityEngine.Debug.Log(this.GetType().Name + " | " +
        System.Reflection.MethodBase.GetCurrentMethod().Name + "
         | Message Handler Thread started..");
62          MessageQueueHandlerThread = new Thread(
        MessageQueueHandler);
63          MessageQueueHandlerThread.Start();
64      }
65
66  ...
```

Listing 6.2: Inizializzazione e avvio dei thread

il metodo Awake[4] è un metodo presente già nelle classi native di Unity e rappresenta quel metodo eseguito da Unity solo una volta durante la durata dell'istanza dello script. La durata di uno script dura fino a quando la scena[22] che lo contiene non viene scaricata. Se la scena viene caricata di nuovo, Unity carica di nuovo l'istanza dello script, quindi Awake verrà chiamato di nuovo. Se la scena viene caricata più volte in modo additivo, Unity carica diverse istanze di script, quindi Awake verrà chiamato più volte. Dato che è collegato alla scena che mostra il menù di avvio (Figura 6.1), di conseguenza avremo che il metodo Awake del Simulation Controller verrà chiamato e tutto ciò che è al suo interno eseguito. Come mostrato nel Listato 6.2 sempre durante l'avvio dell'applicazione, appena dopo l'inizializzazione delle variabili vengono eseguiti i metodi di bootstrap delle componenti interne al Simulation_Controller, come ad esempio l'avvio dei client MQTT per la ricezione e l'invio dei messaggi e i thread per il consumo dei stessi. I client vengono avviati tramite i metodi **StartSimulationClient()** e **StartControlClient()** presenti all'interno del Communication_Controller tramite i metodi di bootstrap nel Listato 6.3.

Figura 6.1: Menù iniziale all'avvio dell'applicazione

```
/// <summary>
/// Bootstrap background tasks
/// </summary>
private void BootstrapControlTasks()
{
    CommController.StartControlClient();
    StartMessageQueueHandlerThread();
}
private void BootstrapSimulationTasks()
{
    CommController.StartSimulationClient();
    StartPerformanceManagerThread();
    StartStepQueueHandlerThread();
}
```

Listing 6.3: Metodi che racchiudono l'avvio dei client MQTT e dei thread di consumo dei messaggi

All'interno di questi medoti è presente anche l'avvio dei thread che gestiscono il consumo dei messaggi nelle code, tramite **StartStepQueueHandlerThread()** e **StartMessageQueueHandlerThread()** mostrati nel Listato 6.4. Lo **StartStepQueueHandlerThread()** si occupa di estrarre dalla coda **simQueue** gli step di simulazione mentre lo **StartMessageQueueHandlerThread()** preleva i messaggi relativi al controllo della simulazione dalla coda **messageQueue**.

```
/// <summary>
/// Start steps message handler
/// </summary>
public void StartStepQueueHandlerThread()
{
    UnityEngine.Debug.Log(this.GetType().Name + " | " +
    System.Reflection.MethodBase.GetCurrentMethod().Name + "
     | Step Management Thread started..");
    StepQueueHandlerThread = new Thread(delegate () {
    StepQueueHandler(ref Simulation.state, ref
    PerformanceManger.SORTING_THRESHOLD); });
    StepQueueHandlerThread.Start();
}
/// <summary>
/// Start messages handler
/// </summary>
public void StartMessageQueueHandlerThread()
{
    UnityEngine.Debug.Log(this.GetType().Name + " | " +
    System.Reflection.MethodBase.GetCurrentMethod().Name + "
     | Message Handler Thread started..");
```

```
16      MessageQueueHandlerThread = new Thread(
        MessageQueueHandler);
17      MessageQueueHandlerThread.Start();
18  }
```

Listing 6.4: Avvio dei thread per la gestione del consumo dei messaggi nelle rispettive code di messaggi di controllo e simulazione

Cliccando su "New Simulation" nel menù iniziale in Figura 6.1 viene mostrata la scena di selezione delle simulazioni, Figura 6.3 e, in background, viene richiesta la lista delle simulazioni disponibili lato server attraverso i metodi nel Listato 6.5 **SendSimListRequest()** e **SendSimInitialize()**. Questa scena viene visualizzata solo se si è l'**admin** della simulazione, ovvero se si è il primo client ad entrare nella scena in Figura 6.3, la scelta è stata fatta empiricamente sul fatto che viene creata una lobby implicitamente dal primo client che si connette in quanto il server supporta comunque una sola simulazione alla volta. In caso in cui non si è il primo a cliccare su "Start Simulation" sarà presente la scritta "Join Simulation" in Figura 6.2 al suo posto e non sarà possibile scegliere la simulazione o modificarne i parametri ma solo aspettare che l'admin la avvii.

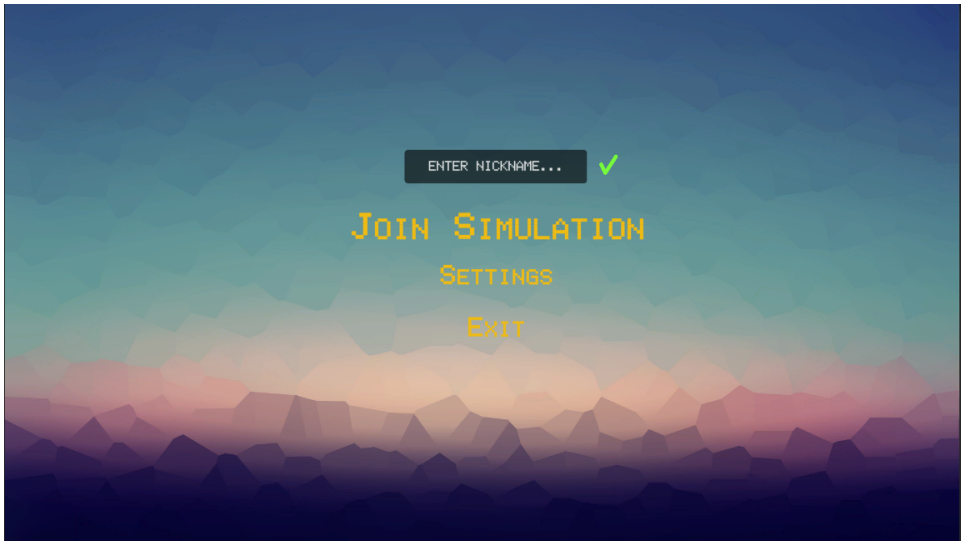

Figura 6.2: Menù di partecipazione ad una simulazione esistente

Dopo aver ricevuto la lista e modificato i parametri interessati, cliccando su "Start Simulation" si entrerà nella scena dove si potrà visualizzare la

simulazione selezionata e tramite “Sim Settings” la modifica dei parametri di simulazione.

