Politecnico di Torino

Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Informatica
A.a. 2023/2024
Sessione di Laurea Luglio, 2024

Sviluppo di un'infrastruttura per la gestione delle simulazioni di oggetti 3D in Astra Data Navigator

Relatori:
Prof. Andrea SANNA
Eugenio TOPA

Candidato:
Fabio MARINARO

Ringraziamenti

Giunto al termine del percorso di studi intrapreso, vorrei prendere un attimo per pensare a ciò che questo ha significato per me e ringraziare le persone che mi hanno sostenuto e reso tutto questo possibile.

Posso dire di me stesso, negli anni del liceo, di essere stato uno studente svogliato, in generale poco dedito allo studio e con una particolare avversione verso materie come la matematica, la fisica e tra tutte l'inglese, tanto da essere stato anche bocciato per queste. Una laurea in ingegneria poteva sembrare sicuramente poco affine ed estremamente difficile da ottenere.

Vorrei quindi iniziare ringraziando i miei genitori che, nonostante le premesse poco rassicuranti e nonostante le difficoltà di quel periodo, hanno deciso comunque di permettermi di frequentare l'università e spostarmi dall'altra parte dell'Italia. Un grazie va anche ai miei familiari, compresi quelli che non conoscevo prima del mio arrivo a Torino.

Un ringraziamento va poi ai miei amici di Fragagnano, con cui ho trascorso tutti gli anni della mia permanenza a Torino, in particolare i primi. Insieme a loro vorrei ringraziare anche i miei colleghi Federico, Kevin e Giulia, che nel corso degli anni si sono rivelati essere degli amici.

Inoltre, un grazie è dovuto a tutti i professori che durante il corso degli anni di studio hanno saputo trasmettermi la passione e l'interesse verso la materia. In particolar modo ringrazio il professor Andrea Sanna, che ho avuto la fortuna di avere come docente nel corso di Computer Animation ed ora come relatore, per la sua disponibilità ed affidabilità.

Ringrazio poi ALTEC e tutte le persone che mi hanno accolto, in particolare Eugenio Topa, sempre disponibile a fornire supporto e grazie ai cui consigli credo di essere maturato professionalmente. Grazie anche a Giuseppe, Bianca e Marta, con i quali ho passato le mie giornate nella sala tesisti di ALTEC.

Infine, grazie a Francesca, che ha reso questi ultimi anni altrimenti fonte di malinconia più sopportabili.

Sommario

La Realtà Virtuale, o Virtual Reality (VR), è una tecnologia oggi conosciuta ed ampiamente sfruttata in vari contesti applicativi, nonostante un percorso travagliato dovuto principalmente alla mancanza di hardware adeguato e strumenti software a supporto dello sviluppo delle applicazioni. Partita dai laboratori universitari negli anni '60, è arrivata oggi ad essere utilizzata a scopo di intrattenimento, ma anche educativo, oltre che per avere rappresentazioni tridimensionali di simulazioni scientifiche di vario tipo ed altro ancora. Avere a disposizione una rappresentazione tridimensionale dei dati ne facilita la comprensione. In ambito educativo fare esperienza diretta di alcuni argomenti favorisce l'apprendimento, che rimarrebbe altrimenti limitato ad una semplice lettura spesso fine a se stessa.

Un altro ambito applicativo di questa tecnologia è quello aerospaziale, dove viene sfruttata per l'addestramento degli astronauti. Questa attività può risultare molto costosa ed oltretutto molto pericolosa. Può inoltre essere quasi impossibile replicare le condizioni che gli astronauti dovranno affrontare nello spazio sulla Terra. È proprio in situazioni come questa che la VR può essere sfruttata a pieno. Questa permette di emulare in maniera verosimile mondi, situazioni e condizioni che sarebbero altrimenti inaccessibili normalmente, come lo spazio esterno.

Gli strumenti software che più di tutti hanno permesso lo sviluppo recente della VR sono sicuramente i game engine, inizialmente nati per essere utilizzati nello sviluppo di videogiochi. Un game engine è un software che permette di facilitare il lavoro dello sviluppatore occupandosi dell'integrazione delle varie librerie di basso livello. Seppure siano nati per supportare lo sviluppo di videogiochi, risultano essere l'alternativa più semplice ed immediata per la realizzazione di applicazioni in VR di qualsiasi tipo. Applicazioni per la simulazione, in ambito educativo ed anche di altro tipo beneficiano immensamente di questi programmi. Lo sviluppo di strumenti di questo tipo richiede solitamente tempo ed elevate competenze di programmazione. Per questo motivo spesso si ricorre all'utilizzo di prodotti di terze parti. Ne sono un esempio i noti game engine *Unity* ed *Unreal Engine*, i quali rappresentano lo stato dell'arte in questo campo.

Una caratteristica molto interessante di questo tipo di software è la presenza di un physics engine integrato, il quale permette di simulare l'interazione fisica in tempo reale. Anche lo sviluppo di un software per i calcoli fisici richiede grande

impegno, per questo spesso si preferisce integrare un prodotto di terze parti. Un esempio tra tutti è *PhysX*, sviluppato da *NVIDIA* ed ampiamente utilizzato ed integrato su diverse piattaforme. Bisogna tenere presente che i programmi per la simulazione necessitano di molte risorse e molto tempo per effettuare calcoli fisici accurati, rendendoli non adatti per una simulazione in tempo reale. Per questo motivo la presenza dei physics engine ha generato molto interesse, tanto che nell'ultimo decennio sono stati svariati i lavori di ricerca che hanno tentato di sfruttare i game engine per implementare delle simulazioni fisiche in tempo reale.

Il lavoro di tesi si colloca nel contesto della VR, delle simulazioni fisiche e dei game engine. Fa uso del software *Blender* per la modellazione 3D, del game engine *Unity* per la realizzazione dell'applicativo finale, Astra Data Navigator (ADN), e del linguaggio di scripting *C#*, con la libreria *.NET* per l'implementazione della logica degli elementi del mondo virtuale. Il physics engine integrato nella versione di Unity utilizzata per il lavoro di tesi è PhysX.

L'obiettivo del lavoro è quello di estendere le funzionalità già presenti nel software ADN, permettendo all'applicazione di simulare il comportamento di oggetti di vario tipo all'interno del mondo virtuale. Inoltre, parallelamente all'estensione delle funzionalità, si vuole osservare la qualità della simulazione fisica che Unity è in grado di fornire.

Per raggiungere lo scopo prefissato è stata creata l'infrastruttura necessaria per la gestione degli elementi del mondo virtuale e l'emulazione del loro comportamento. A scopo dimostrativo è stato creato un server esterno responsabile dell'invio dei comandi da far eseguire ai singoli elementi del mondo virtuale e sono stati introdotti nell'applicazione 2 oggetti: il satellite orbitale Trace Gas Orbiter (TGO) ed il rover Rosalind Franklin. Mentre il primo è servito unicamente per dimostrare la capacità del sistema di gestire oggetti di vario tipo, il secondo è stato sfruttato al fine di verificare la qualità della simulazione fisica raggiungibile da Unity. Il rover infatti necessita dell'implementazione di una serie di funzionalità che possono sfruttare il motore fisico, come ad esempio quelle di movimento sul terreno. Terreno che rappresenta dati altimetrici reali del pianeta cui fa riferimento, Marte nel caso di Rosalind Franklin.

I risultati per quanto riguarda l'integrazione delle funzionalità sono stati soddisfacenti. L'applicazione è in grado gestire la simulazione di oggetti di vario tipo ed inoltre questo avviene senza incrementare in maniera eccessiva l'utilizzo che questa fa delle risorse di sistema. Per quanto riguarda la fisica invece, i risultati sono stati poco soddisfacenti. Il motore fisico si è rivelato poco adatto alla gestione di elementi complessi. Non è possibile effettuare una simulazione vera e propria tramite un software di questo tipo, in quanto è evidente come questo sia nato per soddisfare altri bisogni.

La tesi è composta da 7 capitoli e di seguito ne viene fatta una breve descrizione.

Nel primo capitolo si trova un'introduzione al contesto in cui è collocato il lavoro.

Il secondo capitolo è invece relativo alle tecnologie. Vengono introdotti i concetti di VR e di game engine. Per quanto riguarda la VR si approfondiscono le architetture, i dispositivi comunemente utilizzati e vengono citate alcune applicazioni della stessa.

Il terzo capitolo tratta in particolare delle applicazioni della VR nell'ambito aerospaziale. Vengono poi citati alcuni lavori di ricerca che fanno uso dei game engine per l'implementazione di simulazioni computerizzate.

Nel quarto capitolo si parla invece della VR presso ALTEC, andando nel dettaglio di quelle che sono le funzionalità già presenti in ADN. Vengono inoltre brevemente descritti i dispositivi hardware utilizzati.

Nel quinto capito viene descritto il lavoro svolto per estendere le funzionalità di ADN, assieme alla modellazione delle caratteristiche fisiche di uno dei due oggetti della simulazione forniti come esempio.

Nel sesto capitolo vengono presentati i problemi riscontrati durante l'implementazione, con particolare enfasi sulla fisica, e vengono poi analizzate le prestazioni dell'applicazione.

Nel settimo ed ultimo capitolo vengono tratte le conclusioni sul lavoro e sulla qualità della simulazione fisica offerta dal game engine Unity.

Indice

Capitolo 1

Introduzione

La Realtà Virtuale, o Virtual Reality (VR), e le simulazioni sono tecnologie che suscitano grande interesse e la loro combinazione può portare a dei risultati interessanti. Le simulazioni infatti beneficiano della visualizzazione realistica che può fornire la VR, permettendo a chi studia i dati e la simulazione stessa di comprenderne più facilmente i risultati. Questo in particolar modo nell'ultimo decennio, dove la mole di dati che viene generata ed utilizzata è considerevole.

Astra Data Navigator (ADN), software realizzato da Aerospace Logistics Technology Engineering Company (ALTEC), nasce all'interno del contesto del progetto europeo *NEANIAS* [1], con lo scopo di fornire uno strumento duttile da mettere a disposizione della comunità scientifica. Questo accosta i concetti di VR e simulazione al fine di visualizzare dati stellari. Le principali funzioni dell'applicazione sono:

- la visualizzazione della maggior parte dei corpi celesti del Sistema solare, compresi anche i satelliti artificiali;
- la gestione ed il caricamento dinamico di diversi cataloghi stellari, che possono arrivare a contenere informazioni su miliardi di stelle;
- la visualizzazione delle orbite dei pianeti del Sistema solare e dei satelliti artificiali;
- la gestione di Digital Elevation Model (DEM) ove disponibili. Un DEM è una rappresentazione digitale dell'altezza del territorio. Questi sono tipicamente generati tramite tecniche di telerilevamento, tramite sensori posizionati su satelliti orbitanti attorno al pianeta.

L'obiettivo del lavoro è quello di estendere le funzionalità già presenti nel software ADN, permettendo all'applicazione di emulare il comportamento di oggetti di vario tipo all'interno del mondo virtuale. Si vuole quindi realizzare un'infrastruttura che si prenda carico della gestione della simulazione. Inoltre, parallelamente

all'estensione delle funzionalità, si vuole osservare la qualità della simulazione fisica che Unity, il game engine utilizzato per la realizzazione dell'applicazione, è in grado di fornire. Di seguito vengono esposti i principali requisiti che hanno influenzato le scelte di progetto:

- **simulazione realistica**: ciò che avviene nel mondo virtuale non deve essere semplicemente verosimile, deve essere aderente alla realtà. Tutta l'applicazione è stata costruita attorno a questo requisito e quindi anche le nuove funzionalità, incluse quelle che coinvolgono il physics engine per il calcolo delle interazioni fisiche.

- **Struttura indipendente dalla realizzazione dei singoli elementi che controlla**: la struttura per la gestione delle simulazioni deve essere quanto più generica possibile al fine di essere compatibile con entità di vario tipo.

La volontà di tenere separata l'implementazione della struttura dal modo in cui vengono realizzati i singoli elementi si scontra con la necessità di tenere in considerazione alcune delle caratteristiche che li contraddistinguono. Sono stati identificati due tipi di oggetti:

- oggetti già presenti nella scena, come satelliti, il cui comportamento può essere replicato nella scena principale;

- oggetti non presenti nella scena, come rover, per i quali è stata messa in discussione la fattibilità di avviarne la simulazione nella scena principale. Per questi si presenta il problema di dove e quando istanziarli all'interno del mondo virtuale.

Infine, viene introdotta una distinzione tra le funzioni scritte per l'implementazione dei comandi relativi ai singoli oggetti. Queste possono essere di due tipi:

- funzioni che utilizzano le API fisiche;

- funzioni che non coinvolgono calcoli fisici.

Questa distinzione è necessaria in quanto Unity gestisce la chiamata a questo tipo di funzioni in due momenti differenti.

Capitolo 2

Tecnologie

2.1 La Realtà Virtuale

Virtual Reality (VR) è un termine che viene spesso utilizzato, presente ormai nell'immaginario collettivo, eppure dare una definizione tecnica di cosa sia la VR non è affatto banale. In prima istanza si potrebbe affermare che la VR sia un'interfaccia verso un mondo verosimile, nel quale elementi reali si mescolano ad elementi virtuali più o meno credibili. Secondo Gregor Burdea, il cui contribuito nel campo è stato significativo, questa sarebbe un'interfaccia uomo-macchina di alto livello in cui la grafica computerizzata viene utilizzata per la simulazione in tempo reale di un mondo realistico. Inoltre, ad essere simulata è anche l'interazione con gli elementi che fanno parte di questo mondo sintetico, attraverso canali sensoriali multipli [2]. Analizzare la precedente definizione permette di individuare le caratteristiche fondamentali proprie di questa tecnologia:

- **interattività**: il mondo sintetico non è statico, bensì interattivo. L'interazione inoltre è istantanea, infatti il mondo reagisce immediatamente allo stimolo dell'utente. Questo perchè l'utente percepisce come reale ciò su cui può agire ed apportare modifiche, tanto meglio quanto più le metafore di interazione sono vicine alla realtà e comprensibili dall'utente.

- **Immersione**: è la misura della sensazione provata dall'utente di essere fisicamente presente nel mondo virtuale. Al fine di aumentare il senso di immersione è di primaria importanza sfruttare la grafica computerizzata al meglio, per proporre un ambiente quanto più fotorealistico possibile. Oltre a questo però, è bene coinvolgere anche gli altri sensi, come udito, tatto ed il sistema vestibolare; più sono i sensi coinvolti e maggiore sarà il senso di immersione dell'utente. Ovviamente questo concetto è strettamente legato ai dispositivi che l'applicazione in VR utilizza per interfacciarsi con l'utente.

- **Presenza**: concetto correlato a quello di immersione, ma distinto. Questo indica la sensazione dell'utente di appartenere al mondo virtuale. Maggiore è il senso di immersione e maggiore sarà anche il senso di presenza. Ciò nonostante, l'immersione non è sufficiente da sola a far credere all'utente che il mondo sintetico sia reale. Fondamentale è per questo che il mondo sintetico reagisca in modo concreto e coerente agli stimoli dell'utente, ma anche che l'utente sospenda volontariamente le sue facoltà critiche. Questo concetto è noto come *sospensione dell'incredulità* e vi gioca un fattore fondamentale l'immaginazione, senza la quale la simulazione in VR perderebbe tutto il suo significato.

2.1.1 Sistema visivo umano

Prima di alcuni accenni storici e di descrivere i tipi di realtà virtuale ed i dispositivi che le caratterizzano è bene comprendere il funzionamento del sistema visivo umano.

Gli occhi hanno a disposizione due tipi di fotorecettori, chiamati coni e bastoncelli. I primi sono localizzati in corrispondenza della fovea, ossia la regione centrale della retina in cui l'acutezza visiva è massima, e sono responsabili della visione diurna. I secondi invece si trovano principalmente nell'area periferica della retina e sono sensibili quando il livello di illuminazione è più basso. La distribuzione di questo tipo di fotorecettori e le caratteristiche fisiche dell'occhio determinano cosa del mondo intorno a noi riusciamo a vedere. Un parametro di particolare interesse quando si parla di VR è il *campo visivo*, anche noto come *Field of View (FoV)*; cono che delimita quella porzione di spazio dinnanzi all'osservatore che questo è in grado di osservare. Il campo visivo è ampio circa 170° in orizzontale e 120° in verticale per il singolo occhio. Quando consideriamo entrambi gli occhi questo può anche arrivare ai 220° orizzontali. I singoli occhi catturano immagini distinte, però una

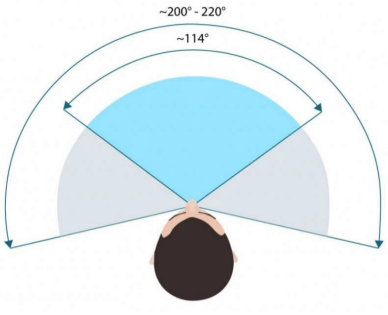

Figura 2.1. Campo visivo di un essere umano. In azzurro l'area di sovrapposizione nella quale è possibile avere percezione della profondità.

parte consistente di queste riprende la stessa porzione di spazio, quindi si sovrappongono, come è possibile notare nella Figura 2.1. È nell'area di sovrapposizione

che avviene il processo di ricostruzione tridimensionale, detto *stereopsi*, ad opera del cervello.

Visione tridimensionale

La percezione della profondità e quindi la visione 3D è possibile per mezzo della visione attraverso entrambi gli occhi. In particolare intervengono i processi di:

- **parallasse**: gli occhi osservano la medesima porzione di mondo circostante da punti nello spazio differenti;
- **stereopsi**: il cervello elabora e fonde le immagini catturate.

Il concetto di parallasse è fondamentale, in quanto da questo dipende la corretta sovrapposizione delle immagini che arrivano agli occhi dell'osservatore. La parallasse dipende dall'angolo di convergenza, ossia l'angolo che si forma tra l'asse della visuale ed il punto fissato, il quale dipende a sua volta dalla Interpupillary Distance (IPD), ossia la distanza tra le due pupille dell'osservatore. La distanza di un oggetto viene infatti valutata dal cervello a partire dall'angolo di convergenza e quindi dal valore della IPD. È possibile sfruttare queste conoscenze per far percepire all'osservatore un punto ad una distanza diversa da quella a cui si trova lo schermo che sta osservando: punti che si trovano nella stessa posizione nelle due immagini che arrivano agli occhi sono detti punti a disparità nulla e sono percepiti sul piano dello schermo. Se il punto osservato dall'occhio destro è più a sinistra del punto osservato dall'occhio sinistro si parla di punti a disparità negativa. Questi vengono percepiti come più vicini rispetto al piano dello schermo. Infine, nel caso contrario si parla di disparità positiva ed il punto è percepito come più lontano, quasi stesse entrando nello schermo. Tutto ciò funziona solo entro una decina di metri di distanza, quando la parallasse è ancora non trascurabile. Oltre tale distanza l'angolo di convergenza è davvero piccolo e cambia in maniera minima al variare del punto osservato, per cui il cervello utilizza altre informazioni per stimare la distanza di un oggetto. A grandi distanze il cervello sfrutta quelli che vengono definiti come *indizi di profondità*, i quali sono scollegati dal concetto di stereoscopia in quanto possono essere estrapolati da una singola immagine. Alcuni esempi di indizi di profondità sono le occlusioni, le ombre ed ombreggiature e la parallasse di moto.