```
/// <summary>
/// Send sim list request
/// </summary>
public void SendSimListRequest()
{
    // Create payload
    JSONObject payload = new JSONObject(); ;
    // Send command
    UnityEngine.Debug.Log(this.GetType().Name + " | " +
    System.Reflection.MethodBase.GetCurrentMethod().Name + "
     | Sending SIM_LIST_REQUEST to MASON...");
    CommController.SendMessage(nickname, "003", payload);
}
/// <summary>
/// Send initialization message
/// </summary>
public void SendSimInitialize(JSONObject sim_initialized)
{
    UnityEngine.Debug.Log(this.GetType().Name + " | " +
    System.Reflection.MethodBase.GetCurrentMethod().Name + "
     | Sending SIM_INITIALIZE to MASON...");
    CommController.SendMessage(nickname, "004",
    sim_initialized);
}
```

Listing 6.5: Richiesta della lista di simulazioni e invio del prototipo di simulazione modificato

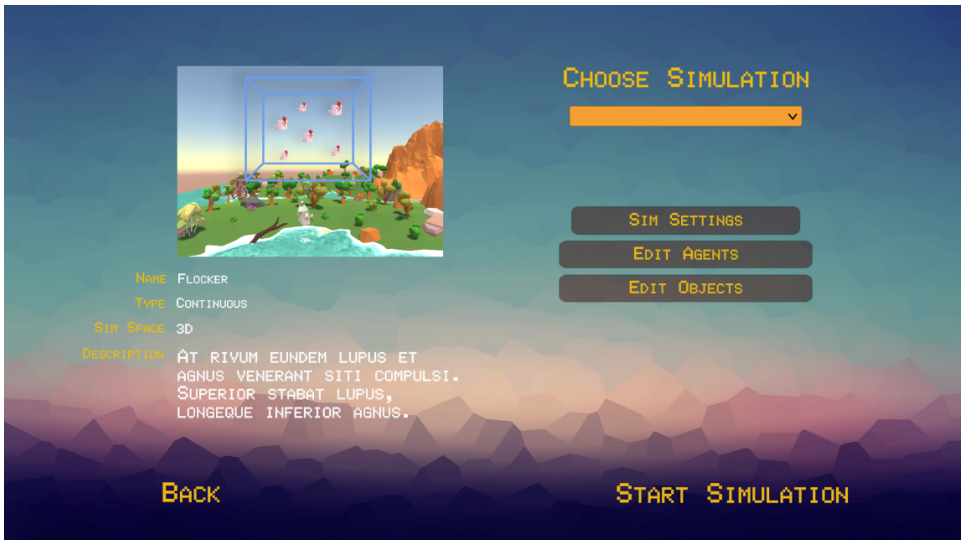

Figura 6.3: Scena di selezione delle simulazioni disponibili

SendSimListRequest() : tramite il control client viene richiesta la lista delle simulazioni, corrisponde al codice di operazione lato server numero 3. A questa richiesta segue in risposta il JSON contenente tutte le definizioni complete delle simulazioni per poter scegliere quella da modificare e utilizzare

SendSimInitialize() : metodo, che sempre tramite control client, invia il prototipo di simulazione in JSON dopo essere stato modificato.

In Figura 6.3 cliccando su "Start Simulation" si entra nella scena che mostra lo spazio di simulazione tridimensionale della simulazione scelta. A questo punto viene visualizzata la scena di simulazione, Figura 6.4, dove inizia la fase di ricezione degli step di simulazione.

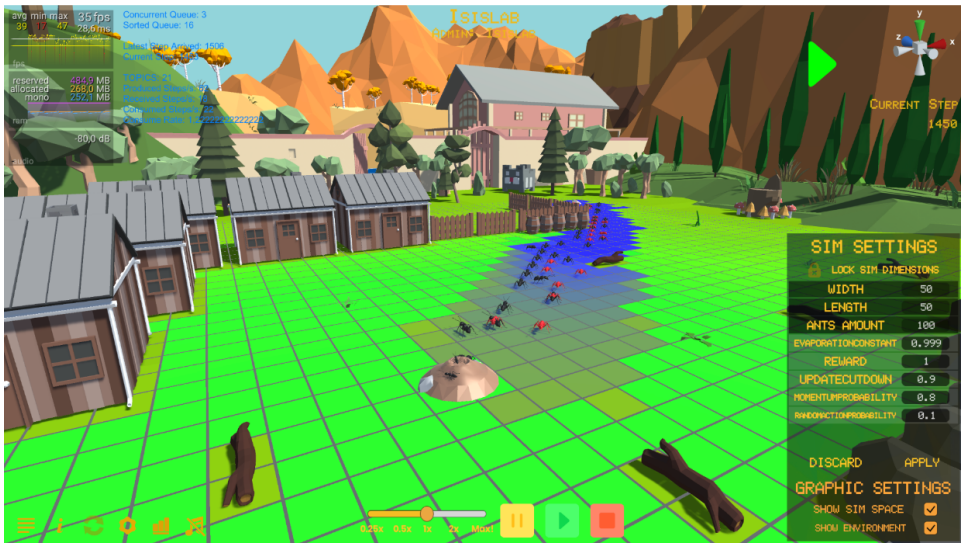

Figura 6.4: Scena di simulazione

In questa fase vengono inizializzate le componenti legate alla gestione della simulazione tramite il metodo **onLoadSimulationScene()** nel Listato 6.6.

```
1 private void onLoadSimulationScene(object sender, EventArgs
      e)
2 {
3     BootstrapSimulationTasks();
4     clientState = StateEnum.IN_GAME;
5     SceneController = GameObject.Find("SceneController").
      GetComponent<SceneController>();
6     UIController = GameObject.Find("UIController").
      GetComponent<UIController>();
7     while (!CommunicationController.SIM_CLIENT_READY) { }
8     Step();
9 }
```

Listing 6.6: Metoto di inizializzazione componenti e strutture dati per la gestione degli step di simulazione

In questo metodo sono presenti il bootstrap dei task di simulazione tramite **BootstrapSimulationTasks()**, mostrato nel Listato 6.7, dove sono presenti i metodi:

CommController.StartSimulationClient() : avvio del client MQTT che riceve gli step di simulazione.

StartPerformanceManagerThread() : avvio del thread di monitoraggio delle code e delle performance del client Unity.

StartStepQueueHandlerThread() : avvio del thread di consumo degli step di simulazione dalla coda.

```
1 private void BootstrapSimulationTasks()
2 {
3     CommController.StartSimulationClient();
4     StartPerformanceManagerThread();
5     StartStepQueueHandlerThread();
6 }
```

Listing 6.7: Metoto di bootstrap dei task di simulazione

6.2.1 Consumo dei messaggi di simulazione

A partire da questa fase nell'applicazione, terminano le procedure di setup e inizializzazione in quanto entrando nella scena di simulazione in Figura 6.4 è stata già decisa da parte del client admin la simulazione da eseguire e confermata lato server. All'interno di **BootstrapSimulationTasks()** è presente il metodo più importante durante la fase di simulazione, ovvero quello che avvia un thread in parallelo per il consumo degli step in arrivo sulla coda **messageQueue** trattata nel Paragrafo 5.2. Siccome il **simClient** si occupa della connessione verso il broker MQTT, della ricezione degli step e dell'inserimento degli stessi nella coda **messageQueue**, c'è necessità di dover ordinare gli step arrivati poichè data l'eterogeneità della rete tra il client Unity e il server, non possiamo essere sicuri che gli step arrivino nell'ordine con cui sono stati prodotti e inviati dal server. Per questo motivo è stata introdotta una seconda coda, **secondaryQueue**, definita nel communication controller nel Capitolo 5. Questa coda ha il compito di funzionare da buffer in cui vengono inseriti i messaggi ordinati per **stepID** ovvero un parametro presente all'interno dello step di simulazione che ne identifica numericamente l'ordine di produzione. All'arrivo, i messaggi nella **messageQueue**, saranno compressi, quindi prima di essere spostati e ordinati dovranno essere decompressi e solo successivamente messi nella **secondaryQueue** dal **Simulation Controller**. La **secondaryQueue** viene riempita fino al raggiungimento di una certa soglia **SORTING_THRESHOLD** definita empiricamente a 15, ovvero al raggiungimento dei 15 messaggi ordinati al suo interno sarà possibile estrarre dalla prima locazione lo step correttamente ordinato e visualizzarlo. La scelta del valore 15 non è casuale ma basata sul fatto che il server essendo

tarato di base per produrre al più 60 step di simulazione al secondo a velocità standard, un valore di 15 corrisponderà ad avere un ritardo tra ciò che sta producendo il server e ciò che sta visualizzando il client di un quarto di secondo più la latenza per l'arrivo del messaggio da server a client più il tempo necessario alla decompressione e all'ordinamento. Questa latenza è il minimo compromesso considerato accettabile per avere un buona sicurezza di visualizzare tutti gli step di simulazione in modo ordinato ed evitare situazioni in cui, magari per problemi di rete, arrivi prima un messaggio con uno **stepId** maggiore ad uno con **stepId** minore e quindi provocare una visualizzazione cronologicamente incoerente. Nel Listato 6.8 è mostrato il codice della funzione eseguita dal metodo **StartStepQueueHandlerThread()** chiamata **StepQueueHandler()** in cui è presente la logica con cui vengono spostati i messaggi dalla **messageQueue** alla **secondaryQueue**.