2.1.2 Storia della Realtà Virtuale

Il concetto di VR non è così nuovo come può sembrare, infatti i primi dispositivi nacquero già negli anni '60 del secolo scorso. Quello che viene ritenuto come il primo sistema completo di realtà virtuale immersiva è *Sensorama*, brevettato nel 1962. Questo era un simulatore *passivo* di motociclo. Passivo perchè le immagini non erano generate da grafica di sintesi, ma era possibile visualizzare solo immagini precedentemente catturate con delle telecamere. Su questo dispositivo erano

state implementate diverse funzioni, quali ad esempio la visione stereoscopica, la vibrazione del sedile ed il manubrio, la simulazione del vento che arriva all'utente tramite l'attivazione di un asciugacapelli, la riproduzione degli odori ed altro ancora. La sua realizzazione racchiudeva al proprio interno molti dei concetti cui fanno riferimento le moderne applicazioni in VR. Nel 1968 venne poi realizzato il primo

Figura 2.2. Sensorama: da molti considerata la prima macchina per la realtà virtuale della storia.

Head Mounted Display (HMD), ossia il primo caschetto per la visione stereoscopica. Il dispositivo era costituito da due schermi CRT ancorati al soffitto. L'utente non poteva muoversi con indosso il caschetto, ma poteva farlo ruotare. Tale rotazione veniva utilizzata per capire quale porzione del mondo sintetico si stesse osservando. Negli anni seguenti vennero sviluppati e realizzati altri dispositivi, come ad esempio *Grope*, un braccio meccanico creato per essere un'interfaccia di tipo tattile appartenente alla categoria *force feedback*, il cui scopo era quello di impedire il movimento dell'utente in corrispondenza di ostacoli nel mondo virtuale. Fu nel 1982, nell'opera

Figura 2.3. Primo HMD della storia, realizzato da Ivan Sutherland.

dello scrittore Damien Brotherick, che venne utilizzato per la prima volta il termine

Virtual Reality [3]. I primi dispositivi commerciali per la realtà virtuale immersiva arrivarono poi nel 1987 da parte dell'azienda VPL, la quale mise sul mercato il primo sistema integrato, composto da visore, guanti, sistema di tracciamento per i movimenti dell'utente e calcolatore necessario alla generazione delle immagini di sintesi.

Le aspettative verso questa tecnologia raggiunsero il loro picco massimo tra gli anni '80 e '90, momento in cui iniziò ad essere evidente che non era ancora matura. Una serie di circostanze la rendevano infatti poco attraente:

- la simulazione in tempo reale di mondi sintetici richiede elevate capacità computazionali, che all'epoca solo calcolatori di fascia professionale, quindi molto costosi, potevano gestire.

- Le interfacce, soprattutto quelle video, erano di scarsissima qualità. Gli schermi a bassa risoluzione, dovendo essere piazzati a poca distanza dagli occhi dell'utente, non garantivano un'esperienza soddisfacente. I dispositivi di tracciamento ed altre interfacce non avevano una grande precisione, rendendo l'interazione col sistema difficoltosa.

- Era evidente la mancanza di software per la progettazione, lo sviluppo e la gestione dell'ambiente virtuale. Non esistevano infatti strumenti che permettessero l'implementazione di funzionalità complesse in tempi ragionevoli o di strumenti e soluzioni adattabili e riutilizzabili su larga scala.

Negli anni successivi invece, una serie di eventi fu responsabile della rinascita e riscoperta di tale tecnologia. A rendere possibile gli sviluppi fatti nel settore è stato senza dubbio lo sviluppo ed il miglioramento delle Graphics Processing Unit (GPU), le quali permisero di avere prestazioni decisamente maggiori e a costi più contenuti, tanto che l'esecuzione di programmi di grafica di sintesi divenne possibile anche su dispositivi di uso comune. Contribuì anche lo sviluppo dei dispositivi di interazione, i quali divennero sempre più precisi, con risoluzione sempre più elevata ed a prezzi sempre più accessibili. Per ultimo, ma non per importanza, ci fu lo sviluppo e distribuzione di strumenti software per la gestione di ambienti complessi, oggi noti come *game engine*. Questi sono costituiti da un software per la gestione del mondo tridimensionale, il *3D engine*, un software per la gestione dei calcoli fisici, il *physics engine* e molti altri strumenti che si occupano della gestione di aspetti di basso livello, di cui il programmatore non deve più farsi carico nel momento in cui va a sviluppare un'applicazione in VR.

2.1.3 Architetture e dispositivi per la Realtà Virtuale

È possibile vedere un'applicazione in VR come costituita da un insieme di blocchi, ognuno dei quali ha una funzione definita. Se ne possono distinguere 3 principali:

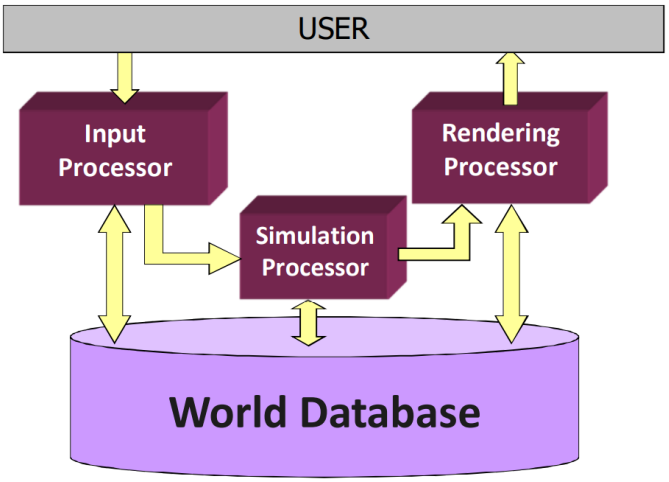

Figura 2.4. Architettura di un'applicazione in VR.

- **world database**: qui sono contenuti tutti i dati del mondo sintetico, dalle informazioni geometriche dei modelli 3D ai comportamenti predefiniti di agenti intelligenti.

- **Input processor**: si occupa di campionare i comandi immessi dall'utente. Questi possono essere di vario tipo, campionati con frequenze e precisioni differenti ed ognuno di questi è utile alla corretta interazione dell'utente con gli elementi del mondo virtuale.

- **Simulation processor**: si occupa di gestire la logica applicativa della simulazione. Elabora gli stimoli forniti dall'utente e calcola quale sarà l'impatto di questi sul mondo di gioco e sull'utente stesso.

- **Rendering processor**: si occupa di fornire i corretti stimoli sensoriali in risposta all'utente in base al risultato dell'elaborazione a carico del simulation processor. Qui non si fa riferimento solo alla componente visiva, ma a tutte le interfacce utilizzate dall'applicazione, le quali andranno a stimolare i diversi sensi coinvolti.

È importante notare però che tale visione è concettuale ed è quindi indipendente da quelli che sono i dispositivi effettivamente utilizzati da un sistema per la sua realizzazione.

Tipi di Realtà Virtuale

Sebbene nel linguaggio comune il termine VR sia associato a dispositivi come i già citati HMD, in ambito tecnico lo si utilizza per fare riferimento ad un concetto più ampio. Con questo termine si fa riferimento infatti ad una simulazione interattiva che sfrutta esclusivamente la grafica di sintesi, ma i dispositivi utilizzati per l'implementazione possono differire. È possibile distingue, a seconda dei dispositivi utilizzati, due tipi di VR:

- **realtà virtuale di tipo desktop**: questa, come suggerisce il nome, utilizza interfacce comuni quali monitor, mouse, tastiere e joystick. I dispositivi in questione permettono una complessità dell'interazione limitata, spesso artificiosa, andando ad impattare sull'esperienza dell'utente in termini di senso di immersione e presenza.

- **Realtà virtuale immersiva**: è la VR cui ci si riferisce nel linguaggio comune. In linea generale, quello che si cerca di fare è aumentare il senso di immersione e presenza dell'utente sfruttando interfacce più *immersive*. Lo sforzo maggiore in tal senso viene indirizzato sempre verso la vista. La notevole differenza rispetto ai sistemi desktop è l'utilizzo di dispositivi per la visione in grado di:

  - far percepire all'utente la tridimensionalità del mondo sintetico;
  - coprire un FoV maggiore.

  Oltre alla vista, si cerca di stimolare più efficacemente anche gli altri sensi o si coinvolgono sensi che in ambienti desktop non sono stimolati affatto. Un esempio sono le interfacce vestibolari, che permettono all'utente di fare esperienza delle forze di inerzia associate ai movimenti che si compiono nel mondo sintetico.

Dispositivi di visualizzazione immersivi

I dispositivi di visualizzazione immersivi sono quei dispositivi che permettono di ingannare l'utente dandogli la sensazione che il mondo virtuale sia tridimensionale. Ne esistono di diversi tipi:

- **sistemi basati su occhialini**: questi dispositivi sono anche detti intrusivi, in quanto prevedono l'utilizzo di un dispositivo più o meno complesso che l'utente deve indossare. Si dividono ulteriormente in:
  - head mounted;
  - multiplexed.
- **Sistemi autostereoscopici**: anche detti sistemi non intrusivi, che quindi non prevedono che l'utente indossi dispositivi aggiuntivi. Questi si dividono ulteriormente in:
  - sistemi basati sulla parallasse;
  - display volumetrici.

È necessario premettere che con il termine multiplexing in questo contesto ci si riferisce al processo di divisione dell'informazione visiva in più unità o *flussi* per essere trasferita separatamente e poi essere riaggregata al termine, ossia dopo che questi giungono agli occhi dell'osservatore, quando vengono elaborati dal cervello. Le principali strategie che verranno menzionate di seguito sono il *multiplexing spaziale* ed il *multiplexing temporale*. Con multiplexing spaziale si fa riferimento a quelle tecniche che permettono di separare le informazioni visive nello spazio, in modo che ogni occhio ne percepisca una parte, ma non la parte percepita dall'altro occhio. Con multiplexing temporale invece ci si riferisce a quelle tecniche che separano i flussi video nel tempo, alternando un fotogramma di un flusso con quello dell'altro. I due flussi anche qui sono diretti separatamente verso i due occhi ed è importante che il flusso percepito da un occhio non sia percepito dall'altro. In entrambi i casi la ricostruzione dell'immagine avviene poi per accumulazione, grazie al fenomeno di persistenza della visione, "fenomeno per il quale l'immagine di un

oggetto sulla retina è percepita più a lungo (circa un decimo di secondo) di quanto duri l'esposizione della retina stessa al flusso luminoso proveniente dall'oggetto" [4].

Sistemi multiplexed con occhialini Questo tipo di sistemi si divide ulteriormente in sistemi a stereoscopia attiva e passiva. La stereoscopia attiva prevede l'utilizzo di proiettori ad alto tasso di aggiornamento, in inglese refresh rate, pari ad almeno $120Hz$, che proiettano alternativamente immagini per l'occhio destro e l'occhio sinistro, sfruttando quindi il mutliplexing temporale. Per separare i due flussi di immagini il sistema di proiezione è sincronizzato con degli occhialini indossati dall'utente, le cui lenti sono dei pannelli LCD, le quali vengono oscurate in maniera alternata. Ogni occhio percepirà quindi un flusso video con un refresh rate di $60Hz$, ottimale per la visualizzazione. Per quanto riguarda invece i dispositivi a stereoscopia passiva, questi possono sfruttare la polarizzazione della luce o in alternativa il multiplexing di colore, entrambe tecniche di multiplexing spaziale. Per quanto riguarda la polarizzazione della luce, la tecnologia utilizzata oggi è la polarizzazione circolare. La luce può essere vista come un'onda che si propaga nello spazio ed oscilla. Tipicamente la luce naturale non è polarizzata, ciò significa che non è possibile identificare un piano su cui l'oscillazione giace; ma nel momento in cui viene polarizzata viene definito anche un piano di oscillazione. Per polarizzazione circolare si intende che il piano su cui oscilla l'onda ruota nel tempo. Per disaccoppiare i due canali visivi una lente verrà polarizzata in senso orario, l'altra in senso antiorario. Per quanto riguarda invece il multipexing del colore, si effettua un filtraggio nello spazio dei colori in modo che il flusso diretto ad un occhio abbia informazioni di colore differenti e non presenti nel flusso diretto all'altro occhio. La prima tecnologia a sfruttare questo concetto è quella degli *anaglifi*. In pratica si utilizzano degli occhiali con dei filtri applicati sulle lenti; nel caso degli anaglifi si utilizza i rosso per la lente sinistra ed il ciano per la lente destra. In questo modo i filtri fanno passare solo la corrispondente porzione di luce. L'immagine percepita per mezzo di questa tecnologia è di qualità molto bassa ed inoltre la riproduzione dei colori non è del tutto esatta. Tecnologie moderne di questo tipo utilizzano due basi differenti dello spazio dei colori come filtri, non più rosso e ciano, realizzati tramite lenti speciali. La distorsione del colore percepito nell'immagine finale rimane comunque un problema.

Sistemi autostereoscopici Questo tipo di dispositivi sfrutta il multiplexing spaziale per la separazione delle immagini. Vengono utilizzate *barriere di parallasse* o *lenti lenticolari*. Le prime sono delle barriere poste tra lo schermo e l'osservatore, le quali sono tagliate in modo apposito da ostruire la visione di un pixel ad un occhio piuttosto che all'altro. Ciò nonostante, queste barriere vengono realizzate per una serie prefissata di punti in cui l'osservatore può trovarsi, per cui quando questo non si trova in corrispondenza di uno dei punti previsti non sarà in grado di visualizzare correttamente l'immagine. In questo modo l'osservatore perde l'illusione del 3D o,

ancora peggio, si verifica una inversione dei canali, ossia l'immagine che deve essere visualizzata da un occhio viene in realtà visualizzata dall'altro. Le lenti lenticolari invece sono delle lenti di ingrandimento, poste tra lo schermo e l'osservatore, progettate in modo tale da deviare la luce. Anche queste quindi, come le barriere di parallasse, indirizzano la luce in modo tale che un determinato pixel sia visualizzato da un occhio piuttosto che dall'altro. Anche per queste la limitazione è data dal fatto che la corretta visualizzazione 3D è garantita solo in alcuni punti prestabiliti in fase di realizzazione.

Display volumetrici Questi dispositivi non sfruttano la stereoscopia, ma creano una vera e propria rappresentazione tridimensionale dell'oggetto. Questa tecnologia è ancora in fase di sviluppo ed è poco conosciuta, ma ci sono un paio di esempi molto interessanti. Uno è il *depth cube*, dispositivo che ha una serie di pannelli LCD paralleli allo schermo, che sono resi trasparenti oppure opachi in successione. Ogni pannello rappresenta una fetta 2D dell'oggetto tridimensionale che si vuole rappresentare ed infine è possibile percepire la profondità per mezzo del fenomeno di accumulazione. Un altro esempio è lo *swept volume display*, il quale crea un immagine tridimensionale sempre per accumulazione ma, a differenza del depth cube, utilizza una superficie circolare che ruota attorno al proprio asse centrale. Ad ogni passo viene proiettata una fetta del volume dell'oggetto. La velocità di rotazione è tipicamente di 30 semi-giri al secondo e ad ogni semi-giro è associata un'immagine differente, che equivale quindi ad avere 30 fotogrammi al secondo.

Head Mounted Display Gli HMD, come accennato, sono dei caschetti indossabili, provvisti di due piccoli schermi posizionati a poca distanza dagli occhi dell'utente. Questo sistema offre, per sua natura, la miglior separazione possibile delle immagini. Questi hanno inoltre una serie di strumenti di contorno che sono necessari a garantire un'esperienza immersiva, quali ad esempio i sistemi di tracciamento, essenziali per mappare movimenti e gesti fatti dall'utente in comandi per il mondo sintetico. Le ottiche di cui sono dotati hanno una particolarità: nonostante si trovino a pochi centimetri dagli occhi, proiettano l'immagine dando l'impressione che lo schermo sia situato a 1-5m dinnanzi all'utente. Questo viene fatto per ridurre la fatica degli occhi ed aumentare anche il FoV coperto. Lo svantaggio però è che diminuisce il numero di pixel che è possibile racchiudere in un grado, andando in sostanza a ridurre la risoluzione. Se la risoluzione in pixel non è abbastanza elevata è possibile percepire un effetto di quadrettatura sullo schermo, dovuto all'eccessiva vicinanza di questo all'occhio. Altro parametro molto importante di cui si è parlato e che bisogna tenere in considerazione è la IPD. La lente mette a fuoco nel punto centrale e questo deve essere allineato all'occhio per avere un esperienza ottimale. Se la distanza tra le lenti non corrisponde a quella tra le pupille le immagini sono sfocate, con conseguente senso di affaticamento della vista e senso di vertigini. Gli HMD possono essere di tipo tetehered, ossia collegati ad un calcolatore, il quale

si occupa del calcolo di grafica e logica applicativa, lasciando al dispositivo il solo compito della visualizzazione e di eventuali attività di tracciamento, o untetehered, che non richiedono collegamenti con elementi esterni. Infine, è bene distinguere HMD a 3 o 6 gradi di libertà, in inglese Degrees of Freedom (DOF). Dispositivi a 3DOF sono in grado di tracciare le rotazioni del dispositivo, mentre dispositivi a 6DOF sono in grado di tracciare, oltre alle rotazioni, anche le traslazioni, grazie a dei sistemi incorporati nel caschetto o separati.

2.1.4 Oltre la Realtà Virtuale

La VR è in realtà solo una piccola parte di un processo di integrazione della grafica computerizzata all'interno del mondo reale. Come è possibile notare nella Figura 2.5, agli estremi ci sono il mondo reale ed il mondo virtuale. Nel mezzo però

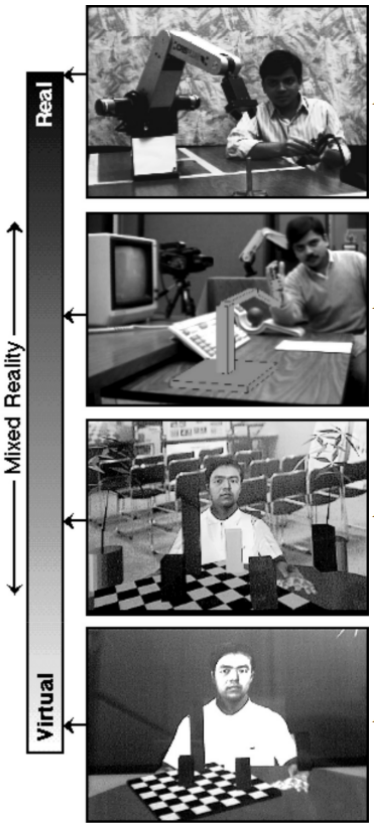

Figura 2.5. VR continuum definito da Paul Milgram. (Fonte: Milgram et al [5])

troviamo una serie di immagini in cui la componente reale progressivamente scompare in favore di quella virtuale. Milgram introduceva il concetto di *Mixed Reality (MR)*, ossia realtà mista [5]. Come suggerisce il nome, la MR mescola elementi

reali con elementi virtuali. A seconda delle quantità relative coinvolte di mondo virtuale e mondo reale parliamo di *Augmented Virtuality (AV)*, quando a prevalere è la componente virtuale, o di *Augmented Reality (AR)*, quando a prevalere è la parte reale, che viene arricchita con elementi del mondo virtuale. Applicazioni in VR ed in MR hanno solitamente dei requisiti tecnici diversi e spesso sono costrette ad adottare dispositivi di interazione differenti, primi tra tutti i dispositivi di visualizzazione. Per la VR è preferibile avere dispositivi immersivi, che coprano al meglio il campo visivo dell'utente per estraniarlo il più possibile dal mondo circostante. Applicazioni in MR invece non hanno questa necessità, sebbene coprire un campo visivo più ampio sia apprezzabile. Generalmente le applicazioni in VR richiedono il massimo realismo, mentre per applicazioni in MR un realismo minimo è più che sufficiente. Per queste ultime è infatti spesso necessario proporre una grafica stilizzata per distinguere in modo chiaro gli elementi del mondo sintetico da quelli del mondo reale. Infine, per quanto riguarda l'accuratezza dei sistemi di tracciamento, le applicazioni in MR richiedono la massima accuratezza possibile, in quanto gli elementi del mondo virtuale devono essere posizionati in modo coerente con quelli del mondo reale, altrimenti è possibile che si verifichino fastidiosi fenomeni di compenetrazione e sovrapposizioni non corrette.

2.1.5 Applicazioni

Sono molteplici le applicazioni di queste tecnologie. Tra queste vi sono:

- videogiochi;
- addestramento;
- manutenzione;
- insegnamento;
- applicazioni mediche;
- architettura e design;
- shopping;
- visualizzazione di dati complessi.

2.2 Game Engine

Il Game Engine, o motore di gioco, è un software storicamente realizzato allo scopo di supportare i programmatori nello sviluppo di videogiochi. I primi videogiochi

venivano programmati dalle fondamenta ed ogni aspetto, dalla grafica ai personaggi non giocanti, doveva essere gestito ed ottimizzato a dovere dagli sviluppatori per cercare di sfruttare a pieno la potenza computazionale a disposizione. Questo approccio era però molto oneroso, infatti non era possibile riutilizzare quasi nulla e quindi la realizzazione del videogioco successivo richiedeva il reiterarsi di tutti i processi. Fu negli anni '90 che nacquero i primi motori di gioco di terze parti; un esempio tra tutti è *Unreal Engine*, nato nel 1998. I primi game engine proprietari nacquero invece già una decina di anni prima. Questi strumenti si dimostrarono essere molto utili agli sviluppatori: permettevano loro di concentrarsi sulla logica di gioco e lasciare da parte gli aspetti di basso livello, quali l'integrazione nell'applicazione di audio, grafica, gestione di agenti intelligenti ed altro ancora. Questi sistemi oggi implementano funzionalità non banali e risultano essere piuttosto complessi. Tra gli elementi costitutivi di questi programmi ce ne sono due di particolare interesse:

- **rendering Engine**: si occupa del processo di creazione dell'immagine da visualizzare a schermo a partire da scene bidimensionali o tridimensionali. Il rendering è un'operazione molto complessa e sebbene gli sviluppatori abbiano la possibilità di modificare il comportamento delle operazioni che vengono eseguite durante questo processo, ciò accade di rado.

- **Physics Engine**: si occupa dei calcoli relativi alla fisica, i quali sono molto onerosi. Tipicamente questi sono ottimizzati per poter funzionare in tempo reale, a discapito dell'accuratezza. Ciò nonostante, il risultato finale è verosimile. Un physics engine si occupa tipicamente di rilevare le collisioni e poi calcolare le forze in gioco nelle interazioni tra corpi ed in generale nelle interazioni fisiche degli oggetti della scena.

Al giorno d'oggi nel mercato esistono svariati game engine, alcuni di terze parti, altri open source ed altri proprietari. Alcuni degni di nota sono:

- Unreal Engine;
- Unity;
- REDengine;
- Godot;
- Rockstar Advanced Game Engine (RAGE);
- CryEngine;
- Northlight.

I game engine negli ultimi anni hanno trovato applicazione anche in ambiti quali quello cinematografico. Un esempio molto recente si trova nella serie *The Mandalorian*, dove l'Unreal Engine 5 è stato utilizzato per creare dei background in tempo reale su delle pareti LED durante le riprese della serie [6]. L'interesse verso questi strumenti decisamente complessi e versatili cresce sempre di più. Negli ultimi anni si sta sperimentando il loro utilizzo per le simulazioni fisiche, grazie alla presenza del physics engine in grado di effettuare calcoli in tempo reale ed al realismo della rappresentazione grafica.

Capitolo 3

Stato dell'arte

3.1 Applicazioni di Realtà Virtuale in ambito aerospaziale

Enti come la National Aeronautics and Space Administration (NASA) possono essere considerati dei pionieri della VR. La NASA infatti ospitò il suo primo progetto in VR nel 1985 e lanciò nell'anno successivo il prototipo del suo Virtual Visual Environment Display (VIVED) al CES. Nell'immaginario dell'organizzazione questa tecnologia avrebbe potuto essere utilizzata per svariati compiti. Oltre a fornire da strumento supplementare per l'addestramento degli astronauti, lo spettro di possibili applicazioni comprendeva anche quello dell'educazione e dell'intrattenimento: le persone comuni avrebbero potuto osservare i pianeti e le loro orbite, oppure vedere attraverso le telecamere di un rover in movimento su un altro pianeta. Tutto comodamente dalla Terra, tramite un semplice apparecchio elettronico, senza dover per forza partire su un razzo ed andare nello spazio. Recentemente, oltre a NASA, anche la European Space Agency (ESA) ed altre agenzie spaziali hanno compreso le potenzialità di questi strumenti e li utilizzano attivamente nei loro processi.

3.1.1 Realtà Virtuale presso NASA

Storicamente presso NASA la VR è stata utilizzata per l'addestramento deli astronauti. Questi infatti compiono il loro lavoro in un ambiente ostico e per sua natura pieno di rischi potenzialmente fatali. Inoltre, i compiti da loro svolti sono molto complessi e spesso è necessario che siano eseguiti in maniera perfetta per evitare costosi danni alle apparecchiature o, peggio ancora, mettere a repentaglio la sicurezza dei colleghi.

Nel lavoro di ricerca svolto da Garcia et al [7] è possibile trovare un resoconto degli avvenimenti e delle tecnologie salienti riguardo l'utilizzo della VR preso NASA. Qui di seguito ne vengono illustrati alcuni di particolare importanza.

Il primo tentativo di utilizzare la VR nacque poco tempo dopo la messa in funzione del telescopio spaziale Hubble, quando venne rilevato un errore nello specchio primario. Però, dato che esistevano già strutture che fornivano gli strumenti necessari alla risoluzione del problema, nelle quali si trovavano mockup appositamente realizzati, il progetto venne considerato ridondante e non ebbe quindi vita lunga. Nonostante ciò, il gruppo formato in quell'occasione continuò a lavorare sul software grafico 3D, oggi noto come *Dynamic On-board Ubiquitous Graphics (DOUG)*. Tramite questo realizzarono un modello 3D del telescopio Hubble, del satellite orbitale e del braccio robotico dello shuttle. Nel 1992 il gruppo venne riconosciuto formalmente ed oggi è noto come *Virtual Reality Laboratory (VRLab)*. Lo strumento realizzato dal gruppo permetteva di visualizzare in maniera immersiva gli elementi precedentemente citati, cosa che non era possibile fare in nessun'altra struttura e con nessun altro strumento.

Mentre inizialmente il VRLab serviva come supporto per l'addestramento degli astronauti alle missioni di riparazione del telescopio Hubble, già dopo pochi anni in esso vennero integrate altre attività di formazione legate ad altri compiti e missioni. Le Extra Vehicular Activity (EVA) sono, come suggerisce il termine, delle attività che gli astronauti svolgono al di fuori del proprio veicolo spaziale, quale che esso sia. Attività di questo genere sono molto rischiose per vari motivi. Uno dei possibili rischi è il distaccamento dal veicolo. Quale che sia il motivo per il quale sia avvenuto, l'astronauta è per sicurezza dotato di un sistema di propulsione, chiamato *The Simplified Aid for EVA Rescue (SAFER)*, che gli permette di ritornare verso il veicolo e mettersi in sicurezza. Nonostante questa sia una eventualità remota, l'astronauta deve essere preparato ad affrontarla. Fu in quest'ottica che l'addestramento all'utilizzo di SAFER venne integrato nelle attività del VRLab. Sebbene l'introduzione all'utilizzo di SAFER facesse già parte dell'addestramento di un astronauta, l'interfaccia di visualizzazione precedentemente utilizzata consisteva in uno schermo a stereoscopia attiva. Dal momento in cui questo è stato integrato nel VRLab, è stato adottato un HMD come dispositivo di visualizzazione, fornendo un'esperienza più immersiva e realistica. Oggi il simulatore di SAFER viene utilizzato per due scopi:

- lezioni introduttive a SAFER: in questa parte gli astronauti sono tenuti a familiarizzare con lo strumento ed inoltre viene simulato un controllo diagnostico delle sue funzionalità. Il controllo diagnostico verrà poi ripetuto sul vero e proprio SAFER all'interno della International Space Station (ISS), prima di ogni EVA. Nella simulazione l'astronauta riceve dei messaggi a schermo che gli indicano la presenza di eventuali errori. L'astronauta deve sapere come reagire ad ognuno di questi per la sua sicurezza.

- Simulare il volo con SAFER in VR: qui gli astronauti vengono addestrati concretamente all'utilizzo del sistema. Per fare questo si replica una situazione di distacco dalla ISS. Dopo un periodo di trenta secondi, tempo tecnico stimato

per prepararsi all'utilizzo del sistema, l'astronauta può iniziare la prova, che consiste appunto nel ritornare verso la stazione. Tipicamente bisogna passare un test all'utilizzo di SAFER. Per prepararsi gli astronauti ripetono svariate volte la simulazione in condizioni di velocità di distacco e velocità di rotazione ogni volta differenti.

Figura 3.1. L'astronauta Andreas Mogensen mentre si addestra al Virtual Reality Laboratory del NASA Johnson Space Center in Texas. (Fonte: ESA [8])

Tra i vari compiti che si sono aggiunti alla gestione del VRLab si annovera anche lo *Charlotte Mass Handling Robot*. Questo sistema permetteva agli astronauti di provare e sperimentare la manipolazione di oggetti in condizioni di microgravità. La simulazione è molto accurata e fornisce un'esperienza molto simile a quella reale. Il sistema era decisamente meno costoso e più facilmente riconfigurabile della soluzione adottata precedentemente, per la quale era necessario emulare perfettamente le proprietà di massa degli oggetti da manipolare.

Questi strumenti vennero sviluppati ed adattati per l'addestramento a terra degli astronauti, nel VRLab, ma sino a quel momento nessuno di questi era stato portato a bordo di una stazione spaziale in orbita attorno al pianeta. È stato nel 2001 che DOUG venne portato sulla ISS. Negli anni successivi, come già accennato, strumenti per l'addestramento in VR per SAFER vennero portati a bordo della stazione, in modo da permettere agli astronauti di rinfrescare le loro abilità nell'utilizzo dello strumento. Mentre a fine degli anni '90 gli strumenti per la VR, in particolare per la visione ed il tracciamento dei gesti delle mani, erano molto acerbi, poco precisi e molto costosi, negli ultimi dieci anni la situazione è migliorata drasticamente. Al VRLab si è passati da strumenti realizzati appositamente al recente utilizzo di strumenti commerciali facilmente reperibili e di costo decisamente inferiore. Oggi infatti gli strumenti utilizzati al laboratorio sono il *HTC Vive Pro* con un sistema di tracciamento *Lighthouse Tracking*, sviluppato da *VALVE Corporation* per HTC Vive. Nonostante gli HTC Vive e SteamVR siano gli

strumenti adottati al momento nel VRLab, il software sviluppato rimane indipendente dall'hardware utilizzato e può essere configurato per funzionare su qualsiasi dispositivo commerciale disponibile in circolazione. Nel laboratorio sono installate due stazioni immersive, denominate EV1 (Extra Vehicular) ed EV2. Solitamente, le attività degli astronauti, come ad esempio le EVA, vengono effettuate in coppia; per questo le due stazioni sono situate una accanto all'altra e permettono a chi si addestra di immergersi nello stesso mondo virtuale.

Figura 3.2. Stazioni immersive EV1 ed EV2 utilizzate dagli astronauti Anne McClain a destra e Nick Hague a sinistra. (Fonte: Garcia et al [7])

3.1.2 Realtà Virtuale presso ESA

La NASA è stata la prima agenzia spaziale a comprendere le potenzialità che la VR, in particolare la VR immersiva, poteva offrire. In questa ottica, seppur con qualche anno di distanza, anche altre agenzie, come ESA, ne hanno compreso il valore. Nel 2015 infatti, presso l'European Astronaut Centre (EAC), è stato introdotto l'eXtended Reality Laboratory (XRLab), con l'obiettivo di adoperare tecnologie di VR, AR e MR per l'addestramento degli astronauti e non solo. L'agenzia, guardando al futuro, è convinta di poter sfruttare queste tecnologie per le future missioni quali il *Lunar Gateway*, a scopi di design della stazione lunare e altro ancora [9]. Un esempio attuale di come ESA utilizza oggi le tecnologie in XR è lo Joint Investigation into VR for Education (JIVE). Questo software viene utilizzato dagli astronauti di ESA per addestrarsi, tra le varie cose, al controllo remoto del braccio robotico Canadarm2. Canadarm2 è uno strumento essenziale a bordo della ISS. Le sue funzioni includono:

- trasporto di oggetti;
- spostamento di astronauti durante una EVA in punti difficili da raggiungere;
- tracciare ed agganciare veicoli da carico che non possono attraccare in autonomia.

Da qui si capisce che l'addestramento per l'utilizzo di questo braccio robotico è centrale ed eventuali errori potrebbero portare a conseguenze disastrose. JIVE non è nato dal nulla, ma come spesso accade è un'evoluzione di un processo che prima avveniva in un contesto non immersivo: JIVE è l'evoluzione di Dynamic Skills Trainer (DST). Questo emulava la postazione presente sulla ISS, fornendo all'utilizzatore un'interfaccia hardware molto simile a quella che avrebbe trovato a bordo. La visualizzazione però avveniva su un normale schermo desktop. Una miglioria apportata dal nuovo software è sicuramente stata l'utilizzo di un'interfaccia immersiva, ma questa non è l'unica. Il nuovo simulatore sfrutta infatti il *metodo dei Loci* [10], noto anche come metodo del *palazzo mentale*. Questo consiste nell'associare un concetto ad un luogo, per cui l'intera applicazione divide le sue esercitazioni in diversi ambienti e ad ogni ambiente è associato un argomento diverso. Gli sviluppatori di JIVE hanno lavorato a stretto contatto con i suoi

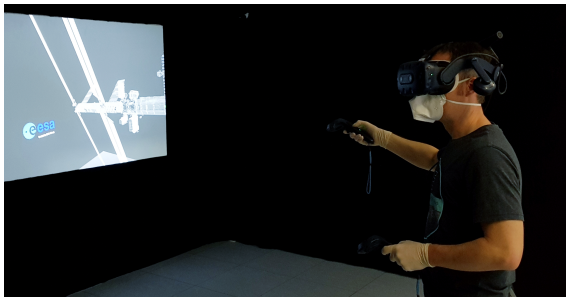

Figura 3.3. L'astronauta Matthias Maurer utilizza JIVE presso l'EAC di ESA in Germania. (Fonte: ESA [11])

utilizzatori per perfezionare al meglio il sistema. Le prime impressioni sono state molto positive ed un aspetto su cui si sono trovati d'accordo tutti coloro che hanno potuto provare il sistema riguarda l'immediatezza con la quale questo strumento immersivo permette di familiarizzare con la struttura della ISS.

Parafrasando il pensiero dell'astronauta Matthias Maurer, che è possibile osservare nella Figura 3.3 mentre utilizza JIVE, la VR permette di *vivere* la missione. Permette di esplorare moduli ed addestrarsi al loro interno prima ancora che un singolo componente di questi sia fisicamente realizzato. Questo sarà di grande utilità per le future missioni sulla Luna e su Marte [11].

3.1.3 Utilizzi di VR ed AR sulla ISS

Secondo quanto riportato sul sito internet della stessa NASA [12], oggi la VR e la AR sono utilizzate sulla ISS per diversi scopi:

- **assistenza**: un'applicazione in AR, funzionante su tablet o HMD, elabora le immagini catturate dalle telecamere e fornisce a schermo una serie di suggerimenti o elementi 3D utili all'astronauta per proseguire nel suo compito.

Questo permette di ridurre drasticamente i tempi di addestramento necessari per chi dovrà salire a bordo della ISS ed anche ridurre la necessità di contattare i tecnici del centro di controllo. L'utilizzo di un dispositivo per la AR come HoloLens permette al personale a bordo di eseguire ricambi ed aggiornamenti dei componenti a strumenti molto complessi, nonostante non abbiano una conoscenza specifica dell'apparecchiatura in questione. Senza un dispositivo come HoloLens bisogna utilizzare delle telecamere posizionate dietro l'astronauta o al di sopra di questo per permettere ai tecnici da terra di osservare l'operato. Con gli occhialini menzionati invece i tecnici possono seguire le azioni del personale a bordo come se stessero loro in prima persona operando sul dispositivo, favorendo l'immediatezza dell'interazione.

- **Controllo di robot**: uno studio condotto da ESA sta sfruttando la VR per controllare da remoto veicoli spaziali e bracci robotici.

- **Intrattenimento**: Delle telecamere a 360° sono state utilizzate per filmare alcune delle attività che gli astronauti a bordo della ISS compiono quotidianamente. Queste riprese sono state utilizzate per la creazione di una serie immersiva in VR chiamata *The ISS Experience*. Oltre al semplice intrattenimento però c'è confidenza nel fatto che chi guarderà questa serie potrà fornire degli spunti utili a migliorare la permanenza degli astronauti sulla stazione ed ispirare nuove ricerche in condizioni di microgravità.

- **Analisi delle variazioni delle facoltà psicofisiche**: la microgravità è una condizione estranea agli essere umani. L'assenza di gravità porta infatti una serie di disturbi e distorsioni nel modo in cui il soggetto percepisce ciò che lo circonda. A bordo della stazione, ed in realtà anche prima e dopo il periodo di permanenza su questa, vengono condotte varie analisi tramite delle applicazioni in VR. Queste si focalizzano ad esempio su come gli astronauti interagiscono con gli oggetti e lo spazio attorno a loro in condizioni di microgravità e, una volta sulla Terra, in condizioni normali. Altri studi monitorano la percezione che i membri a bordo della stazione hanno del tempo e della velocità degli oggetti, fattore che può influire pesantemente sulla riuscita dei compiti loro assegnati.

3.2 VR oltre l'addestramento

L'utilizzo di queste tecnologie funziona egregiamente nell'addestramento degli astronauti a compiti specifici ed anche molto complessi, quindi le varie agenzie ed anche le ricerche negli ultimi anni hanno iniziato a volgere lo sguardo verso altre possibili applicazioni.