```
1  /// <summary>
2  /// Orders steps and updates last arrived one
3  /// </summary>
4  public void StepQueueHandler(ref Simulation.StateEnum
       sim_state, ref int SORTING_THRESHOLD)
5  {
6    JSONObject step = new JSONObject();
7    MqttMsgPublishEventArgs message;
8    while (true)
9    {
10     if (!clientState.Equals(StateEnum.IN_GAME)) continue;
11     if ((CommController.SecondaryQueue.Count >
       SORTING_THRESHOLD && sim_state.Equals(Simulation.
       StateEnum.PLAY)) || (CommController.SecondaryQueue.Count
        > 0 && steps_to_consume > 0))
12   {
13       if (!sim_state.Equals(Simulation.StateEnum.PLAY))
14       {
15           while (steps_to_consume > 0)
16           {
17               try
18               {
19                   StepMessageEventArgs e = new
       StepMessageEventArgs();
20                   e.Step = CommController.SecondaryQueue.
       Values[0];
21                   StepMessageEventHandler?.Invoke(this, e);
22                   CommController.SecondaryQueue.RemoveAt(0);
23                   --steps_to_consume;
24               }
25               catch (ArgumentOutOfRangeException e)
26               {
```

```
27                      UnityEngine.Debug.Log(this.GetType().Name
    + " | " +System.Reflection.MethodBase.GetCurrentMethod()
    .Name + " | Step Queue Empty!");
28                      --steps_to_consume;
29                  }
30              }
31          }
32          else
33          {
34              try
35              {
36                  StepMessageEventArgs e = new
    StepMessageEventArgs();
37                  e.Step = CommController.SecondaryQueue.Values
    [0];
38                  StepMessageEventHandler?.Invoke(this, e);
39                  CommController.SecondaryQueue.RemoveAt(0);
40              }
41              catch (ArgumentOutOfRangeException e)
42              {
43                  UnityEngine.Debug.Log(this.GetType().Name + "
    | " + System.Reflection.MethodBase.GetCurrentMethod().
    Name + " | Step Queue Empty!");
44              }
45          }
46      }
47      if (CommController.SimMessageQueue.Count > 0)
48      {
49          if (CommController.SimMessageQueue.TryDequeue(out
    message))
50          {
51              if (message.Message.Length.Equals(0))
52              {
53                  UnityEngine.Debug.Log(this.GetType().Name + "
    | " + System.Reflection.MethodBase.GetCurrentMethod().
    Name + " | Found a blank Step");
54                  continue;
55              }
56              if (Utils.GetStepId(message.Message) <= simulation
    .currentSimStep)
57              {
58                  UnityEngine.Debug.Log(this.GetType().Name + "
    | " + System.Reflection.MethodBase.GetCurrentMethod().
    Name + " | Found an old Step: n " + Utils.GetStepId(
    message.Message));
59                  continue;
60              }
61              latestSimStepArrived = Utils.GetStepId(message.
    Message);
```

```
62            if (latestSimStepArrived.Equals(1))
63            {
64                StepMessageEventArgs e = new
      StepMessageEventArgs();
65                e.Step = message.Message;
66                StepMessageEventHandler?.Invoke(this, e);
67                ChangeState(Command.STEP);
68                --steps_to_consume;
69            }
70            else
71            {
72                CommController.SecondaryQueue.Add(
      latestSimStepArrived, message.Message);
73            }
74        }
75        else { UnityEngine.Debug.Log(this.GetType().Name + " |
       " + System.Reflection.MethodBase.GetCurrentMethod().
      Name + " | Cannot Dequeue!"); }
76    }
77  }
78 }
```

Listing 6.8: Funzione StepQueueHandler() che gestisce il consumo dei messaggi di simulazione

6.2.2 Controllo remoto della simulazione

Facendo riferimento alla Figura 6.4, in basso al centro ci sono i pulsanti di **controllo simulazione** per la modifica della velocità di simulazione e i pulsanti Pausa, Play e Stop. Nella parte in basso a sinistra invece, in ordine da sinistra verso destra, i pulsanti per:

Menù modifica parametri : questo menù, visibile nell'angolo in basso a destra della Figura 6.4, permette di cambiare durante l'esecuzione della simulazione, alcuni parametri.

Info e statistiche : questo pulsante mostra a schermo i parametri statistici sulle performance real-time dell'applicazione. Queste statistiche sono mostrate nell'angolo in alto a sinistra e presentano il conteggio degli fps, il grafico della stabilità degli stessi e la memoria occupata nei riquadri di sinistra, mentre di fianco le statistiche sullo stato delle code e dei messaggi.

I pulsanti di **controllo simulazione** azionano i metodi mostrati nel Listato 6.9.

```
 /// CONTROL ///

/// <summary>
/// get one Step of simulation
/// </summary>
public void Step()
{
    // check state
    SendSimCommand(Command.STEP, 0);
}
/// <summary>
/// Play simulation
/// </summary>
public void Play()
{
    // check state
    if (simulation.State != Simulation.StateEnum.PLAY &&
    simulation.State != Simulation.StateEnum.NOT_READY)
    {
        SendSimCommand(Command.PLAY, 0);
    }
    else if (simulation.State == Simulation.StateEnum.
    NOT_READY)
    {
        SendSimInitialize((JSONObject)sim_list_editable[
    sim_id]);
        SendSimCommand(Command.PLAY, 0);
    }
}
/// <summary>
/// Pause simulation
/// </summary>
public void Pause()
{
    // check state
    if (simulation.State == Simulation.StateEnum.PAUSE ||
    simulation.State == Simulation.StateEnum.STEP ||
    simulation.State == Simulation.StateEnum.READY) { Step()
    ; return; }
    SendSimCommand(Command.PAUSE, 0);
}
/// <summary>
/// Stop simulation
/// </summary>
public void Stop()
{
    // check state
    if (simulation.State == Simulation.StateEnum.NOT_READY)
    { return; }
```

```
43      SendSimCommand(Command.STOP, 0);
44  }
45
46  /// <summary>
47  /// Change simulation speed
48  /// </summary>
49  public void ChangeSpeed(Simulation.SpeedEnum speed)
50  {
51      // check state
52      if (simulation.Speed == speed) { return; }
53      SendSimCommand(Command.SPEED, (int)speed);
54  }
```

Listing 6.9: Metodi per il controllo remoto della simulazione

Prendendo come esempio il metodo **Pause()** viene controllato lo stato attuale della simulazione per verificare che lo stato che viene richiesto sia compatibile con quello attuale e successivamente con il metodo **SendSimCommand()** viene effettivamente inviata la richiesta al server.

```
1  /// <summary>
2  /// Send sim commands
3  /// </summary>
4  public void SendSimCommand(Command command, int value)
5  {
6      // Create payload
7      JSONObject payload = new JSONObject();
8      payload.Add("command", (int)command);
9      payload.Add("value", value);
10     // Send command
11     UnityEngine.Debug.Log(this.GetType().Name + " | " +
       System.Reflection.MethodBase.GetCurrentMethod().Name + "
       | Sending SIM_COMMAND to MASON...");
12     CommController.SendMessage(nickname, "006", payload);
```

Listing 6.10: Metodo usato per l'invio dei comandi al server

Come si vede nel Listato 6.9 tutti i metodi che richiedono al server un cambio di stato non presentano anche la relativa modifica lato client, infatti per ogni operazione è necessaria la conferma da parte del server che effettivamente la richiesta, ad esempio di pausa, è stata applicata. Alla ricezione del messaggio di conferma viene eseguito lato client il metodo **changeState()**, Listato 6.11.