Figura 3.4. L'astronauta Megan McArthur testa un'applicazione in AR con HoloLens. (Fonte: NASA [12])

Nell'articolo presente sul sito della NASA riguardo al VR team del Goddard Space Flight Center [13] si parla di come un'applicazione in VR possa essere utile anche per la visualizzazione di dati scientifici e perchè no, portare a qualche scoperta interessante. In particolare si fa riferimento al fatto che un gruppo di astronomi e ricercatori abbia utilizzato il software *PointCloudsVR* [14], sviluppato dal team appena menzionato, per visualizzare una base di dati contenente vari cataloghi stellari. Ci si è accorti di come questo strumento per la visualizzazione renda decisamente più immediato comprendere come le stelle si muovono e si raggruppano, cosa che potrebbe portare in futuro a ridefinire i raggruppamenti stellari dati per assodati fino ad ora. Citando le parole di Susan Higashio: "Invece di guardare prima una base di dati e poi l'altra, perchè non volarci in mezzo ed osservarle tutte insieme" [13].

Per quanto riguarda invece le simulazioni non realizzate a fini di addestramento un esempio è *Rover Analysis, Modeling and Simulation (ROAMS)*, risalente al 1999, utilizzato per replicare il comportamento di un rover. Oltre a servire da strumento per la verifica e la validazione del software utilizzato per il controllo dei rover stessi, compito solitamente svolto con l'ausilio di apposita strumentazione, questo sarebbe servito anche per supportare lo sviluppo di nuove tecnologie ed altro ancora. Nel successivo lavoro di ricerca ad opera di un gruppo del *Jet Propulsion Laboratory (JPL)* [15], pubblicato nel 2004, si cercava di migliorare le funzionalità di ROAMS. Già qui si prestò molta attenzione alla visualizzazione della replica virtuale del rover.

Nello stesso ambito, un più recente esempio di simulazione computerizzata è il lavoro di ricerca svolto presso il Politecnico di Milano, in collaborazione con l'Agenzia Spaziale Italiana (ASI) [16]. Il lavoro portato avanti da questi enti punta allo sviluppo di un sistema in grado di emulare il comportamento dei rover spaziali nella loro totalità, con l'obiettivo finale di utilizzarlo per analizzare lo stato di salute del veicolo e sviluppare algoritmi che possano adattarsi alle circostanze. Nel progetto citato, per modellare il rover ed i suoi componenti è stato utilizzato *MATLAB Simulink*. Il modello, a detta dei ricercatori, si è dimostrato in grado di emulare il sistema sia in condizioni di corretto funzionamento, che in condizioni in cui erano stati iniettati appositamente dei malfunzionamenti. Il modello digitale inoltre può essere molto utile in quanto fonte di una grande quantità di dati. I ricercatori li hanno salvati in una base di dati, la quale potrà essere sfruttata in futuro per la creazione di algoritmi di apprendimento automatico utili poi a riconoscere prontamente eventuali malfunzionamenti nel sistema reale.

3.3 Simulazioni computerizzate e Game Engine

Nei vari ambiti applicativi ed industriali, non solo in ambito aerospaziale, è importante condurre delle verifiche su ciò che si va a realizzare. I processi di verifica

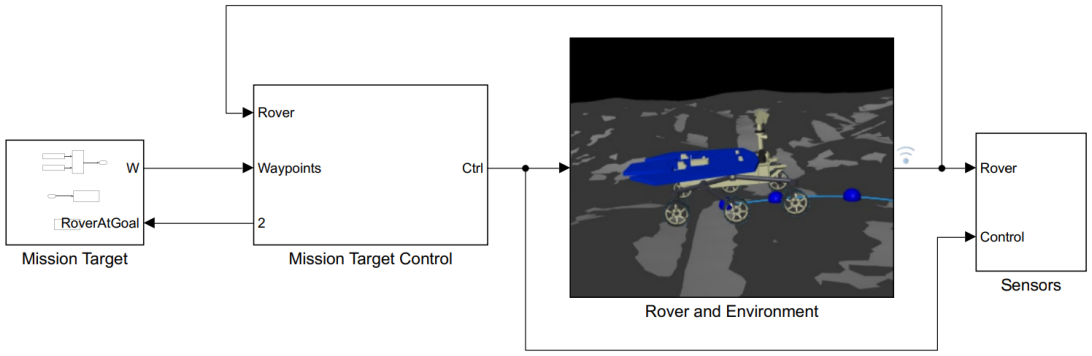

Figura 3.5. Schema concettuale del sistema realizzato dai ricercatori del Politecnico di Milano. (Fonte: Pinello et al [16])

però sono spesso macchinosi e molto costosi. L'approccio classico sarebbe quello di prendere il prodotto stesso o dei mockup e verificarne il comportamento. In entrambi i casi spesso è necessario ricreare le condizioni in cui questi oggetti devono agire, introducendo ulteriori spese e criticità. Oggi gli strumenti informatici e la potenza dei calcolatori hanno aperto una nuova frontiera in questo ambito. Sempre più spesso in letteratura si trovano esempi di come i calcolatori vengano utilizzati per la *simulazione* di sistemi ed apparecchiature. Esistono svariati software che permettono lo sviluppo di simulatori sofisticati e spesso per gestire tutti i vari aspetti di questi sistemi complessi bisogna utilizzarne diversi. Quando si sviluppa un sistema di questo tipo bisogna dedicare tempo alla gestione della simulazione, ma bisogna gestire anche la grafica e le caratteristiche tridimensionali, come anche le interazioni fisiche ed altri aspetti non di poco conto. È quindi evidente che spesso il programmatore si ritrova a dover integrare svariati software, in modo che comunichino adeguatamente tra di loro. Questo richiede molto tempo e quindi molti fondi, ma sopra ogni cosa grande abilità nella programmazione. Ultimamente l'attenzione della ricerca si sta spostando vero i game engine, per via, tra le varie cose, della rappresentazione tridimensionale realistica e per la presenza di un physics engine integrato. Di seguito verranno illustrati alcuni lavori di ricerca che sfruttano questi strumenti per la realizzazione di simulazioni in svariati ambiti applicativi.

3.3.1 Game Engine in ambito edilizio

Il lavoro di ricerca svolto da Kang et al [17] ha come scopo quello di analizzare i vantaggi e gli svantaggi di utilizzare un game engine come ambiente di sviluppo per realizzare una simulazione fisica in ambito edilizio. Nel lavoro viene ricreato

in 3D un carrello elevatore tramite il software *Blender* e ne viene replicato il comportamento tramite il *Blender Game Engine*. Per riprodurre la complessità del sistema reale è necessario che il modello tridimensionale, con le relative proprietà fisiche, sia quanto più accurato possibile. In questa ottica, nel modello realizzato dai ricercatori i vari elementi vengono collegati tramite i componenti *joint*, i quali possono modellare vari tipi di giunti, da giunti fissi a giunti completamente definiti dall'utente. Per quanto riguarda le collisioni, i componenti *collider*, responsabili di delimitare i confini entro i quali vengono calcolate le intersezioni tra i corpi, sono stati ottenuti a partire dalla geometria degli elementi del modello stesso. In conclusione, il lavoro evidenzia la presenza di alcune limitazioni nel motore fisico utilizzato. Alcuni peculiari problemi riscontrati dagli autori sono ad esempio il collasso dei giunti, i quali si separano in maniera innaturale, e la compenetrazione di alcuni elementi, come è possibile osservare in Figura 3.6. Viene osservato inoltre che i motori fisici dei game engine possono essere alle volte limitanti, in quanto difficilmente possono essere modificati i comportamenti che li regolano. Invece, la scrittura di un sistema apposito, se ben realizzato, può portare sicuramente a dei risultati migliori. Gli autori osservano inoltre che i motori di gioco offrono sicuramente un vantaggio per quanto riguarda la ricostruzione dei modelli tridimensionali. Gli ambienti creati sono infatti di alta qualità ed il tutto è ottenuto con poco sforzo. Inoltre forniscono un ambiente di sviluppo più facile ed intuitivo da gestire, grazie anche alle interfacce di cui sono dotati, che permettono di avere tempi di sviluppo più brevi. Secondo quanto riportato nell'articolo stesso, il tempo necessario per mettere in piedi il sistema è stato pari a 40 ore di lavoro.

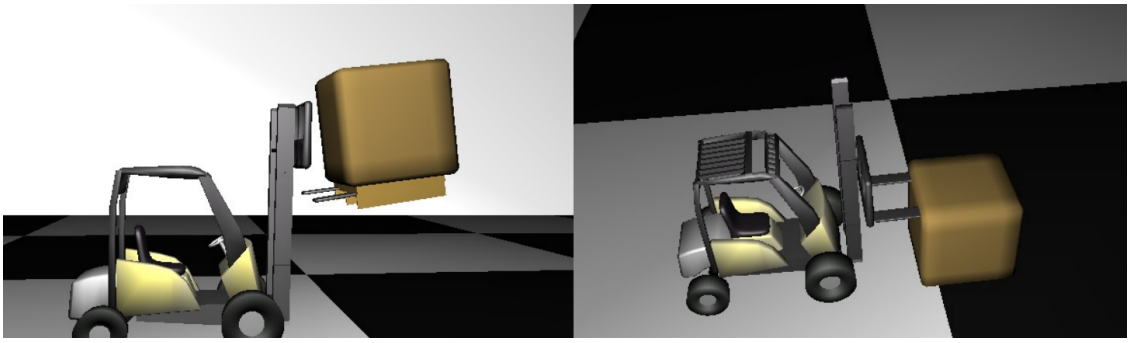

Figura 3.6. Immagini raffiguranti il carrello elevatore durante la simulazione. A sinistra la separazione innaturale dovuta all'elevato peso assegnato all'oggetto da sollevare. A destra la compenetrazione dovuta a dei problemi con la geometria dei componenti *collider*. (Fonte: Kang et al [17])

3.3.2 Game Engine per il settore dell'energia

Il lavoro di Sørensen et al [18] utilizza l'Unreal Engine 5 per esaminare vantaggi e svantaggi di utilizzare un game engine nello sviluppo di un simulatore per lo studio e l'utilizzo dell'energia eolica. Per condurre lo studio il gruppo ha realizzato una versione digitale dell'ambiente circostante e di due particolari modelli di pale eoliche. Per la realizzazione sono stati proficuamente sfruttati i vari strumenti messi a disposizione da Unreal Engine 5. Per quanto riguarda i parametri numerici della simulazione delle pale eoliche, questi sono stati emulati tramite il linguaggio di programmazione integrato in Unreal, C++. Il software è stato realizzato per permettere di stimare la quantità di energia prodotta da diversi dispositivi in diverse condizioni e metterle a confronto. I ricercatori affermano che Unreal Engine può essere utilizzato con ottimi risultati per via delle svariate funzionalità che offre e del livello di dettaglio che riesce a dare all'ambiente virtuale.

Figura 3.7. Immagine raffigurante il lavoro di Sørensen et al, realizzato in Unreal Engine 5. (Fonte: Sørensen et al [18])

3.3.3 Game Engine per l'industria automobilistica

Il lavoro di Wang et al [19] invece si concentra sullo sviluppo di un simulatore per l'analisi di automobili connesse ed a guida autonoma. Per questa ricerca è stato utilizzato il game engine Unity. Per emulare il software di controllo di questi sistemi ed eventuali strumenti esterni sono stati utilizzati il linguaggio di scripting C#, linguaggio supportato in Unity, *MATLAB*, script python ed altro ancora. La ricerca nasce dall'esigenza di condurre test su questi veicoli, test che su strade reali sono sempre sottoposti a regolamentazioni stringenti e costi elevati. Unity poi è uno strumento versatile e compatibile con gran parte delle interfacce immersive, quali ad esempio pedali e volanti usati nei videogiochi di guida, che qui rendono possibili

studi relativi alla presenza del fattore umano. In alternativa è sempre possibile utilizzare strumenti di interazione classici. Inoltre, avere un'elevata qualità del mondo virtuale permette di inscenare in maniera piuttosto accurata casi di guida in diverse condizioni meteorologiche.

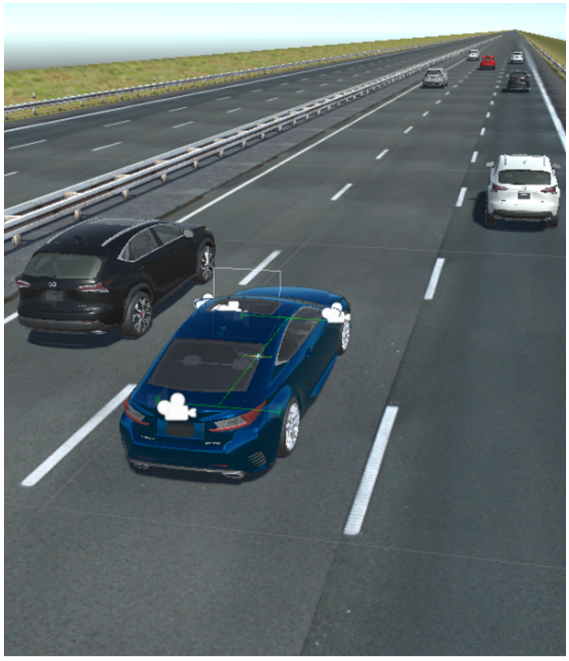

Figura 3.8. Immagine raffigurante il lavoro di Wang et al per Toyota, realizzato tramite Unity. (Fonte: Wang et al [19])

3.3.4 Game Engine in ambito aerospaziale

Nel lavoro di Xie et al [20] si cerca di utilizzare la VR per riprodurre gli stimoli sensoriali di cui si farebbe esperienza alla guida di un rover in movimento su di un terreno. Il sistema nasce con lo scopo di fornire all'equipaggio che controlla il veicolo da remoto più consapevolezza dell'ambiente circostante, cercando di portare il pilota ad una scelta più oculata del percorso da seguire. Per raggiungere questo scopo il gruppo ha realizzato una interfaccia uomo-macchina che coinvolge più canali: sono state utilizzate sia delle interfacce vestibolari che di tipo tattile. Il sistema vestibolare è stimolato da una piattaforma a 6DOF che simula lo stato di moto del rover, riproducendo vibrazioni, accelerazioni ed inclinazioni in base alla conformazione del terreno nel punto in cui si trova. L'interfaccia tattile utilizzata è di tipo *force feedback* e consiste in una leva in grado di opporre resistenza ai movimenti dell'utente in base alla conformazione del terreno in quel punto. L'ambiente della simulazione è stato generato per mezzo del software Unity. Il suolo lunare è

stato ricreato in maniera procedurale per simulare una distribuzione a piacere di crateri e rocce. Questi ostacoli e la loro generazione sono un punto cruciale per la simulazione, dato il fine che la ricerca si propone di raggiungere. Una volta raccolti i dati sul terreno circostante, questi vengono poi elaborati da un algoritmo che determina quali aree sono facilmente percorribili e quali non lo sono, ad esempio per la presenza di una pendenza troppo elevata. Partendo da queste analisi viene poi tracciato un percorso fino al punto che si vuole raggiungere, ritenuto ottimo. I ricercatori hanno effettuato degli esperimenti per validare il sistema. Agli utenti veniva prima chiesto di guidare il rover fino ad un dato punto sfruttando solo la vista. A questi veniva in seguito chiesto di raggiungere nuovamente lo stesso punto, però questa volta, oltre al sistema visivo, veniva stimolato anche il sistema tattile. Il percorso scelto la seconda volta si è rivelato essere simile a quello ottimo calcolato dall'algoritmo, evidenziando come il coinvolgimento di più canali sensoriali possa effettivamente rendere i piloti più consapevoli del terreno circostante.

Capitolo 4

Realtà virtuale presso ALTEC

Aerospace Logistics Technology Engineering Company (ALTEC) si occupa da diversi anni dello sviluppo di un'applicazione in VR, chiamata Astra Data Navigator (ADN). Qui di seguito verranno esposte alcune delle principali funzionalità già presenti nel software e verranno descritti i dispositivi hardware utilizzati.

4.1 Navigazione della scena

Una delle funzionalità primarie di ADN è la navigazione dell'Universo, che permette all'utente di osservare i vari elementi che sono stati catalogati. Si possono trovare i pianeti del Sistema solare, i satelliti artificiali, il Sole ed altre stelle. La navigazione dell'ambiente virtuale può avvenire in due modi differenti:

- per mezzo di una ricerca: l'elemento di interesse può essere ricercato dall'utente per nome attraverso un apposito pannello di ricerca. In questo caso, una volta riscontrata l'esistenza dell'elemento, l'applicazione da la possibilità all'utente di avvicinarvisi tramite un'animazione che viene scatenata con la pressione di un tasto. La camera verrà quindi bloccata in automatico sull'elemento selezionato, in modo che questo rimanga sempre al centro della visuale.

- movimento libero: in alternativa l'utente può usare i dispositivi di input per muoversi all'interno del mondo virtuale. In questa modalità si controllano la posizione, l'orientamento e la velocità di movimento dell'osservatore virtuale nella scena.

Inoltre, nel caso in cui l'entità osservata sia un pianeta ci si può avvicinare ulteriormente fino ad entrare in modalità di atterraggio. Quando ci si trova in questa modalità è possibile osservare con maggiore dettaglio la superficie del pianeta, la quale sarà riprodotta tramite dei Digital Elevation Model (DEM), modelli digitali

Figura 4.1. Cattura dello schermo mentre l'applicazione ha appena terminato il caricamento iniziale. La camera punta verso l'origine, posizione nel quale si trova il Sole.

che codificano al loro interno dati altimetrici reali del pianeta cui fanno riferimento, ove presenti.

Un problema che si presenta in applicazioni di questo tipo, dove lo spazio che si vuole rappresentare ha una grande estensione, è quello dell'accuratezza della rappresentazione delle informazioni di posizione ed orientamento degli elementi presenti nel mondo virtuale. In generale, nei game engine queste informazioni vengono salvate in virgola mobile, rappresentazione numerica che purtroppo ha un problema nell'accuratezza. Infatti, più l'oggetto si trova lontano dall'origine del mondo 3D, più cifre vengono perse dopo la virgola, andando a ridurre così la precisione delle informazioni. Nel corso del tempo sono state ideate varie soluzioni al problema:

- cambiamento al volo delle coordinate: questa soluzione prevede di riportare gli oggetti interessati vicino all'origine prima di ogni operazione di rendering o di ogni calcolo in cui è richiesta una certa precisione.

- Coordinate locali multiple: in questo caso si identificano più sistemi di riferimento all'interno del mondo virtuale. La posizione di un oggetto è quindi ricostruita a partire dalle coordinate relative al sistema locale utilizzato, combinate alla posizione del sistema locale nel sistema di riferimento globale.

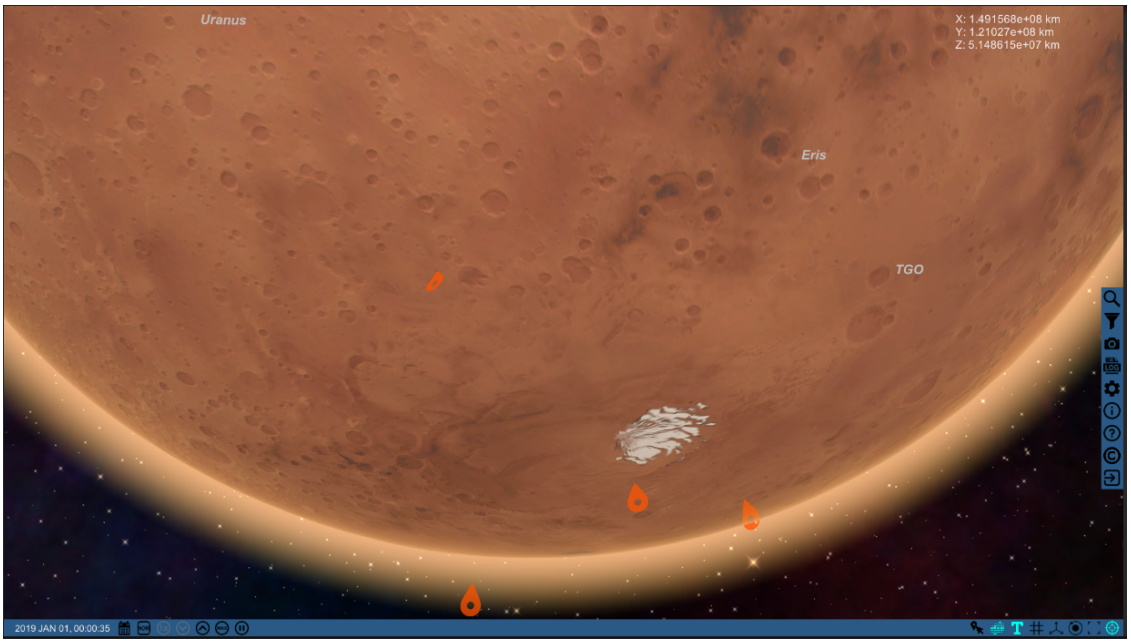

Figura 4.2. Cattura dello schermo mentre l'utente si trova in modalità di atterraggio sul pianeta Marte. La superficie del pianeta è modellata tramite una DEM globale. I punti di interesse rappresentati dai pin in rosso indicano invece la presenza di una DEM locale nella zona. Queste presentano una risoluzione maggiore, ma rappresentano solo una sezione limitata della superficie.

- Origine flottante: questa alternativa prevede che a muoversi nel mondo virtuale non sia la camera, ma tutto il resto, mantenendo così la prima sempre nell'origine e gli altri elementi della scena vicino all'origine, in modo da non perdere precisione nella rappresentazione numerica.

In ADN si fa utilizzo dello Space Graphics Toolkit (SGT), strumento ottenuto dallo Unity Asset Store, che risolve il problema della precisione della rappresentazione in virgola mobile combinando le soluzione dell'origine flottante e quella delle coordinate locali multiple. Il mondo virtuale è diviso in celle il cui lato è pari a $5 \cdot 10^7$ unità e la camera è tenuta sempre vicino all'origine. Questa non viene tenuta fissa però, in quanto lo spostamento degli altri elementi avviene solo quando questa si è mossa di 100 unità. Per identificare la posizione dell'oggetto all'interno della cella vengono quindi utilizzati tre numeri in virgola mobile e le informazioni per identificare la cella vengono invece salvate su tre interi da 8byte.

4.2 Gestione dei cataloghi

Le entità rappresentate nel mondo virtuale sono caricate da diverse fonti. Gli elementi del Sistema solare, dal Sole ai vari pianeti e satelliti, sono caricati da un file `JSON`, soluzione adottata per motivi di semplicità, dato che questo non è destinato a contenere una grande quantità di dati. I dati stellari invece vengono estrapolati da dei cataloghi, quali ad esempio quelli di *Tycho-2*, *Hipparcos* o *Gaia*, messi a disposizione da ESA [21]. Questi contengono informazioni sulle varie stelle mappate nel corso delle omonime missioni e hanno registrato dati per migliaia, se non milioni come nel caso di Gaia, di stelle. Nasce quindi in questo caso la necessità di utilizzare un metodo strutturato per filtrare i dati necessari all'applicazione e salvarli. Questi sono stati salvati su di una base di dati, realizzata tramite la libreria *SQLite3*, la quale implementa tutte le funzionalità di una base di dati relazionale. L'applicazione permette di indicare, tramite un file di configurazione, eventuali cataloghi esterni da cui caricare i dati, che devono essere presenti in formato `CSV` oppure come basi di dati relazionale. Nel caso in cui non sia specificata alcuna fonte alternativa l'applicazione carica un sottoinsieme definito di stelle appartenenti al catalogo Tycho-2. Il caricamento dei dati stellari può avvenire in fase di inizializzazione, durante il caricamento iniziale dell'applicazione, oppure in modo dinamico, in base alla posizione dell'utente. Nel secondo caso vengono caricate solo le stelle che hanno un rapporto tra diametro e distanza dall'osservatore maggiore di una data soglia scelta empiricamente. In questo modo i tempi di caricamento iniziali vengono ridotti ed è possibile gestire più facilmente cataloghi di grandi dimensioni. Lo svantaggio di questa soluzione risiede però nel caricamento necessario quando è richiesta l'operazione di aggiornamento delle stelle visibili.

4.3 Concetto di tempo nella simulazione

Gli elementi del Sistema solare presenti in scena vengono mossi facendo loro seguire le traiettorie che questi descrivono nella realtà. La posizione iniziale dei corpi celesti è impostata coerentemente con il momento scelto come punto iniziale per lo scorrere del tempo nella simulazione. Questo corrisponde al primo gennaio 2000, alle ore 12:00:00.000 del sistema relativistico di coordinate temporali Temps Dynamique Barycentrique (TBD), pensato per tenere conto delle dilatazioni temporali per il calcolo di orbite all'interno del Sistema solare. Questo sistema di riferimento temporale è anche noto come *J2000*. La posizione all'istante iniziale è calcolata tramite gli *SPICE Kernel* (Spacecraft Planet Instrument C-matrix Events). Questi strumenti permettono di ottenere molti parametri di interesse dell'oggetto analizzato, quali ad esempio i vettori di posizione ed orientamento, le dimensioni, la forma e molto altro. Il loro utilizzo richiede però tempi di calcolo non trascurabili. Perciò, quando le orbite vengono replicate per mezzo di questi, si calcola la posizione di

un numero limitato di punti nel tempo e le posizioni intermedie vengono ottenute per interpolazione. Alternativa all'utilizzo degli SPICE sono le leggi di Keplero. Queste non tengono conto di vari fattori e quindi non sono da ritenersi completamente accurate, al contrario del metodo precedentemente presentato, il quale può essere utilizzato quando si ricerca un certo livello di accuratezza nella visualizzazione. L'utente ha la possibilità, per mezzo dell'interfaccia grafica, di modificare la velocità con la quale il tempo scorre all'interno della simulazione. Quando questa aumenta, il calcolo dei punti delle orbite, se si utilizzano gli SPICE, a confronto viene effettuato con frequenza minore, portando ad una perdita di accuratezza. Se si sfruttano le leggi di Keplero invece non è necessaria alcuna operazione particolare ed il tutto funziona con la stessa accuratezza, indipendentemente dalla velocità con cui scorre il tempo.

Figura 4.3. Cattura dello schermo dove sono visibili le orbite dei vari elementi nella zona interna del Sistema solare e parte delle orbite degli elementi situati nelle regioni più remote. Le orbite in verde sono quelle dei pianeti.

4.4 Interfaccia utente

L'interfaccia utente è necessaria per permettere l'interazione con il mondo virtuale ed al contempo fornire informazioni utili. Gli elementi principali dell'interfaccia di ADN sono i seguenti:

- pannello di ricerca: permette all'utente di fare una ricerca per nome delle entità caricate nel mondo virtuale. Su di esso viene poi visualizzata una lista di tutti gli oggetti che corrispondo alla chiave di ricerca.

- pannello informativo: questo compare nel momento in cui si clicca su di un oggetto o a seguito di una ricerca ed avvicinamento espliciti. Su di esso si possono trovare le principali informazioni dell'entità cui fa riferimento, come ad esempio il nome, il tipo e la posizione.

- barra laterale: in questa sono presenti una serie di comandi che permettono all'utente di abilitare o disabilitare alcuni pannelli, come quello di ricerca e dei filtri, di accedere al menu di impostazioni dell'applicazione, catturare la schermata corrente o chiudere l'applicazione.

- pannello dei filtri: permette di selezionare cosa visualizzare nella scena nascondendo selettivamente gli elementi di un dato tipo.

- barra inferiore: qui sono raggruppate sulla sinistra tutte le informazioni e le funzioni inerenti allo scorrere del tempo nella simulazione. A destra invece ci sono altre opzioni, come ad esempio quelle che permettono la visualizzazione delle orbite, degli assi del sistema di riferimento ed altro ancora.

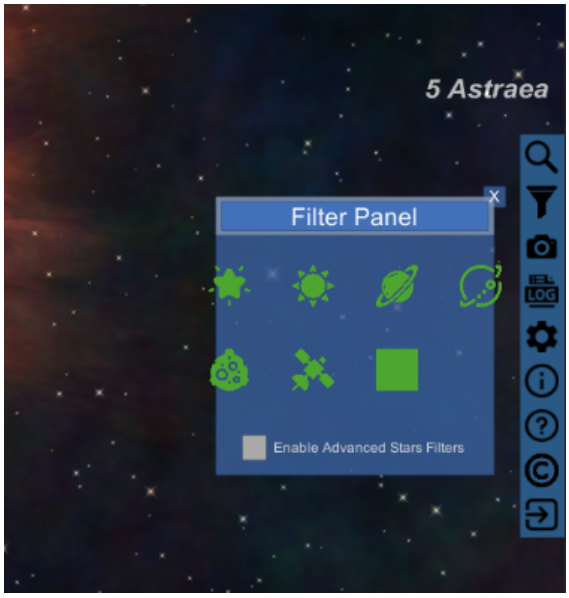

Figura 4.4. In figura la barra laterale ed il pannello dei filtri.

4.5 Hardware

ALTEC ha dedicato alla VR una sala apposita. In essa si trova un proiettore a stereoscopia attiva Barco RLM-W14, con refresh rate pari a $120Hz$ e risoluzione di $1920 \times 1080 pixel$. Il dispositivo in questione è riconducibile alla categoria di dispositivi identificati come *multpilexed ad occhialini*; in particolare sfrutta il multiplexing temporale. Per la corretta visualizzazione sono quindi necessari degli occhialini, i quali vengono sincronizzati con il proiettore tramite segnali ad infrarossi. L'applicazione è stata realizzata per funzionare su di un calcolatore Dell Precision Tower 7810, dotato di due CPU Intel Xeon E5-2650 v3, ognuna delle quali con dieci core e frequenza massima di $2{,}30GHz$, una GPU NVIDIA Quadro K5200 e $32GB$ di RAM.

Capitolo 5

Progettazione e sviluppo

5.1 Struttura generale

Il progetto mira alla creazione di un'infrastruttura in grado di permettere la connessione tra l'applicazione ed un calcolatore esterno, il quale ha lo scopo di inviare comandi agli elementi del mondo virtuale di ADN, in modo da simularne il comportamento. Nel realizzare ciò si è cercato di fornire un'implementazione quanto più generica e flessibile possibile, al fine di permettere la gestione di oggetti di diverso tipo. La struttura è costituita da un file di configurazione, uno script legato ad un oggetto della scena che svolge il ruolo di orchestratore, chiamato `AdnExternalSimulatorManager`, ed uno script legato ad ogni oggetto connesso al calcolatore esterno, chiamato `AdnExternalSimulatorObject`, che espleta alcune funzioni di gestione delle connessioni e chiamata alle funzioni delle singole entità. L'interpretazione e l'implementazione dei comandi che il calcolatore esterno scambia con il singolo elemento non possono essere oggetto di una generalizzazione, per cui è necessario, nel momento in cui si vuole estendere le funzionalità di simulazione ad un determinato oggetto, creare uno script apposito che ne implementi i comandi rispettando il modo di operare della struttura creata.

5.1.1 File di configurazione

Il file di configurazione, denominato `ConfFile.yml`, è stato inserito nella cartella del progetto Unity con lo scopo di contenere tutte le informazioni necessarie all'applicazione per instaurare correttamente la connessione e gestire in modo appropriato gli elementi della simulazione. Il file è in formato `YAML` e contiene essenzialmente un dizionario in cui le chiavi sono una stringa rappresentate il nome dell'oggetto nella scena ed il valore è costituito da una serie di attributi. I campi di ogni elemento del dizionario sono:

- `active`: valore intero, posto a 0 o 1, che indica se l'entità è predisposta alla connessione verso il nodo esterno per la ricezione di comandi.

- `simulationType`: valore intero, posto a 0 o 1, che indica il tipo di simulazione cui può essere sottoposto l'oggetto. Nell'applicazione sono stati identificati due principali tipi di simulazione in base alle caratteristiche dell'elemento che riceve i comandi. Si distingue tra:
  - simulazione nella scena principale: simulazione che parte senza soluzione di continuità al click di un bottone all'interno della scena principale. Ad esempio per l'emulazione delle attività di un satellite in orbita attorno ad un pianeta.
  - Simulazione in una scena separata: simulazione che necessita di un cambio di scena. Questo tipicamente accade nel momento in cui si va a replicare il comportamento di un rover spaziale. Nella nuova scena vengono caricati dinamicamente tutti gli elementi necessari alla simulazione.
- `script`: nome dello script che implementa le funzionalità specifiche dell'elemento. Per garantire la massima flessibilità del sistema è necessario che questo script venga associato all'oggetto durante l'esecuzione e non in fase di compilazione. La sola programmazione ad oggetti e tramite interfacce di C# non permette di avere la flessibilità richiesta. Per questo motivo l'unica opzione valida è la riflessione.
- `IP` e `port`: vengono indicati indirizzo IP e numero di porta per la connessione con il nodo remoto da cui ricevere i comandi. Ogni oggetto ha quindi il suo canale di comunicazione con il nodo o i nodi esterno/i.
- `commands`: dizionario che ha per chiave il valore dei possibili comandi che il nodo esterno può impartire all'oggetto. Per semplicità è stato utilizzato un numero intero incrementale. Il valore relativo al singolo elemento del dizionario contiene i campi:
  - `name`: indica il nome della funzione che implementa il comando all'interno dello script relativo all'entità; in questo modo la funzione può essere invocata per mezzo della riflessione
  - `isPhysics`: valore booleano utilizzato per distinguere due tipi di funzioni: funzioni che utilizzano il motore fisico di Unity e funzioni che invece non lo utilizzano.
  - `parameters`: lista che indica il numero ed il tipo di parametri che il nodo esterno invia, i quali sono necessari per l'implementazione del comando.
- `minTimeSpeed`: valore intero utile per la simulazione dei rover spaziali. Indica il fattore moltiplicativo minimo da mantenere perchè ci sia coerenza tra la velocità del rover e lo scorrere del tempo dettato da `AdnTimeManager`. I rover spaziali possono avere velocità molto basse, cosa che per il motore fisico di

Unity non è semplice da gestire. Se la velocità minima a cui si riesce a far viaggiare il rover non corrisponde a quella reale, il valore del parametro avrà valore maggiore di 1.

5.1.2 Script AdnExternalSimulatorManager

Lo script `AdnExternalSimulatorManager.cs` è associato all'omonimo `gameObject` nella scena di partenza di ADN, `AdnUniverse`. Quando una scena Unity viene rimpiazzata tutti gli elementi in essa contenuti vengono distrutti. Questo oggetto invece viene creato all'avvio dell'applicazione e viene esplicitamente mantenuto in vita durante tutta l'esecuzione della stessa. I compiti che svolge sono:

- gestione della configurazione: lo script si occupa della lettura del file di configurazione e della memorizzazione del contenuto in un dizionario.

- Assegnazione dello script `AdnExternalSimulatorObject.cs`: questo viene associato ai `gameObject` in modo dinamico in fase di esecuzione. Solo gli oggetti il cui relativo parametro `active` nel file di configurazione è a 1 ricevono questo componente. Si distingue inoltre tra oggetti dei due tipi di simulazione: se la simulazione si svolge nella scena principale viene fatto quanto appena detto; se l'oggetto da emulare richiede un cambio di scena allora, dato che questi oggetti non sono di fatto istanziati nella scena principale, ma si tiene solo traccia della loro esistenza, non viene fatto nulla. Il gestore provvederà a collegare lo script `AdnExternalSimulatorObject.cs` a questo tipo di oggetti solo dopo il cambio di scena.

- Avvio della simulazione: lo script si occupa di preparare la simulazione quando richiesto dall'utente. Nel caso questa avvenga nella scena principale vengono fatti alcuni controlli sull'oggetto e viene poi richiesta la connessione con il calcolatore esterno. Nel caso di oggetti da simulare in una scena separata, lo script si occupa di salvare alcuni parametri di interesse per la simulazione prima di richiedere il caricamento della nuova scena. Una volta che la scena è stata caricata in memoria, il gestore si occupa di istanziare l'oggetto, associarvi lo script `AdnExternalSimulatorObject.cs`, richiamare le funzionalità fornite da `AdnExternalSimulatorTerrainManager` per la gestione del terreno, richiedere la connessione con il calcolatore esterno ed avviare la simulazione.

- Gestione di variabili globali dopo un cambio di scena.

5.1.3 Script AdnExternalSimulatorObject

Lo script `AdnExternalSimulatorObject.cs` viene associato ad ogni oggetto definito attivo dal parametro `active` ed espone la seguente funzione.

```
public int EnstConn()
```

Questa viene richiamata da `AdnExternalSimulatorManager` per instaurare la connessione del singolo oggetto con il nodo esterno. Qui tutti i parametri sono letti dal dizionario presente in `AdnExternalSimulatorManager`. Durante l'esecuzione del programma, una volta che è stata stabilita la connessione, il componente si occupa di controllare la presenza di eventuali nuovi messaggi ad ogni ciclo di aggiornamento di Unity. La funzione

```
void ParseComm(int recv)
```

riceve come parametro in ingresso il numero di byte ricevuti e controlla il contenuto del buffer di memoria, andando ad estrarre i singoli messaggi e salvarli in un'apposita coda. Una volta effettuato il controllo di eventuali nuovi messaggi, sempre nella funzione di `Update`, viene eseguito un controllo sulla coda di messaggi presenti. Questi vengono estratti dalla coda e viene chiamata le seguente funzione.

```
public void ApplyCommand(byte[] message)
```

Le operazioni svolte da questa sono la lettura dell'identificativo del comando, corrispondente a quello presente nel file di configurazione, e la lettura del nome della funzione e dei relativi parametri. Questa è scritta appositamente per funzionare con i tipi di dato presenti nel file di configurazione, ossia interi. Nel momento in cui si volesse gestire questi parametri in modo differente è possibile definire una versione alternativa di tale metodo, eseguendo l'*overload*, oppure definire uno strumento per la traduzione del tipo di parametri. Da notare che la funzione richiede un vettore di byte dal quale leggere i comandi. Questo perchè si è scelto di non utilizzare librerie di più alto livello, ma usare direttamente la classe `Socket` messa a disposizione dalla libreria *.NET*. A questo punto il comando viene smistato tra i seguenti dizionari.

```
public Dictionary<String, bool> MoveTowardsFunctions;
public Dictionary<String, bool> PhysicsFunctions;
```

Come detto in precedenza, esistono funzioni che sfruttano il motore fisico, raggruppate all'interno del secondo dizionario, ed altre che non lo fanno, inserite all'interno del primo. Viene effettuato questo smistamento per distinguere il momento in cui queste funzioni, il cui nome corrisponde alla chiave, verranno invocate. Inoltre, il dizionario è utile anche per capire se in un dato momento una data funzione è attiva; da qui l'utilità del valore booleano. Una volta terminata l'esecuzione della funzione avviene l'ultimo passo: viene effettuato il controllo sugli elementi del dizionario `MoveTowardsFunctions`. Se il campo booleano assume valore `true`, allora la funzione è attiva e viene quindi invocata tramite riflessione. Le funzioni che invece utilizzano il motore fisico di Unity, ossia quelle relative al dizionario `PhysicsFunctions`, vengono richiamate tramite riflessione allo stesso modo nella

funzione `FixedUpdate`, messa a disposizione appositamente per le operazioni che coinvolgono la fisica. A differenza della funzione `Update`, la quale viene richiamata ad ogni fotogramma, la `FixedUpdate` viene richiamata dopo un lasso di tempo prestabilito, regolabile tramite le impostazioni di Unity. Infine, viene richiesto che lo script implementante le funzioni del singolo oggetto esponga un metodo per la gestione del cambio di velocità nello scorrere del tempo, dato da `AdnTimeManager`, ed il componente `AdnExternalSimulatorObject` si occupa di invocarlo quando necessario. Questo perchè ogni oggetto potrebbe avere la necessità di svolgere azioni differenti in funzione della variazione di velocità con cui viene fatta scorrere la simulazione.