```
/// <summary>
/// Change simulation state
/// </summary>
private void ChangeState(Command command)
{
    switch (command)
    {
        case Command.STEP:
            simulation.State = Simulation.StateEnum.STEP;
            steps_to_consume++;
            break;
        case Command.PLAY:
            simulation.State = Simulation.StateEnum.PLAY;
            break;
        case Command.PAUSE:
            simulation.State = Simulation.StateEnum.PAUSE;
            break;
        case Command.STOP:
            simulation.State = Simulation.StateEnum.
    NOT_READY;
            CommController.EmptyQueues();
            latestSimStepArrived = 0;
            UIController.UIControllerThreadQueue.Enqueue(()
    => { UIController.step_id.text = "0"; });
            simulation.ClearSimulation();
            SceneController.ClearSimSpace();
            if (admin) UIController.UIControllerThreadQueue.
    Enqueue(() =>
             {
                 UIController.UnlockSimDimensionsButton();
             });
            break;
        case Command.SPEED:
            break;
    }
```

Listing 6.11: Metodo ChangeState() usato per cambiare lo stato della simulazione

Nella sezione dei pulsanti di **Info e statistiche** in Figura 6.4, il primo da sinistra, apre il pannello in cui si possono andare a modificare una serie di parametri della simulazione ma non tutti, come ad esempio la grandezza dell'area di simulazione o il numero di agenti, in generale tutti quei paramentri che, in base alla simulazione, andandoli a cambiare causerebbero inconsistenze nel corretto sviluppo della simulazione nel tempo. Dopo aver modificato i parametri interessati per inviare il messaggio al server è necessario cliccare su "APPLY" e li verrà eseguito il metodo **SendSimUpdate()**, Listato 6.12 il quale contiene oltre ai metodi per l'invio del messagio anche il metodo **StoreUncommittedUpdatesToJSON()**, Listato 6.13, che serve a raccogliere tutte le modifiche che vengono apportate ai parametri oppure ad oggetti e agenti all'interno dello spazio di simulazione per essere mandati in una sola volta in seguito alla conferma tramite il tasto "APPLY", inoltre, serve anche a gestire modifiche multiple sullo stesso ogegtto/parametro per far si che venga inviata, e applicata, al server solo l'ultima modifica effettuata.

```
1 /// <summary>
2 /// Send sim update
3 /// </summary>
4 public void SendSimUpdate()
5 {
6     UnityEngine.Debug.Log(this.GetType().Name + " | " +
      System.Reflection.MethodBase.GetCurrentMethod().Name + "
      | Sending SIM_UPDATE to MASON...");
7     StoreUncommittedUpdatesToJSON();
8     CommController.SendMessage(nickname, "005",
      uncommitted_updatesJSON);
9 }
```

Listing 6.12: Metodo SendSimUpdate() usato per inviare le modifiche ai parametri di simulazione

```
1 private void StoreUncommittedUpdatesToJSON()
2 {
3   string type = "", op = "";
4   foreach (KeyValuePair<(string op, (SimObject.SimObjectType
      type, string class_name, int id) obj), SimObject> entry
      in uncommitted_updates)
5   {
6     JSONObject obj = new JSONObject();
7     JSONNode obj_params = new JSONArray();
8     switch (entry.Key.obj.type)
9     {
10        case SimObject.SimObjectType.AGENT:
11            type = "agents";
```

```
12              break;
13          case SimObject.SimObjectType.GENERIC:
14              type = "generics";
15              break;
16          case SimObject.SimObjectType.OBSTACLE:
17              type = "obstacles";
18              break;
19      }
20      switch (entry.Key.op)
21      {
22          case "MOD":
23              op = "update";
24              break;
25          case "CRT":
26              op = "create";
27              break;
28          case "DEL":
29              op = "delete";
30              break;
31      }
32      if (!entry.Key.op.Equals("CRT")) obj.Add("id", entry.Key
        .obj.id);
33      else obj.Add("quantity", 1);
34      obj.Add("class", entry.Key.obj.class_name);
35      if (!entry.Key.op.Equals("DEL"))
36      {
37          obj_params = (JSONNode)JSON.Parse(JsonConvert.
        SerializeObject(entry.Value.Parameters, new
        TupleConverter<string, float>(), new TupleConverter<
        string, Vector3>(), new TupleConverter<string, bool>(),
        new Vector2IntConverter(), new TupleConverter<string,
        Quaternion>(), new Vec3Conv(), new QuaternionConv()));
38          obj.Add("params", obj_params);
39      }
40      uncommitted_updatesJSON[type + "_" + op].Add(obj);
41    }
42 }
```

Listing 6.13: Metodo StoreUncommittedUpdatesToJSON() usato per raccogliere tutte le modifiche alla simulazione da parte del client

6.3 La classe *Simulation*

Come visto nella Sezione 6.2.1, i messaggi dopo essere arrivati in primo luogo nella **simMessageQueue**, essere stati decompressi, ordinati e inseriti nella **secondaryQueue**, arrivano al punto che devono essere estratti uno alla volta e processati. Processare questi messaggi contenenti gli step di simulazione, vuol dire essere in grado di leggerli e estrarre le informazioni da inserire all'interno di una istanza della classe "Simulation". In questa classe, mostrata nel Listato 6.14, ci sono le strutture dati che rappresentano lo stato di una simulazione, e possiede il metodo più importante dell'intero modulo logico del client Unity ovvero **UpdateSimulationFromStep()**, responsabile della lettura dello step di simulazione e dell'aggiornamento delle strutture dati presenti in questa classe.
La lettura avviene passando tutto il messagggio sotto forma di byte array alla funzione che dovrà prima leggere l'header del messaggio per poter interpretare correttamente il payload poichè il messaggio contente lo step non viene mandato sottoforma di stringa ma direttamente come successione di byte quindi non leggibili senza sapere esattamente la loro struttura.
Le strutture dati principali sono:

agent_class_names, generic_class_names, obstacle_class_names : lista delle classi di agenti, generic[18] e ostacoli presenti nel prototipo di simulazione. Vengono lette nella parte "Header" del messaggio per poter leggere successivamente nel payload i dati nel modo corretto.

n_agents_for_each_class : rappresenta quanti agenti ci sono per ogni classe presente nel prototipo, stessa cosa vale per n_generic_for_each_class e n_obstacles_for_each_class.

Name, description : nome della simulazione e descrizione testuale mostrata a menù in fase di scelta.

agent_prototypes : dizionario contenente i prototipi di ogni tipo di agente, idem per generic_prototype e obstacle_prototype. Un prototipo è la lista dei parametri che caratterizza un oggetto di simulazione.

editableInPlay : lista dei parametri di simulazione che sono modificabili durante il Play, ovvero durante il consumo dei messaggi e la normale visualizzazione della simulazione.

[18] Classe di oggetti generici, ovvero tutti gli oggetti facenti parte della simulazione ma che non sono agenti oppure ostacoli

editableInInit : lista di parametri modificabili prima di entrare nella scena di simulazione quindi, successivamente alla selezione della simulazione dalla lista ma prima della sua inizializzazione, come ad esempio la grandezza o il numero di agenti.

editableInPause : lista di parametri modificabili dopo l'inizializzazione della simulazione non durante l'esecuzione della stessa, è necessario che la simulazione venga messa in pausa dall'admin per poter modificare questi parametri.

SimTypeEnum : enumerator che definisce il tipo di simulazione tra CONTINUOUS, ovvero simulazione in cui le posizione che possono assumere gli oggetti è definita da numeri continui, e DISCRETE invece dove gli oggetti possono assumere coordinate discrete, ad esempio una griglia.

StateEnum : enumerator contenente gli stati in cui si può trovare una simulazione.

agents, generics, obstacles : dizionari in cui vengono salvati rispettivamente gli agenti, i generics e gli obstacles con il relativo id.

```
1  public class StepAppliedEventArgs : EventArgs
2  {
3      public long step_id;
4      public List<string> agent_class_names;
5      public List<string> generic_class_names;
6      public List<string> obstacle_class_names;
7      public int[] n_agents_for_each_class;
8      public int[] n_generics_for_each_class;
9      public int[] n_obstacles_for_each_class;
10 }
11
12
13 public class Simulation
14 {
15     // Events
16     public static event EventHandler<StepAppliedEventArgs>
       OnStepAppliedEventHandler;
17
18     // Simulation data
19     public string name, description;
20     public SimTypeEnum type;
21     public int id;
22     private bool is_discrete;
23     public ConcurrentDictionary<string, int> dimensions =
       new ConcurrentDictionary<string, int>();
```

```
24      public ConcurrentDictionary<string, SimObject>
        agent_prototypes = new ConcurrentDictionary<string,
        SimObject>();
25      public ConcurrentDictionary<string, SimObject>
        generic_prototypes = new ConcurrentDictionary<string,
        SimObject>();
26      public ConcurrentDictionary<string, SimObject>
        obstacle_prototypes = new ConcurrentDictionary<string,
        SimObject>();
27      public ConcurrentDictionary<string, object> parameters =
         new ConcurrentDictionary<string, object>();
28      public List<string> editableInPlay = new List<string>();
29      public List<string> editableInInit = new List<string>();
30      public List<string> editableInPause = new List<string>()
        ;
31
32      public enum SimTypeEnum
33      {
34          CONTINUOUS = 0,
35          DISCRETE = 1
36      }
37      public enum StateEnum
38      {
39          NOT_READY = -1,
40          READY = 0,
41          PLAY = 1,
42          PAUSE = 2,
43          STEP = 3,
44          BUSY = 4
45      }
46      public enum SpeedEnum
47      {
48          X0_25,
49          X0_5,
50          X1,
51          X2,
52          MAX
53      }
54
55      // Runtime data
56      public static StateEnum state = StateEnum.NOT_READY;
57      public SpeedEnum speed = SpeedEnum.X1;
58      public long currentSimStep = 0;
59      public ConcurrentDictionary<string,int>
        n_agents_for_each_class = new ConcurrentDictionary<
        string, int>();
60      public ConcurrentDictionary<string,int>
        n_generics_for_each_class = new ConcurrentDictionary<
        string, int>();
```

```
61      public ConcurrentDictionary<(string class_name, int id),
         SimObject> agents = new ConcurrentDictionary<(string,
        int), SimObject>();
62      public ConcurrentDictionary<(string class_name, int id),
         SimObject> generics = new ConcurrentDictionary<(string,
         int), SimObject>();
63      public ConcurrentDictionary<(string class_name, int id),
         SimObject> obstacles = new ConcurrentDictionary<(string
        , int), SimObject>();
64      private List<(SimObject.SimObjectType type, string
        class_name, int id)> toDeleteIfAbsent = new List<(
        SimObject.SimObjectType type, string class_name, int id)
        >();
65      private List<(SimObject.SimObjectType type, string
        class_name, int id)> toKeepIfAbsent = new List<(
        SimObject.SimObjectType type, string class_name, int id)
        >();
66      private List<(SimObject.SimObjectType type, string
        class_name, int id)> temp = new List<(SimObject.
        SimObjectType type, string class_name, int id)>();
67      private List<(SimObject.SimObjectType type, string
        class_name, int id)> temp2 = new List<(SimObject.
        SimObjectType type, string class_name, int id)>();
68      private SimObject defaultSimObject;
69
70      }
```

Listing 6.14: Strutture dati della classe Simulation

Passando al metodo **UpdateSimulationFromStep()** verranno descritte le sezioni più importanti, andando ad analizzarne le sezioni principali.
Nel Listato 6.15 l'array di byte che viene passato al metodo in primo luogo viene decompresso, e viene letta la prima locazione di memoria che comprende il booleano "complete" ovvero un valore di **0** o **1** a seconda se lo step che sta per leggere contiene tutte le informazione degli oggetti di simulazione in modo esplicito oppure no. Questo viene fatto per questioni di ottimizzazione della memoria e dalla banda occupata dal messaggio. Passaggio successivo è quello di leggere la lista dei prototipi per gli agenti, gli oggetti generici e gli ostacoli, per far si che successivamente alla lettura, ad esempio degli agenti, si conosce la quantità di parametri e la loro tipologia. Stessa cosa va fatta per la lista dei parametri per ogni singola classi di oggetti di simulazione, come mostrato dalla riga 59.

```
1  public void UpdateSimulationFromStep(byte[] step, JSONObject
       sim_prototype){
2      // variabili
3
4      byte[] decompressed_step = Utils.DecompressStepPayload(
       step);
5  ...
6
7      // creo stream e reader per leggere lo step
8      MemoryStream deserialize_inputStream = new MemoryStream(
       decompressed_step);
9      BinaryReader deserialize_binaryReader = new BinaryReader
       (deserialize_inputStream);
10
11     // estraggo la flag 'complete'
12     bool complete = BitConverter.ToBoolean(
       deserialize_binaryReader.ReadBytes(1).Reverse().ToArray
       (), 0);
13     agent_prototypes = ((JSONArray)sim_prototype["
       agent_prototypes"]).Linq.ToList();
14     generic_prototypes = ((JSONArray)sim_prototype["
       generic_prototypes"]).Linq.ToList();
15     obstacle_prototypes = ((JSONArray)sim_prototype["
       obstacle_prototypes"]).Linq.ToList();
16
17
18     List<string> agent_class_names = new Func<List<string
       >>(() => {
19         List<string> array = new List<string>();
20         foreach (JSONObject p in agent_prototypes)
21         {
22             array.Add(p["class"]);
23         }
```

```
24          return array;
25      })();
                                        // nomi delle classi
    degli agenti
26      List<string> generic_class_names = new Func<List<string
    >>(() => {
27          List<string> array = new List<string>();
28          foreach (JSONObject p in generic_prototypes)
29          {
30              array.Add(p["class"]);
31          }
32          return array;
33      })();
34      // nomi delle classi degli oggetti
35      List<string> obstacle_class_names = new Func<List<string
    >>(() => {
36          List<string> array = new List<string>();
37          foreach (JSONObject p in obstacle_prototypes)
38          {
39              array.Add(p["class"]);
40          }
41          return array;
42      })();
43
44      // nomi delle classi degli ostacoli
45      List<List<(string, string)>> agent_params_for_each_class
     = new Func<List<List<(string, string)>>>(() => {
46              List<List<(string, string)>> array = new List<
    List<(string, string)>>();
47              for (int i = 0; i < agent_class_names.Count; i
    ++)
48              {
49                  JSONObject c = (JSONObject)agent_prototypes.
    ElementAt(i);
50                  array.Add(new List<(string, string)>());
51                  foreach (JSONObject p in ((JSONArray)c["
    params"]))
52                  {
53                      array[i].Add((p["name"], p["type"]));
54                  }
55              }
56              return array;
57          })();
58
59
60      List<List<(string, string)>>
    generic_params_for_each_class = new Func<List<List<(
    string, string)>>>(() =>
61          {
```

```
62              List<List<(string, string)>> array = new List<
    List<(string, string)>>();
63              for (int i = 0; i < generic_class_names.Count; i
    ++)
64              {
65                  JSONObject c = (JSONObject)
    generic_prototypes.ElementAt(i);
66                  array.Add(new List<(string, string)>());
67                  foreach (JSONObject p in ((JSONArray)c["
    params"]))
68                  {
69                      array[i].Add((p["name"], p["type"]));
70                  }
71              }
72              return array;
73          })()
74
75      // (nome_param, tipo_param) per ogni parametro per ogni
    classe di oggetto
76      List<List<(string, string)>>
    obstacle_params_for_each_class = new Func<List<List<(
    string, string)>>>(() =>
77      {
78          List<List<(string, string)>> array = new List<List<(
    string, string)>>();
79          for (int i = 0; i < obstacle_class_names.Count; i++)
80          {
81              JSONObject c = (JSONObject)obstacle_prototypes.
    ElementAt(i);
82              array.Add(new List<(string, string)>());
83              foreach (JSONObject p in ((JSONArray)c["params"
    ]))
84              {
85                  array[i].Add((p["name"], p["type"]));
86              }
87          }
88          return array;
89      })();
90  }
```

Listing 6.15: Metodo UpdateSimulationFromStep(), lettura dell'header

Dopo aver letto l'header, parte iniziale di un messaggio di simulazione, si hanno a disposizione tutte le informazioni necessarie per poter leggere la restante parte, molto più grande in termini di dimensione e soprattutto dinamica.