5.1.4 Simulazione in scena principale

Come accennato, questa modalità prevede che la simulazione avvenga nella scena `AdnUniverse`, senza soluzione di continuità. Al momento dell'avvio dell'applicazione i vari elementi vengono istanziati e sono ricercabili all'interno dell'ambiente virtuale. Gli elementi presenti nel file di configurazione che sono attivi vedranno ad essi associato il componente `AdnExternalSimulatorObject`. Non è in questo momento che viene creata la connessione con il calcolatore remoto. Per avviare la simulazione è necessario ricercare l'oggetto all'interno del mondo virtuale. Partirà quindi un'animazione di avvicinamento all'oggetto ed al contempo comparirà un pannello informativo, il quale mostrerà all'utente alcune informazioni riguardo l'oggetto. A questo punto l'utente, tramite l'interazione con un bottone presente sul pannello, richiederà ad `AdnExternalSimulatorManager` di instaurare la connessione con il nodo remoto ed avviare la simulazione. In questo momento all'oggetto in questione viene anche associato lo script che ne implementa i comandi specifici. In Figura 5.1 è possibile osservare il diagramma di flusso di ciò che è stato descritto. In Figura 5.2 è possibile osservare il Trace Gas Orbiter (TGO) all'interno della scena principale ed il relativo pannello informativo, il quale comunica all'utente che è possibile simulare il comportamento dell'oggetto.

5.1.5 Simulazione in scena secondaria

Per gli elementi che sono associati a questo tipo di simulazione il procedimento è differente. Questi non sono istanziati all'avvio dell'applicazione perchè verrebbero distrutti a seguito del cambio di scena. All'interno dell'applicazione, quando ci si trova nella scena principale, si tiene solo traccia della loro esistenza. L'oggetto viene effettivamente istanziato dopo il cambio di scena e gli viene quindi associato lo script generico `AdnExternalSimulatorObject.cs`. Al contrario di quanto accade per gli altri oggetti, questi non sono direttamente ricercabili dall'utente per quanto appena detto, quindi la serie di azioni necessarie per avviarne la simulazione differisce leggermente. È necessario sapere a quali entità questi sono collegati: nel caso

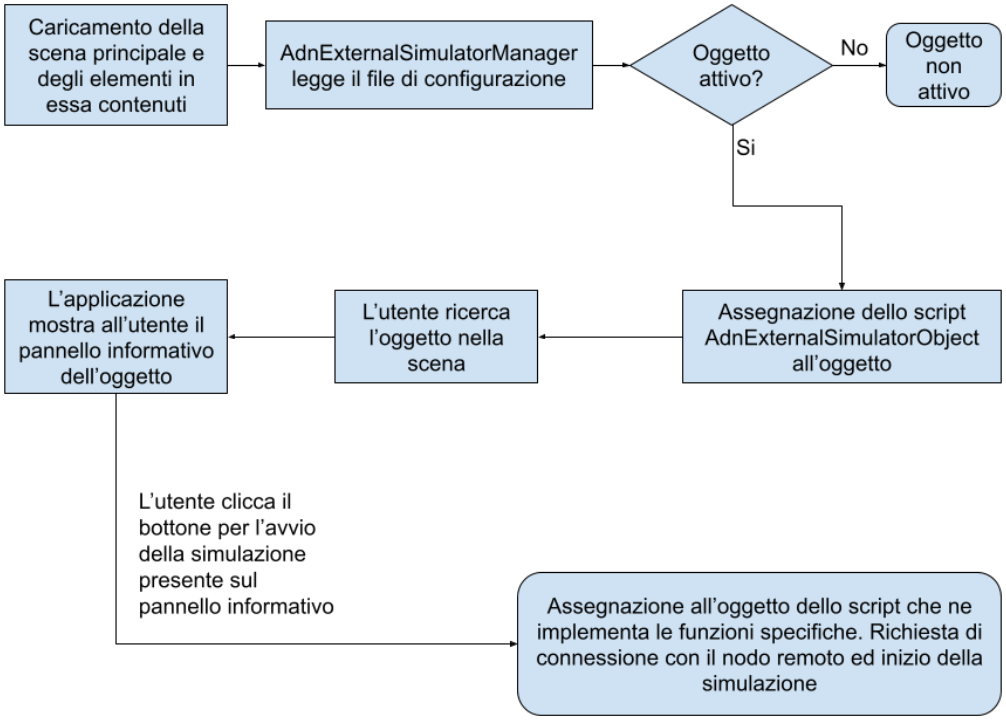

Figura 5.1. Diagramma di flusso che illustra la serie di avvenimenti ed interazioni necessarie all'avvio della simulazione nella scena principale.

di un rover spaziale è necessario conoscere il pianeta su cui questo opera. Va quindi effettuata la ricerca del pianeta e, come nel caso precedente, verrà mostrato il pannello informativo relativo. Da qui l'utente può scegliere di far partire la simulazione con un dato oggetto e su un dato terreno, entrambi selezionabili per mezzo di un menù a tendina. I terreni presenti sono tutti rappresentanti dati reali del pianeta a cui sono associati. In Figura 5.3 è possibile osservare il diagramma di flusso di quanto è stato detto. In Figura 5.4 è possibile osservare il pianeta Marte ed il relativo pannello informativo, il quale comunica all'utente che è possibile simulare il comportamento dell'elemento selezionato su di un DEM a piacere.

5.1.6 Il rover

La struttura descritta è stata studiata per la gestione di oggetti di vario tipo; ciò nonostante, una parte considerevole del tempo è stata dedicata alla realizzazione

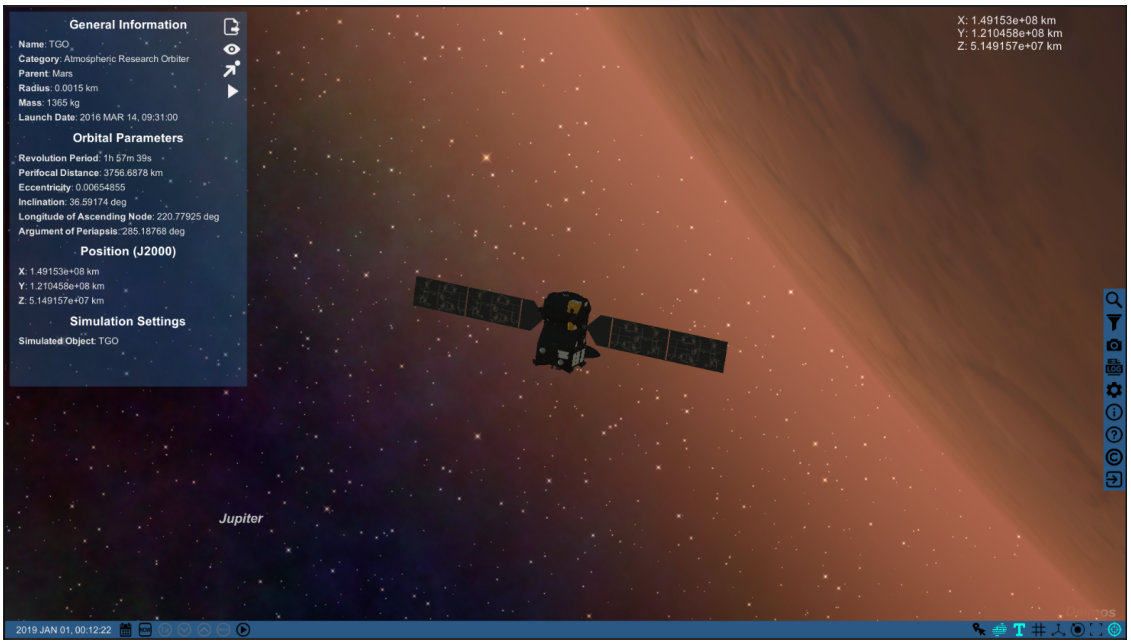

Figura 5.2. Cattura dello schermo durante l'esecuzione dell'applicazione ADN. In figura è possibile osservare il pannello informativo dell'orbiter TGO. La presenza della sezione *Simulation settings* mostra come l'applicazione riconosca la possibilità di simulare il comportamento dell'oggetto.

del modello e della struttura di un rover spaziale: il rover *Rosalind Franklin* di ESA.

Il modello è stato scaricato dal sito di ESA [22]. Questo però si è rivelato non adatto all'animazione in Unity, per cui è stato svolto un lavoro preliminare di separazione delle mesh che costituiscono i vari componenti per essere poi sistemate in una struttura gerarchica adatta allo scopo. Il modello del rover così ottenuto può essere osservato in Figura 5.5

Le caratteristiche fisiche

Per modellare le caratteristiche fisiche del rover sono stati utilizzati i componenti `Rigidbody`, `Mesh Collider` ed infine `Hinge Joint`. Il componente `Rigidbody` è stato assegnato all'elemento `RosalindFranklin`, padre della gerarchia del modello e poi agli elementi `BogieClean`, `WheelClean` e `SteeringClean`. Il componente in questione è utile in quanto permette all'oggetto a cui è associato di rispondere alle leggi fisiche. Per quanto riguarda i 3 componenti `BogieClean`, a questi sono stati assegnati inoltre degli `Hinge Joint`. Questi componenti modellano un giunto meccanico in grado di ruotare attorno ad un asse definito. I due elementi laterali

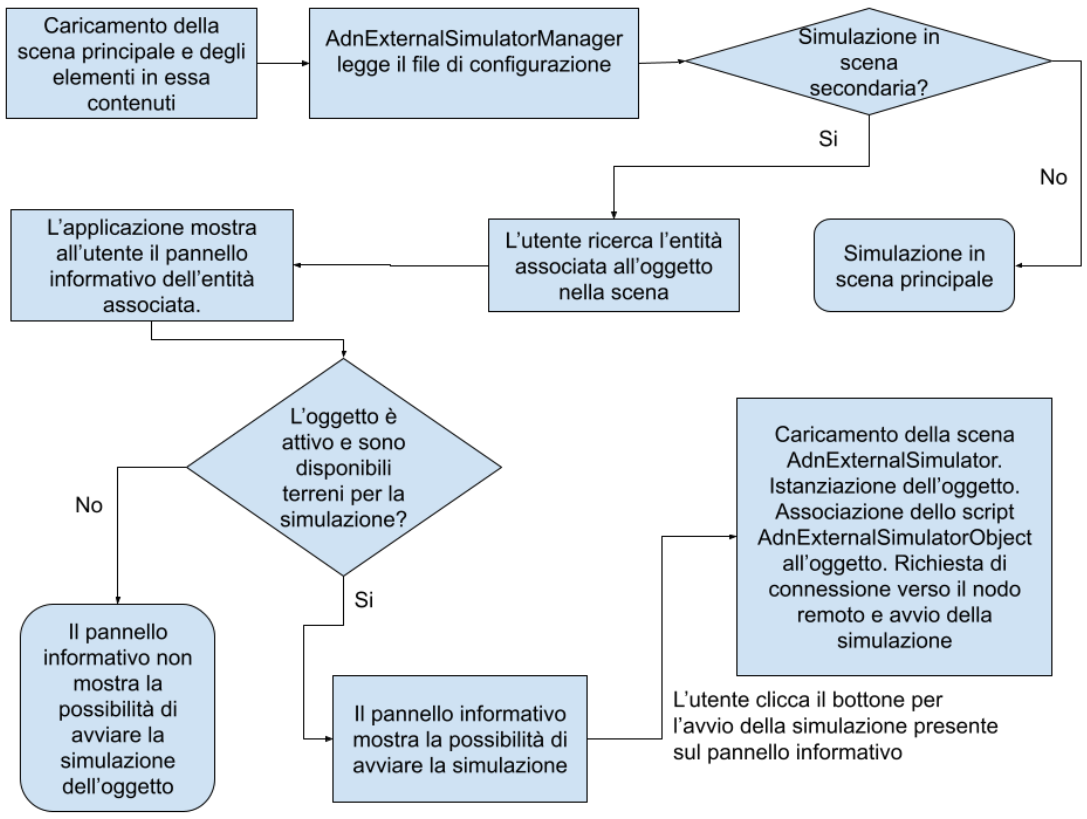

Figura 5.3. Diagramma di flusso che illustra la serie di avvenimenti ed interazioni necessarie all'avvio della simulazione nella scena *AdnExternalSimulator*.

sono liberi di ruotare attorno all'asse `X`, mentre quello posteriore rispetto all'asse `Z`. Da notare che in Unity l'asse `Z` rappresenta la lunghezza, l'asse `X` la larghezza e l'asse `Y` l'altezza. Questa libertà permette ai giunti di far ruotare i bracci meccanici in base alla conformazione del terreno su cui il rover si sta muovendo. Il componente menzionato è stato utilizzato anche per i componenti `SteeringClean`, liberi di ruotare attorno all'asse `Y`, e per gli elementi che costituiscono le ruote, che girano invece attorno all'asse `X`. L'asse di rotazione di questi componenti è passante per il *pivot* della mesh stessa, o anche centro di massa, posizionato accuratamente in fase di modellazione. Infine, i componenti `Mesh Collider`, essenziali per il rilevamento delle collisioni, sono stati utilizzati per l'elemento `platformClean`, `drill_mechanismClean` e per tutte e 6 le ruote. Per i primi due è stata utilizzata la mesh stessa dell'oggetto; per quanto riguarda le ruote invece sono state fatte varie prove, dalla mesh della ruota a degli `Sphere Collider`, ossia collider con la geometria di una sfera. La scelta più adatta per permettere al rover di rientrare nei limiti di velocità, dell'ordine dei $\frac{cm}{s}$, senza gli svantaggi dati dall'enorme sensibilità

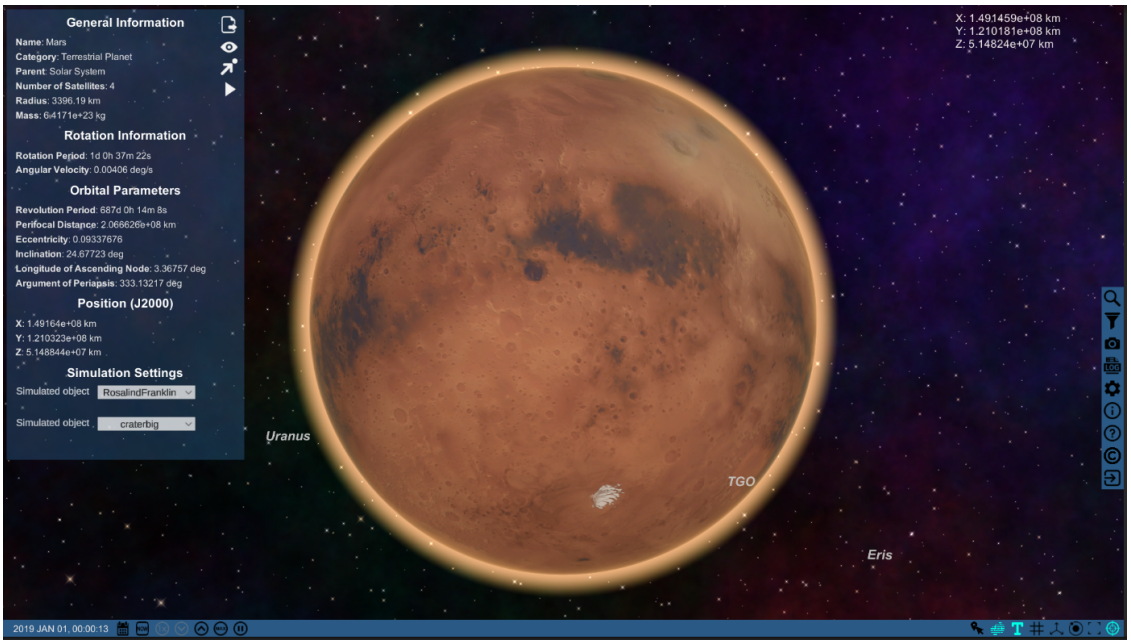

Figura 5.4. Cattura dello schermo durante l'esecuzione dell'applicazione ADN. In figura è possibile osservare nel pannello informativo del pianeta Marte, sotto la voce *Simulation Settings*, la presenza di un menù a tendina che permette la scelta del rover e del terreno per la simulazione nella scena *AdnExternalSimulator*.

degli `Sphere Collider`, si è rivelata essere un `Mesh Collider` la cui geometria è costituita da un cilindro modellato appositamente, con basi da 128 vertici ognuna, visibile in Figura 5.6.