```
1
2 // estraggo l'ID
3 CurrentSimStep = BitConverter.ToInt64(
      deserialize_binaryReader.ReadBytes(8).Reverse().ToArray
      (), 0);
4 // estraggo il numero di agenti per classe presenti nello
      step
5 for (int i = 0; i < n_agents_classes; i++)
6 {
7     n_agents_for_each_class[i] = BitConverter.ToInt32(
      deserialize_binaryReader.ReadBytes(4).Reverse().ToArray
      (), 0);
8 }
9 // estraggo il numero di oggetti per classe presenti nello
      step
10 for (int i = 0; i < n_generics_classes; i++)
11 {
12     n_generics_for_each_class[i] = BitConverter.ToInt32(
      deserialize_binaryReader.ReadBytes(4).Reverse().ToArray
      (), 0);
13 }
14 // estraggo il numero di ostacoli per classe presenti nello
      step
15 for (int i = 0; i < n_obstacles_classes; i++)
16 {
17     n_obstacles_for_each_class[i] = BitConverter.ToInt32(
      deserialize_binaryReader.ReadBytes(4).Reverse().ToArray
      (), 0);
18 }
19 // AGENTI
20 // estraggo ogni agente per classe
21 for (int i = 0; i < n_agents_classes; i++)
22 {
23     int n_agents_of_a_class = n_agents_for_each_class[i];
24     if (n_agents_of_a_class == 0) { continue; }
25
26     for (int j = 0; j < n_agents_of_a_class; j++)
27     {
28
29     //estraggo l'id dell'agente
30     //l'agente nuovo quindi dobbiamo crearlo e leggere tutti
       i parametri anche quelli non dynamic
31     //estraggo le coordinate di ogni agente
32
```

```
33      //estraggo i paramtri (coordinate spaziali) in base alla
        tipologia
34
35      case "System.Single":
36          // aggiorno i parametri dell'oggetto di simulazione
37          break;
38      case "System.Int32":
39          // aggiorno i parametri dell'oggetto di simulazione
40          break;
41      case "System.Boolean":
42          // aggiorno i parametri dell'oggetto di simulazione
43          break;
44      case "System.String":
45          // aggiorno i parametri dell'oggetto di simulazione
46          break;
47      }
48  }
```

Listing 6.16: Metodo UpdateSimulationFromStep(), lettura degli agenti

Successivamente alla lettura degli agenti vengono letti gli oggetti. Una delle differenze rispetto agli agenti è che un oggetto può anche occupare più spazi all'interno dello spazio di simulazione, e dato che possono avere forme irregolari è necessario conoscere le "celle"[19] che andranno ad occupare.

```
1  // OGGETTI
2  // estraggo ogni oggetto per classe
3  for (int i = 0; i < n_generics_for_each_class.Length; i++)
4  {
5      int n_generics_of_a_class = n_generics_for_each_class[i
       ];
6
7      if (n_generics_of_a_class == 0) { continue; }
8      for (int j = 0; j < n_generics_of_a_class; j++)
9      {
10         int id = BitConverter.ToInt32(
       deserialize_binaryReader.ReadBytes(4).Reverse().ToArray
       (), 0);
11
12         parameters = complete ?
       generic_params_for_each_class : BitConverter.ToBoolean(
       deserialize_binaryReader.ReadBytes(1).Reverse().ToArray
       (), 0) ? generic_params_for_each_class :
       d_generic_params_for_each_class;
13
14         if (!Generics.TryGetValue((generic_class_names[i],
       id), out so))
```

[19] Al momento dell'implementazione gli oggetti sono supportati solo in alcune simulazioni

```
15
16          // oggetto nuovo quindi dobbiamo crearlo e leggere
        tutti i parametri anche quelli non object
17          {
18              Generic_prototypes.TryGetValue(
        generic_class_names[i], out SimObject g);
19
20              so = g.Clone();
21              so.Type = SimObject.SimObjectType.GENERIC;
22              so.Class_name = generic_class_names[i];
23              so.Id = id;
24              AddGeneric(so);
25          }
26
27          so.present = true;
28
29          // per tutti i parametri degli oggetti
30          foreach ((string, string) p in parameters[i])
31          {
32              switch (p.Item2)
33              {
34                  case "System.Position":
35                      ...
36                      //leggo le coordinate
37                      ...
38                      so.UpdateParameter("position", value);
39                      break;
40                  case "System.Rotation":
41                      ...
42                      //leggo la rotazione
43                      ...
44                      so.UpdateParameter("rotation", value);
45                      break;
46                  case "System.Cells":
47                      ...
48                      //leggo le celle che occupa
49                      ...
50                      so.UpdateParameter("position", value);
51                      break;
52                  case "System.Single":
53                      //leggo i parametri
54                      break;
55                  case "System.Int32":
56                      //leggo i parametri
57                      break;
58                  case "System.Boolean":
59                      //leggo i parametri
60                      break;
61                  case "System.String":
```

```
62                          //leggo i parametri
63                          break;
64                  }
65              }
66          }
67  }
```

Listing 6.17: Metodo UpdateSimulationFromStep(), lettura degli oggetti

Durante una simulazione è possibile che alcuni oggetti vengano eliminati oppure non debbano essere più aggiornati, questa evenienza viene causata sia dalla natura stessa di un modello di simulazione sia dal fatto che lato Unity è possibile elminare alcuni oggetti. Questo ha portato ad implemetare un'ottimizzazione della gestione della memoria andando ad eliminare, ad ogni step arrivato, tutti quegli oggetti di cui il server non ha mandato aggiornamenti di stato, considerandoli così implicitamente eliminati e non più necessario aggiornarli.

```
1  // Elimina SimObject non presenti
2  Agents.ToList().ForEach((entry) => {
3      if (!entry.Value.present && entry.Value.Is_in_step)
4      {
5          if (!entry.Value.To_keep_if_absent) entry.Value.
       toDelete = true;
6          else
7          {
8              string param = (string)agent_prototypes.Where((p
       ) => p.Value["class"].Equals(entry.Value.Class_name)).
       ToArray()[0].Value["state_if_absent"];
9              entry.Value.Parameters[param] = true;
10         }
11     }
12     entry.Value.present = false;
13 });
14 Generics.ToList().ForEach((entry) => {
15     if (!entry.Value.present && entry.Value.Is_in_step)
16     {
17         if (!entry.Value.To_keep_if_absent) entry.Value.
       toDelete = true;
18         else
19         {
20             string param = (string)generic_prototypes.Where
       ((p) => p.Value["class"].Equals(entry.Value.Class_name))
       .ToArray()[0].Value["state_if_absent"];
21             entry.Value.Parameters[param] = true;
22         }
23     }
```

```
24      entry.Value.present = false;
25  });
26  Obstacles.ToList().ForEach((entry) => {
27      if (!entry.Value.present && entry.Value.Is_in_step)
28      {
29          if (!entry.Value.To_keep_if_absent) entry.Value.
        toDelete = true;
30          else
31          {
32              string param = (string)obstacle_prototypes.Where
        ((p) => p.Value["class"].Equals(entry.Value.Class_name))
        .ToArray()[0].Value["state_if_absent"];
33              entry.Value.Parameters[param] = true;
34          }
35      }
36      entry.Value.present = false;
37  });
38
39  StepAppliedEventArgs e = new StepAppliedEventArgs();
40  e.step_id = CurrentSimStep;
41  e.agent_class_names = agent_class_names;
42  e.generic_class_names = generic_class_names;
43  e.obstacle_class_names = obstacle_class_names;
44  e.n_agents_for_each_class = n_agents_for_each_class;
45  e.n_generics_for_each_class = n_generics_for_each_class;
46  e.n_obstacles_for_each_class = n_obstacles_for_each_class;
47  OnStepAppliedEventHandler?.Invoke(this, e);
48
49  if (complete) UnityEngine.Debug.Log("SIMULATION UPDATED FROM
        STEP " + currentSimStep + " : e " + (complete ? "
        completo" : "non completo."));
```

Listing 6.18: Metodo UpdateSimulationFromStep(), eliminazione degli oggetti di simulazione non più esistenti

Capitolo 7

Performance manager

L'ultima macro componente dell'architettura client è il **Performance Manager**. Questo modulo si occupa di monitorare le prestazioni del client e regolare l'afflusso dei messaggi dal broker verso di esso. Riprendendo l'architettura generale mostrata nel capitolo 4 in Figura 4.1, abbiamo mostrato come i messaggi prodotti dal server di simulazione vengano spalmati su 60 topic, al quale i client si sottoscrivono per riceverne i messaggi. Di default i client si iscrivono a tutti i 60 topic, ma c'è anche la possibilità di iscriversi solo ad un sottoinsieme purchè in ogni sottoinsieme ci si iscriva anche topic 0[20], ritenuto essenziale. Questa possibilità permette di potere ricevere meno messaggi rispetto a quanti ne vengono prodotti andando così ad alleggerire il carico sull'hardware e dare la possibilità a client meno performanti di poter usufruire dell'applicazione e rimanere al passo anche di client più prestazionali che riescono a consumare tutti i messaggi in arrivo.
Questa soluzione però può causare dei problemi, in alcuni tipi di simulazione, tra cui:

Perdita di informazioni importanti : all'interno dei messaggi anche la mancanza di informazione viene sfruttata come informazione nella procedura di eliminazione degli oggetti eliminati, quindi la perdita di alcuni messaggi potrebbe voler dire perdere l'informazione dell'eliminazione di un oggetto.

Riproduzione non fluida della simulazione : andando a disiscrivere il client da alcuni topic è possibile che si creino dei "buchi" in cui non arrivano i messaggi. Facendo un esempio, se a partire dai 60 topic, se viene rimossa l'iscrizione ai primi o ultimi 30 topic andremo a dimezzare

[20] I topic sono numerati dal numero 0 al 59

i messaggi in arrivo, ma in questo modo, se i 60 messaggi corrispondono ad un secondo di simulazione, allora il client riceverà messaggi solo dai 30 topic che significherebbe visualizzare solo la prima metà del secondo di simulazione prodotto, perdendo l'altra e risultando così in una visualizzazione non fluida.

Per risolvere il problema della **Perdita di informazioni importanti** si ricorre all'obbligatorietà della sottoscrizione al topic 0 in modo tale da ricevere comunque un aggiornamento ed essere sicuri di rimuovere oggetti non più necessari, mentre per il problema della fluidità è necessario introdurre le strutture dati e le variabili nel Listato 7.1 e le funzioni nel Listato 7.2.

```
1  public class PerformanceManager
2  {
3      /// Load Balancing
4      public static int SORTING_THRESHOLD = 15;
5      public static int MAX_SPS = 60;
6      public static int MIN_SPS = 8;
7      public int produced_sps = 60;
8      public int received_sps = 60;
9      public int consumed_sps = 60;
10     public int attempted_sps = 60;
11     public double consume_rate = 1d;
12     public int[] topics = new int[MAX_SPS];
13
14     /// Time related vars
15     public long timeout_target_up = 0;
16     public long timeout_target_down = 0;
17     public long timestamp_last_update = 0;
18
19     public int[] TOPICS { get => topics; set => topics =
       value; }
20     public int PRODUCED_SPS { get => produced_sps; set =>
       produced_sps = value; }
21     public int RECEIVED_SPS { get => received_sps; set =>
       received_sps = value; }
22     public int CONSUMED_SPS { get => consumed_sps; set =>
       consumed_sps = value; }
23     public int ATTEMPTED_SPS { get => attempted_sps; set =>
       attempted_sps = value; }
24     public double CONSUME_RATE { get => consume_rate; set =>
        consume_rate = value; }
25     public long TIMEOUT_TARGET_UP { get => timeout_target_up
       ; set => timeout_target_up = value; }
26     public long TIMEOUT_TARGET_DOWN { get =>
       timeout_target_down; set => timeout_target_down = value;
        }
```

```
27      public long TIMESTAMP_LAST_UPDATE { get =>
        timestamp_last_update; set => timestamp_last_update =
        value; }
28  }
```

Listing 7.1: Strutture dati e variabili del Performance Manager

I dati principali su cui si basa il Performance Manager sono i topic e la quantità dei messaggi all'interno delle code **simMessageQueue** e **secondaryQueue**. Le variabili e strutture dati della classe sono:

SORTING_THRESHOLD : soglia di messaggi all'interno della **secondaryQueue** dopo il quale inizia il consumo di messaggi.

MAX_SPS : numero massimo di topic a cui iscriversi, nella nostra implementazione è 60.

MIN_SPS : numero minimo di topic a cui iscriversi, nella nostra implementazione è 8.

statistiche riguardo i messaggi :

- **produced_sps**: messaggi prodotti al secondo dal server dal punto di vista del client.
- **received_sps**: messaggi al secondo ricevuti.
- **consumed_sps**: messaggi consumati al secondo dal client.
- **attempted_sps**: dopo il cambiamento al numero di topic a cui iscriversi serve calcolare la distribuzione migliore per evitare i "buchi", attempted_sps è il numero usato per calcolarli.

consume_rate : rapporto tra i messaggi ricevuti e i messaggi consumati.

topics : array contente l'id di tutti i topics a cui iscriversi.

Variabili di timeout : servono per definire il tempo dopo il quale in caso di miglioramento o peggioramento delle prestazioni ci si iscrive a più o meno topic.

In riferimento al Listato 7.2 andremo ad illustrare i metodi del Performance Manager che sono responsabili del monitoraggio delle prestazioni del client.

CalculatePerformance() : funzione principale, si occupa del monitoraggio vero e proprio delle code ed esegue le seguenti operazioni

- prende in input il Communication Controller per poter usare il simClient e modificare la quantità di topic a cui si deve iscrivere, lo stato di simulazione, e le variabili **stepsConsumed**, **stepsReceived**.
- essendo una funzione che deve essere sempre in esecuzione è presente un ciclo while infinito, all'interno del quale viene eseguito il calcolo solo se la simulazione si trova nello stato PLAY
- per prima cosa si calcola il rapporto tra i messaggi consumati e ricevuti in riga 10 tramite la funzione **CalculateConsumeRate()**
- in riga 14 avviene il controllo sull'andamento del client, ovvero se in quel determinato momento riesce a stare al passo con il numero di messaggi in arrivo (CONSUME_RATE $\geq 1d$) e la somma dei messaggi nelle code è maggiore del doppio della SORTING_THRESHOLD (scelta empiricamente durante i test) allora vorrà dire che il client riesce a consumare i messaggi quindi se non si è già iscritti a tutti i messaggi, si andrà ad aumentare il numero dei topic iscritti (riga 16) e a ricalcolare a quale tra i 60 topic iscriversi con la funzione **CalculateTargetTopics()**
- se invece il client non riesce ad essere al passo con il numero di messaggi al secondo in arrivo (riga 20), bisogna diminuire i topic a cui si è iscritti. Tutto ciò è controllato da timestamp per far si che nel caso in cui il client sia in difficoltà si reagisce subito diminuendo i topic, metre nel caso in cui il client riesce a gestire i messaggi e cerca di iscriversi a più topic, questo timeout sarà più grande per evitare che sia solo un momento in cui il client sembri stia migliorando quando magari in realtà l'apparenza di buone prestazione è solo un rallentamento dell'arrivo dei messaggi probabilmente per problemi di rete o lato server.

CalculateConsumeRate() : questa funzione calcola semplicemente il rapporto tra i messaggi consumati e i messaggi ricevuti, utile a capire se la tendenza del client è quella di accumulare i messaggi o di essere al passo e riuscirli a processare tutti.

CalculateTargetTopics() : nel momento che, ad esempio, si deve passare da 60 topic a 40 è necessario calcolare come distribuire più equamente possibile questi 40 topic a cui iscriversi sui 60 topic a disposizione.