Il movimento

Per implementare il movimento dei vari elementi che costituiscono il rover si fa utilizzo dell'infrastruttura illustrata nella sezione 5.1.3. Le funzioni che non coinvolgono la fisica, come il movimento dei pannelli solari, delle camere e del meccanismo di perforazione del terreno, sono state catalogate come `MoveTowardsFunctions` e vengono invocate all'interno del metodo `Update`. Durante ogni chiamata a questa tipologia di funzioni vengono utilizzati i metodi `MoveTowards` e `RotateTowards`, rispettivamente delle classi `Vector3` e `Quaternion`, i quali spostano o ruotano un oggetto dalla posizione attuale verso quella di destinazione di una piccola quantità ogni volta. L'azione occupa così un certo numero di fotogrammi ed una volta terminata il metodo non viene più invocato. Un'alternativa a questa soluzione potevano essere le `Coroutine` di Unity, create appositamente per dividere l'esecuzione di un'animazione in più fotogrammi. Ciò nonostante, nel manuale stesso di Unity [23]

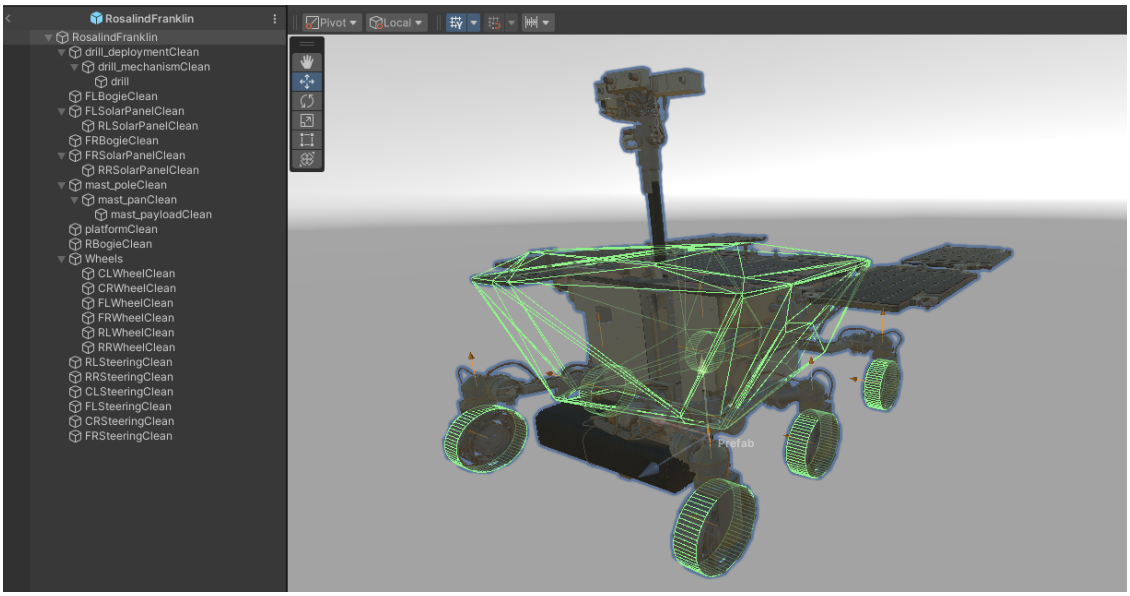

Figura 5.5. Immagine raffigurante il modello 3D del rover Rosalind Franklin utilizzato in ADN.

si raccomanda di usare il minor numero possibile di `Coroutine` per via dell'elevato uso di memoria che queste fanno. Si è quindi scelto di non lavorare con questo strumento, anche per via della semplicità di queste animazioni. Particolare attenzione è da dedicare al movimento del rover e, di conseguenza, al modo in cui vengono gestite le ruote. Le funzioni di movimento sono classificate come funzioni fisiche, quindi come `PhysicsFunctions`, in quanto coinvolgono i componenti `Rigidbody`, `Hinge Joint` e `Mesh Collider`. La gestione di queste funzioni avviene durante il `FixedUpdate`. Il movimento è dato dalla rotazione delle ruote e dall'interazione dei componenti `Mesh Collider` delle stesse con il terreno della scena, il quale, a sua volta, è dotato di un `Collider`. Per innescare un moto rotatorio viene sfruttato il campo `Motor` del componente `Hinge Joint`. I campi assumono per tutte le ruote i valori visibili in Figura 5.6, tranne che per la voce `Connected Anchor`. I campi `Target Velocity` e `Force` sono quelli di maggiore interesse, in quanto è la loro modifica che produce variazioni nella velocità di movimento del rover. Quando `Target Velocity` è a `0` il motore fisico applicherà una forza pari al valore indicato nel campo `Force` perchè la velocità angolare delle ruote sia nulla. Discorso analogo quando si vuole far accelerare o decelerare il rover. Questi valori vengono inoltre moltiplicati per il fattore moltiplicativo che regola la velocità della simulazione, parametro gestito da `AdnTimeManager`, in modo da avere sempre una velocità coerente in relazione allo scorrere del tempo.

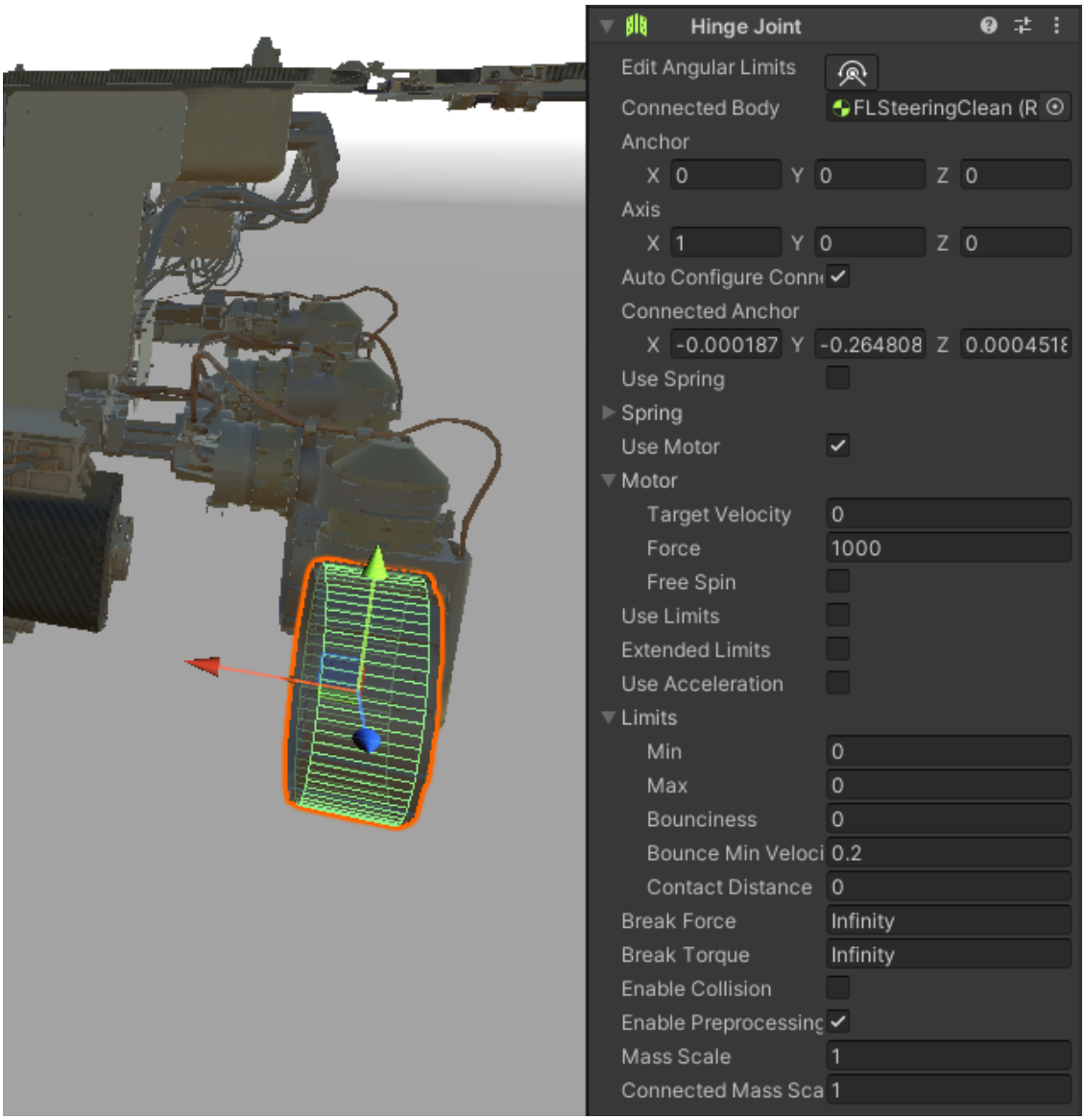

Figura 5.6. Valori del componente *Hinge Joint* della ruota anteriore sinistra del modello. Nell'immagine è possibile notare evidenziata in verde la mesh utilizzata per il componente *Mesh Collider*.

5.1.7 La gestione del terreno

La gestione del terreno all'interno della scena `AdnExternalSimulator` è un compito complesso, per cui è stato deciso di demandarlo ad uno script apposito:

AdnExternalSimulatorTerrainManager.cs

Questo ha il compito di caricare una parte del terreno quando la scena è stata caricata e di gestire l'allocazione e la deallocazione di porzioni di terreno attorno all'oggetto attivo in simulazione. Nella scena vengono utilizzati i `Terrain` di Unity. Questi rappresentano dati di altezze e conformazioni del territorio reali: i valori dei parametri `size` della classe `Unity.TerrainData` dell'oggeto `Terrain`, come anche i valori delle altezze, sono tratti da dei DEM. La seguente funzione messa a disposizione da Unity permette di modificare le altezze dei punti che costituiscono il terreno a partire da una matrice.

```
TerrainData.SetHeights(int xBase, int yBase,
float[,] heights);
```

Il caricamento iniziale

Nel momento in cui la scena `AdnExternalSimulator` viene caricata e l'oggetto della simulazione viene istanziato, bisogna caricare i dati relativi al terreno. Oltre alla generazione del terreno, bisogna svolgere anche una serie di azioni preliminari utili per la gestione dell'allocazione e la deallocazione di porzioni dello stesso. La seguente funzione si occupa di quanto detto.

```
private void LoadTerrainAtStartup()
```

In particolare questa richiama la funzione per la lettura dei dati relativi al terreno selezionato dall'utente, messa a disposizione dallo script di utilità `AdnTiffReader`, la quale restituisce un vettore contenente i dati altimetrici contenuti nel DEM, con i relativi metadati, contenuti invece in un file `JSON` separato. Tra questi, di interesse nel contesto attuale sono:

- `Size`: dimensione della matrice contenente i dati del DEM;
- `PixelSize`: misurato in $\frac{m}{pixel}$, indica quanti metri rappresenta un pixel dell'immagine;
- `HeightInterval`: indica l'intervallo di altezze, dalla minima alla massima, presenti nel DEM.

Dato che i DEM sono di dimensione variabile ed in generale possono esse di grandi dimensioni, ci sono dei problemi con la corretta gestione di questi quando si utilizza un solo `Terrain` di Unity. La massima risoluzione supportata da un singolo `Terrain` è pari a 4097×4097 pixel. Questo comporta due problemi: in primo luogo, per utilizzare un solo terreno bisognerebbe avere un'immagine quadrata, mentre i DEM possono avere un rapporto di forma arbitrario. In secondo luogo, essendo queste immagini spesso di dimensioni maggiori, bisognerebbe effettuare un sottocampionamento. Per evitare ciò è stato scelto di utilizzare più `Terrain` adiacenti

ed il gestore si preoccupa di allocarne un numero sufficiente in prossimità dell'oggetto della simulazione. Si genera quindi una matrice $n \times n$, con $n = 3$, in cui, in corrispondenza dell'elemento centrale, che è un `Terrain` con risoluzione 513×513, si trova l'oggetto della simulazione. Il valore di n, come anche la dimensione della risoluzione del terreno, sono stati scelti in modo arbitrario e sembrano essere un buon compromesso. Per gestire in modo efficiente la creazione e la rimozione di questi `Terrain` in modo dinamico, in base alla posizione dell'oggetto, è stata creata la seguente struttura dati:

```
private class WorldBounds
{
        public float2 bottomLeft;
        public float2 topLeft;
        public float2 topRight;
        public float2 bottomRight;
}
private class DemMatrixElement
{
        public float[,] slice;
        public GameObject terrain;
        public int2 startIndex;
        public WorldBounds bounds;
}
```

L'utilità dei campi che la compongono è:

- `slice`: rappresenta una matrice 513×513 contenente una porzione dei dati del DEM. Nel caso in cui la dimensione dovesse essere minore o la sotto-matrice trovarsi agli estremi della matrice originale, in una posizione tale da andarne oltre i confini, vengono ripetuti i valori presenti ai bordi al fine di evitare errori nell'accesso alla memoria ed effetti grafici poco gradevoli.

- `terrain`: viene salvato il `gameObject` che rappresenta quella fetta di terreno e che quindi contiene il componente `Terrain` generato con i dati della sotto-matrice ad esso relativa.

- `startIndex`: coppia di interi che indica gli indici di partenza da cui è stata prelevata la sotto-matrice nella matrice contenente i dati del DEM.

- `bounds`: contiene i vertici del `gameObject` che rappresenta il terreno nella scena. I vertici sono quindi delle coordinate relative alla scena e non alla matrice contenente i dati altimetrici.

I `DemMatrixElement` sono utilizzati come elementi della matrice `DemMatrix`. Tale matrice viene inizializzata e popolata dalla funzione precedentemente menzionata.

Questa permette di mappare ciò che viene rappresentato nella scena sul contenuto della matrice contenente i dati delle altezze. Inoltre, la posizione dell'oggetto viene mappata nelle coordinate della `DemMatrix`. In questo modo il monitoraggio della posizione dell'oggetto, del calcolo della sua posizione relativa rispetto ai bordi del `Terrain` su cui si trova e quindi la gestione della generazione e distruzione delle porzioni di terreno possono essere gestite in modo semplice ed immediato, come si vedrà nella sezione successiva.

Caricamento di nuove porzioni di terreno

Il gestore del terreno, come detto, si occupa di allocare o deallocare porzioni di terreno sfruttando la struttura dati matriciale precedentemente descritta. Nella funzione `Update` viene effettuato il controllo sulla posizione dell'oggetto e vengono di conseguenza aggiornati i terreni presenti nella scena. Di seguito il contenuto del metodo `Update`.

```
if(StartupCompleted)
{
        int2 index = IsObjectOutOfSliceBounds();
        if (!index.Equals(ObjectIndexInMatrix))
        {
                // the object is in a new portion of the dem
                // need to update indexes and spawn new
                // terrains
                PreviousObjectIndexInMatrix =
                        ObjectIndexInMatrix;
                ObjectIndexInMatrix = index;
                GenerateTerrains();
                DeleteTerrains();
        }
}
```

La funzione `IsObjectOutOfSliceBounds()` esegue una verifica sulla posizione dell'oggetto e ritorna una coppia di indici che indica la posizione dell'oggetto all'interno della `DemMatrix`. Tenendo traccia di questi è possibile capire se l'oggetto si è spostato su una sotto-parte del terreno che fa riferimento ad un altra sotto-sezione della matrice contente i dati altimetrici. Se questo dovesse essere il caso si procede con la generazione di nuovi terreni ed alla cancellazione di quelli che non rientrano più nella matrice $n \times n$ centrata attorno all'oggetto. La generazione in questo caso è molto semplice in quanto si va a fare un controllo sulla `DemMatrix` nell'area $n \times n$ centrata sull'indice dell'oggetto della simulazione. Se il campo `terrain` è vuoto allora viene generato il nuovo terreno. Non c'è bisogno di altre attività preliminari: i dati di altezza, come anche quelli relativi alla posizione che il terreno dovrà assumere nella scena, sono stati calcolati in precedenza e pronti all'uso. Bisogna solo

utilizzare i metodi messi a disposizione da Unity per istanziare il nuovo terreno e modificarne i parametri. La cancellazione di un terreno, in modo duale, prevede il controllo sulla matrice per l'individuazione dei campi `terrain` della `DemMatrix` che contengono un valore ma che sono fuori dalla matrice $n \times n$ attorno all'oggetto. Si provvede quindi a distruggere il `gameObject` e porre `terrain = null`.

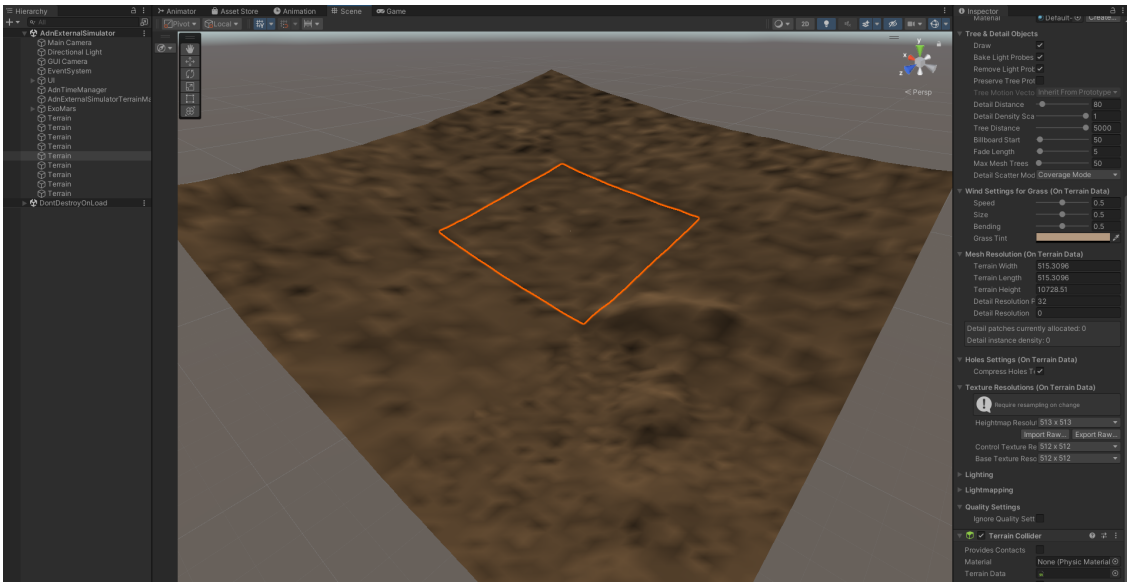

Figura 5.7. Vista della scena *AdnExternalSimulator* che evidenzia la gestione dei terreni. Nell'immagine è selezionato il terreno centrale in una matrice 3×3, al cui centro, talmente piccolo da non essere visibile, si trova il rover Rosalind Franklin. A sinistra, nel pannello *Hierarchy*, si vede come nella scena ci siano 9 terreni distinti.

Capitolo 6

Test e analisi del lavoro

6.1 Problemi per la gestione della fisica

Durante il lavoro di tesi sono emerse varie problematiche legate alla simulazione fisica. Seppure Unity sia un game engine potente e versatile, ottenere delle simulazioni che rispecchiano la realtà e che non siano solo verosimili è un'impresa ardua, se non impossibile. Unity integra al suo interno un physics engine per la gestione della fisica. L'engine utilizzato è *PhysX*, sviluppato da *NVIDIA*, la cui nuova versione punta ad una simulazione fisica sempre più accurata. Bisogna tenere presente però che la versione di Unity utilizzata, 2022.3, non integra al suo interno la versione più recente di PhysX, PhysX 5, il cui SDK è stato rilasciato da NVIDIA nel novembre del 2022. PhysX 5 introduce una serie di migliorie decisamente interessanti, quali la possibilità di definire dei `Collider` che non siano per forza convessi, come anche l'ottimizzazione per il calcolo direttamente su GPU [24]. Per contro, la versione utilizzata in Unity prevede ancora l'utilizzo di `Collider` convessi ed i calcoli fisici sono interamente a carico della CPU.