```
public void CalculatePerformance(CommunicationController
    CommController, ref Simulation.StateEnum state, ref int
    stepsConsumed, ref int stepsReceived)
    {
        System.Diagnostics.Stopwatch stopwatch = new System.
    Diagnostics.Stopwatch();
        stopwatch.Start();

        while (true)
        {
            if (state == Simulation.StateEnum.PLAY)
            {
                CalculateConsumeRate(ref stepsConsumed, ref
    stepsReceived);
                stopwatch.Stop();
                timestamp_last_update = stopwatch.
    ElapsedMilliseconds;
                stopwatch.Start();
                if (CONSUME_RATE >= 1d &&
    timestamp_last_update > TIMEOUT_TARGET_UP && !(
    CommController.SimMessageQueue.Count + CommController.
    SecondaryQueue.Count >= 2 * SORTING_THRESHOLD))
                {
                    ATTEMPTED_SPS = (CommController.
    SimMessageQueue.Count == 0) ? Math.Min(TOPICS.Length +
    2, MAX_SPS) : Math.Min(TOPICS.Length + 1, MAX_SPS);
                    CalculateTargetTopics();
                    CommController.SubscribeOnly(TOPICS);
                }
                else if (CONSUME_RATE < 1d &&
    timestamp_last_update > TIMEOUT_TARGET_DOWN)
                {
                    ATTEMPTED_SPS = (CommController.
    SimMessageQueue.Count + CommController.SecondaryQueue.
    Count >= 2*SORTING_THRESHOLD) ? Math.Max(TOPICS.Length -
     4, MIN_SPS) : Math.Max(TOPICS.Length - 2, MIN_SPS);
                    CalculateTargetTopics();
                    CommController.SubscribeOnly(TOPICS);
                }
            }
        }
    }
    public void CalculateConsumeRate(ref int stepsConsumed,
    ref int stepsReceived)
    {
        stepsConsumed = 0;
        stepsReceived = 0;
        Thread.Sleep(500);
        RECEIVED_SPS = stepsReceived * 2;
```

```
35          CONSUMED_SPS = stepsConsumed * 2;
36          CONSUME_RATE = RECEIVED_SPS > 0d ? (CONSUMED_SPS / (
        double)RECEIVED_SPS) : 0d;
37      }
38      public void CalculateTargetTopics()
39      {
40          TOPICS = new int[ATTEMPTED_SPS];
41          bool zero = false;
42
43          for (int t = 1; t <= ATTEMPTED_SPS; t++)
44          {
45              int topic = t * MAX_SPS / (ATTEMPTED_SPS + 1);
46              if (topic == 0) zero = true;
47              topics[t - 1] = topic;
48          }
49          if (zero) TOPICS = TOPICS.Skip(1).ToArray();
50      }
```

Listing 7.2: Classe Performance Manager

In conclusione il Performance Manager agisce in background su un thread a se monitorando costantemente le prestazioni, calcolando il rapporto tra messaggi consumati e ricevuti e regolando la quantità di topic a cui bisogna iscriversi sia nel caso in cui un client sia in difficoltà e quindi è necessario ridurre questo numero, oppure anche in caso di un miglioramento di prestazione durante la visualizzazione e quindi anche la possibilità di recuperare i topic a cui si era disiscritto in precedenza. Questa implementazione è chiave dal punto di vista dell'esperienza collaborativa poichè livella l'esperienza di visualizzazione attraverso hardware differenti facendo in modo che anche i dispositivi meno prestanti possano essere al passo della simulazione e degli altri client riuscendo a visualizzare in contemporanea la stessa scena dal proprio punto di vista.

Capitolo 8

Conclusioni e sviluppi futuri

Questo progetto di tesi sviluppato in ISISLab da me, Pietro Russo e Gerardo Barone aveva come obiettivo comune quello di creare un'infrastruttura client server in grado si visualizzare i modelli di simulazioni ad agenti che da decenni sono presenti in letteratura ma che non hanno mai avuto spazio nel mondo della grafica tridimensionale. Nello specifico è stata raggiunto l'obiettivo dell'interazione, con la possibilità di modificare la simulazione in modo dinamico, della visualizzazione, migliorando graficamente lo spazio in cui si muovono gli agenti, ed infine la collaboratività, grazie alla quale è possibile visualizzare insieme ad altri utenti la stessa simulazione ma dal proprio punto di vista potendo anche modificarla condividendo queste modifiche con altri utenti. La possibilità di interagire con le ABM è un elemento cruciale per massimizzare la personalizzazione delle simulazioni e renderle più fruibili e dinamiche. In questo lavoro di tesi, abbiamo presentato oltre le singole componenti anche tutta l'architettura client-server comprensiva di un server, un broker di messaggi e un client, con il focus sulla componente logica implementata con il motore grafico Unity. Il lavoro svolto in questa tesi può aprire molte opportunità per sviluppi futuri. In primo luogo, è possibile aggiungere ulteriori funzionalità per migliorare gestione della simulazione, come la modifica più approfondita dei parametri del modello e l'implementazione di sistema completo di lobby come succede per i videogiochi online. Inoltre, potrebbe essere interessante esplorare la possibilità di generalizzare ancor più la gestione delle simulazione per integrare il modulo con altri motori di simulazione, al fine di fornire una soluzione più versatile e adattabile. Un altro sviluppo potenziale riguarda la trasposizione dell'architettura su provider di cloud computing come Amazon AWS e Microsoft Azure per sfruttarne la modularità e l'ampia offerta di servizi. In definitiva, il lavoro svolto in

questa tesi ha dimostrato come l'interazione con le ABM e l'integrazione con il comparto grafico 3D possano migliorare notevolmente la fruibilità delle simulazioni ad agenti. Siamo convinti che questo lavoro possa aprire molte opportunità per sviluppi futuri nel campo delle simulazioni ad agenti, migliorando la loro utilità e applicabilità in vari settori, dall'educazione alla ricerca scientifica.

Bibliografia

[1] *.NET*. URL: https://dotnet.microsoft.com/en-us/.

[2] *Advances and Techniques for Building 3D Agent-Based Models for Urban Systems*.
URL: https://www.researchgate.net/publication/267636855_Advances_and_Techniques_for_Building_3D_Agent-Based_Models_for_Urban_Systems.

[3] *Autodesk 3DS Max website*.
URL: https://www.autodesk.it/products/3ds-max/.

[4] *Awake method Unity docs*. URL: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Awake.html.

[5] Kostas Cheliotis. ≪ABMU 3D Simulation≫. In: (2020).

[6] *Computer-Aided Design definition*.
URL: https://www.techopedia.com/definition/2063/computer-aided-design-cad.

[7] *Eclipse Mosquitto MQTT Broker*. URL: https://mosquitto.org/.

[8] *Geographic Information System definition*.
URL: https://www.esri.com/en-us/what-is-gis/overview.

[9] *George Mason University Website*. URL: https://www.gmu.edu/.

[10] Camille El-Habr et al. ≪Runner≫. In: (2019). URL: https://www.sciencedirect.com/science/article/pii/S235271101930024X.

[11] Spencer J. Ingley et al. ≪anyFish 2.0≫. In: (2015). URL: https://www.sciencedirect.com/science/article/pii/S2352711015000114.

[12] *Internet of Things*. URL:
https://www.oracle.com/internet-of-things/what-is-iot/.

[13] *ISISLab Website*. URL: https://www.isislab.it.

[14] Arthur Juliani et al.
≪Unity: A General Platform for Intelligent Agents≫. In: (2020).
URL: https://arxiv.org/abs/1809.02627.

[15] Brandon K.Horton et al. ≪GEARS≫. In: (2019). URL: https://www.sciencedirect.com/science/article/pii/S2352711018300633.

[16] José L.Soler-Domínguez, Manuel Contero e Mariano Alcañiz.
≪Workflow and tools to track and visualize behavioural data from a Virtual Reality environment using a lightweight GIS≫. In: (2019).
URL: https://www.sciencedirect.com/science/article/pii/S2352711018300931.

[17] *MASON Website*.
URL: https://cs.gmu.edu/~eclab/projects/mason/.

[18] *MQTT Protocol Website*. URL: https://mqtt.org/.

[19] *NetLogo*. URL: https://ccl.northwestern.edu/netlogo/.

[20] *Replicating capacity and congestion in microscale agent-based simulations*. URL: https://www.sciencedirect.com/science/article/pii/S2214367X22000771.

[21] Pietro Russo. ≪Simulazioni ABM in Unity3D: Un framework di integrazione per simulazioni MASON≫.

[22] *Scene Unity component docs*.
URL: https://docs.unity3d.com/Manual/CreatingScenes.html.

[23] *Unity3D Asset Store*. URL: https://assetstore.unity.com/.

[24] *Unity3D Physics Solutions*.
URL: https://unity.com/solutions/programming-physics.

[25] *Unity3D Website*. URL: https://unity.com/.

[26] *What is ABM*. URL: https://sites.google.com/view/3d-science-abm/home/what-is-abm.

Elenco delle figure

Listings