Nel progetto molta attenzione è stata dedicata all'emulazione del movimento del rover, focalizzandosi in modo particolare sul vincolo della velocità, per il quale sono state ricercate e provate svariate soluzioni realizzative. Dapprima si è cercato di utilizzare i componenti `Wheel Collider`, messi a disposizione da Unity appositamente per la realizzazione di veicoli con ruote. Questi componenti però si sono rivelati non adatti allo scopo per i seguenti motivi:

- rimbalzi: le ruote del rover erano soggette a continui rimbalzi quando a contatto con il terreno, producendo spesso dei comportamenti inaspettati, scorretti e non verosimili. Nonostante la modifica dei parametri al fine di eliminare questo fenomeno, i risultati sono stati poco soddisfacenti.

- Presenza di elementi che modellano gli ammortizzatori: questi componenti sono stati realizzati per modellare il comportamento delle sospensioni di un

veicolo con ruote e sono in questo caso superflui. Inoltre, sono probabilmente i responsabili dei rimbalzi delle ruote.

- Problemi nella gestione della velocità: il vincolo sulla velocità è impossibile da rispettare facendo uso di questi componenti e delle relative API per il loro controllo.

In alternativa si è fatto ricorso alla funzione `AddTorque` e ad un `Mesh Collider` ottenuto dalla geometria della ruota. In questo modo è stato eliminato il problema dei rimbalzi dato dai `Wheel Collider`, ma è rimasto il problema della velocità. Per poter innescare il moto era necessario fornire alle ruote un momento torcente sufficiente a vincere l'attrito iniziale. Il momento fornito però portava il rover ad una velocità maggiore di quella prefissata e l'adattamento del momento fornito durante il moto si è rivelato di difficile gestione. Il tipo di `Collider` utilizzato è stato identificato come il responsabile delle problematiche riscontrate ed è quindi stata ricercata un'alternativa. Tra i vari tipi di `Collider` già presenti in Unity si trova lo `Sphere Collider`. Questo si è rivelato ottimo per la gestione della velocità: è stato possibile ottenere velocità di movimento arbitrariamente basse. Nonostante la possibilità di un controllo fine sulla velocità di movimento, l'eccessiva sensibilità del `Collider`, dovuta al basso attrito statico generato dalla sua geometria, ha introdotto il problema dello slittamento delle ruote. Infine, per i componenti `Collider` delle ruote si è realizzata una geometria tramite il software *Blender*, alla ricerca di un compromesso tra l'eccessiva sensibilità delle sfere e la poca sensibilità delle altre alternative. Ciò nonostante, con la sola sostituzione del `Collider` il problema sul controllo della velocità non è stato risolto. È stata quindi ricercata un'alternativa per la funzione `AddTorque`, che è stata trovata nel campo `Motor` del componente `Hinje Joint`, il quale si è rivelato adatto a mantenere una velocità aderente a quella desiderata.

Oltre alle considerazioni appena fatte, per cercare di ottenere ulteriore stabilità nel movimento delle ruote si è provato ad aggiungere dei vincoli sulla rotazione dei componenti `Rigidbody` delle stesse. Si è cercato di applicare vincoli sulla rotazione attorno agli assi `Y` e `Z`; questi infatti tendevano a disallinearsi dagli assi `Y` e `Z` del corpo centrale del rover. Tali vincoli però non sono stati mantenuti in quanto causavano l'esplosione dell'intero modello non appena a contatto con il terreno; effetto dovuto probabilmente al tipo di algoritmo utilizzato per la gestione del calcolo delle forze dei corpi rigidi.

Oltre ai problemi citati, l'interazione dei componenti `MeshCollider` di ruote e terreno ha generato altri comportamenti inaspettati. Inizialmente il peso assegnato ai `Rigidbody` degli elementi del rover era pari a quello reale. Con questi valori però vi erano dei problemi di compenetrazione tra le ruote del rover ed il terreno dovuti all'interazione dei rispettivi `Collider`, probabilmente causati dal metodo numerico utilizzato dal motore fisico per il calcolo delle collisioni. Il rover, in alcuni casi, cadeva completamente al di sotto del terreno. La massa dei componenti è stata

di conseguenza ridimensionata: al corpo centrale del rover è stata assegnata una massa di $10kg$, mentre agli altri elementi è stata assegnata una massa di $1kg$.

In tutti i casi sono stati applicati vari *vincoli* ai componenti `Rigidbody` degli elementi `Steering` e `Bogie` del modello per cercare di ridurre al minimo eventuali scostamenti dal comportamento aspettato. Un esempio rilevante è il vincolo sulla rotazione degli elementi `Steering`. Questi devono ruotare attorno all'asse `Y` per poter permettere alle ruote di girarsi sul terreno e quindi far cambiare direzione al veicolo. Il comportamento di questi giunti però si è rivelato instabile ed è stato quindi necessario provvedere a muovere questi elementi utilizzando un metodo alternativo. Si è scelto di porre un vincolo proprio sulla rotazione attorno all'asse `Y` e muovere questi elementi tramite la funzione `RotateTowards`, che di fatto non coinvolge il sistema fisico.

In conclusione, la modellazione delle caratteristiche fisiche del rover non è stata priva di problemi: si è speso molto tempo ad aggirare problemi e comportamenti inaspettati dei componenti Unity. Il risultato, seppure visivamente soddisfacente, non si avvicina minimamente ad essere una simulazione fisica affidabile.

6.2 Analisi delle prestazioni

È stato osservato il comportamento dell'applicazione durante l'esecuzione sfruttando l'`Analizer` messo a disposizione da Unity. Il flusso video generato durante l'analisi è stato impostato alla risoluzione di $1920 \times 1080 pixel$. Durante l'analisi sono state monitorate le risorse principali del calcolatore: utilizzo di CPU, GPU, RAM, oltre che alle risorse impiegate dal motore fisico. In Figura 6.1 è possibile osservare il comportamento dell'applicazione appena dopo il caricamento iniziale, con la telecamera del mondo virtuale rivolta verso l'origine della scena, posizione in cui si trova il Sole. Come è possibile osservare, il collo di bottiglia è la CPU, che impiega nel peggiore dei casi poco più di $16ms$ per la generazione del fotogramma. Il carico sulla GPU è invece più basso. La memoria utilizzata in totale dall'applicazione è pari a $2{,}97GB$, di cui $305{,}8kB$ utilizzati dal motore fisico. Si può osservare anche il numero di chiamate alle API fisiche, decisamente poche nel frangente mostrato. In Figura 6.2 è possibile osservare invece l'andamento delle prestazioni nel caso in cui è in corso la simulazione nella scena principale. La schermata riportata è stata catturata durante la simulazione di TGO. Rispetto al caso precedente non ci sono grandi differenze in termini di utilizzo di CPU, GPU ed utilizzo del motore fisico. Si può però vedere un aumento della memoria utilizzata, che è passata da un valore di $2{,}97GB$ ad un valore di $3{,}03GB$. Questo aumento è giustificato dalla necessità di avere in memoria texture, materiali e mesh degli oggetti da visualizzare nella scena, oltre che strutture dati per la gestione della connessione con il nodo esterno e la coda di comandi da eseguire. In Figura 6.3 viene invece mostrato l'utilizzo di risorse dell'applicazione nel caso di simulazione nella scena secondaria, in

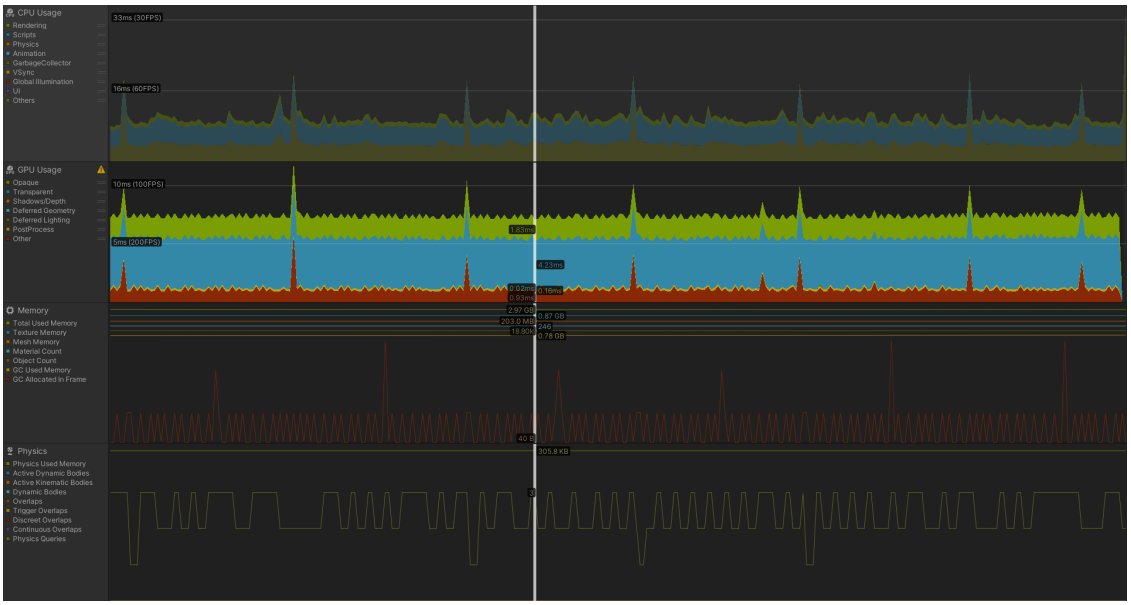

Figura 6.1. Contenuto della schermata dell'*Analizer* di Unity mentre ADN è in esecuzione e l'utente non immette alcun comando.

Figura 6.2. Contenuto della schermata dell'*Analizer* di Unity mentre è in corso la simulazione nella scena principale dell'orbiter TGO.

particolare mentre il rover Rosalind Franklin è in movimento. In questo caso cala

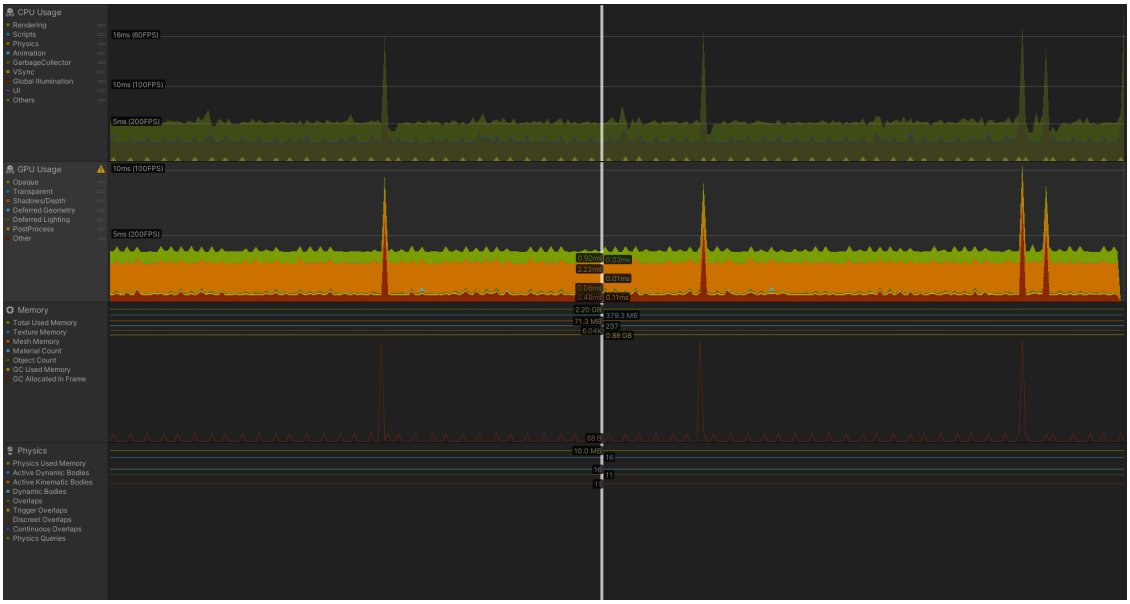

Figura 6.3. Contenuto della schermata dell'*Analizer* di Unity mentre è in corso la simulazione del movimento del rover Rosalind Franklin.

decisamente la memoria allocata, che scende a 2,20GB. Nella scena secondaria sono infatti presenti meno elementi: ci sono il rover, i terreni, i quali sono gli elementi che occupano più memoria, ed i pochi gestori necessari per la simulazione. Il tempo medio richiesto per la generazione dei fotogrammi è leggermente inferiore, portando ad avere un numero più alto di fotogrammi al secondo nella scena. Ciò che invece è aumentato è il numero di chiamate alle API fisiche e la memoria utilizzata dal physics engine, che ha raggiunto i 10MB circa. Può essere interessante soffermarsi un momento sull'analisi delle risorse nel momento in cui avviene il caricamento delle nuove porzioni di terreno nella scena, mostrato in Figura 6.4. In questo caso è evidente la presenza di un picco molto pronunciato in corrispondenza del cursore, momento nel quale sia la CPU, che la GPU, sono impegnate nel caricamento dei nuovi `Terrain` e nella distruzione di quelli più lontani. L'elemento più stressato è sicuramente la CPU: si nota infatti come il tempo impiegato solo per l'esecuzione dello script supera i 100ms. Nonostante questo aumento nel tempo richiesto per la generazione del fotogramma, il calo complessivo dei fotogrammi è poi di fatto impercettibile per l'utente. Al fine di ridurre il tempo richiesto per la generazione del singolo fotogramma potrebbe essere possibile spalmare l'operazione su più fotogrammi, complicando però la logica di gestione. In virtù di quanto appena detto, si è scelto di non modificare la soluzione. Una soluzione preferibile sarebbe stata l'utilizzo di più thread; è però impossibile eseguire queste operazioni si più

Figura 6.4. Contenuto della schermata dell'*Analizer* di Unity in corrispondenza del momento in cui avviene il caricamento delle nuove porzioni di terreno e la de-allocazione di quelle più lontane.

flussi di esecuzione paralleli, in quanto le chiamate alle API di Unity possono essere effettuate solo su quello che viene definito *Main thread*.

Capitolo 7

Considerazioni finali

Il lavoro di tesi svolto ha integrato all'interno dell'applicazione ADN, sviluppata da ALTEC, la possibilità di connettere gli oggetti presenti nel mondo virtuale con un calcolatore esterno. La connessione è finalizzata all'invio di comandi da parte del calcolatore, che saranno poi interpretati ed implementati dagli oggetti della simulazione. Assieme all'infrastruttura generale, la quale permette di gestire svariati tipi di oggetti, sono state realizzate anche delle implementazioni che fanno da esempio: Rosalind Franklin e TGO. Attenzione maggiore è stata rivolta verso la realizzazione di Rosalind Franklin e delle sue funzionalità che sfruttano il physics engine.

Nei capitoli precedenti si è parlato di come i game engine vengano sempre più utilizzati come validi strumenti per la realizzazione di simulazioni di vario tipo. In luce del lavoro svolto e dell'esperienza fatta, non è tuttavia possibile concordare pienamente con ciò. Il game engine Unity è uno strumento versatile e potente, ma con molte limitazioni. Un esempio lampante è la forte limitazione sull'utilizzo dei thread imposta dalla struttura stessa del sfotware. Seppure la rappresentazione grafica sia in grado di raggiungere livelli di realismo molto elevati, la qualità raggiungibile della simulazione fisica è a confronto decisamente deficitaria. È evidente come il suo campo applicativo principe sia quello dei videogiochi. L'esperienza svolta ha evidenziato più volte come il physics engine integrato non sia adatto ad una simulazione accurata della fisica dei corpi rigidi, nonostante i vari compromessi e le varie semplificazioni del modello, oltre che alla semplificazione concettuale dell'interazione con il terreno, trattato come un semplice corpo rigido. Queste considerazioni valgono per Unity e non per altri game engine. Ciò nonostante, è difficile immaginare che la qualità realizzativa delle soluzioni fornite da altri prodotti simili sia tale da permettere una simulazione fisica in piena regola. Nonostante questo, i game engine sono strumenti promettenti che vanno via via migliorando e sfruttano in maniera sempre più efficace la potenza hardware ed in particolare delle sempre più performanti GPU. Sebbene i risultati siano stati deludenti e non comparabili con quelli dei software dedicati, è ancora presto per bocciare definitamente prodotti di questo tipo nel campo delle simulazioni.

Infine, tornando ad ADN, possibili sviluppi futuri possono essere finalizzati a:

- integrazione di un modello migliorato del rover e delle sue funzionalità, magari quando verrà introdotto nelle nuove versioni di Unity il supporto a PhysX 5;
- implementazione di un server in grado di codificare ed inviare comandi più articolati con lo scopo di ottenere delle simulazioni sempre più complesse;
- integrazione di altri oggetti interattivi oltre ai due già presenti.

Appendice A

AdnExternalSimulatorObject

A.1 EnstConn

```
Ip = new IPEndPoint(IPAddress.Parse(
    AdnExternalSimulatorManager.Connections[this.name].ip),
    AdnExternalSimulatorManager.Connections[this.name].port);
Socket = new Socket(Ip.AddressFamily,
                SocketType.Stream,
                ProtocolType.Tcp);
while (!Socket.Connected) {
    try
    {
        Socket.Connect(Ip);
    }
    catch (SocketException)
    {
        Socket.Close();
        return -1;
    }
}
byte[] idForServer = System.Text.Encoding.UTF8.GetBytes(this.name);
Socket.Send(idForServer);
Debug.Log("Connected to ip " + Ip.ToString());

// attach script to game object
string scriptName = AdnExternalSimulatorManager.
    Connections[this.name].script;
ScrType = Type.GetType(scriptName);
gameObject.AddComponent(ScrType);

// Initialize globals
```

```
Buffer = new byte[1024];
Support = new byte[1024];
NByteLen = AdnExternalSimulatorManager.Connections[this.name].nByteLen;
CommandsMapper = AdnExternalSimulatorManager.
    Connections[this.name].commands;
AdnTimeManager.SimulationTimeSpeedChanged +=
    SetObjectBehaviourWhenSpeedChanges;

return 0;
```

A.2 ParseComm

```
int messageLen = 0;
int byteRecv = recv;
while (true)
{
    try
    {
        // extrapolate message len and single message content
        messageLen = ParseNumber(Support, NByteLen, 0);
        byte[] message = new byte[messageLen - NByteLen];
        Array.ConstrainedCopy(Support, NByteLen, message,
            0, messageLen - NByteLen);

        Messages.Enqueue(message);

        if (byteRecv == messageLen)
        {
            Support = new byte[1024];
            break;
        }
        else if (byteRecv > messageLen)
        {
            byte[] tmpArray = new byte[byteRecv - messageLen];
            Array.ConstrainedCopy(Support, messageLen,
                tmpArray, 0, byteRecv - messageLen);
            Support = tmpArray;
            byteRecv -= messageLen;
        }
    }
    catch (ArgumentException)
    {
        // the exception will occour if the message in not completely read
        Support = new byte[1024];
```

```
    }
}
```

A.3 ApplyCommand

```
int commId = ParseNumber(message, LenCommId, 0);
var functName = CommandsMapper[commId].name;
List<string> parameters = CommandsMapper[commId].parameters;
Debug.Log(functName);

int startIndex = LenCommId;
for (int i = 0; i < parameters.Count; i++)
{
    (object commAmt, int offset) =
        ParseNumber(message, parameters[i], startIndex);
    Targets.Add(commAmt);
    startIndex += offset;
}

if (CommandsMapper[commId].isPhysics)
{
    if (!PhysicsFunctions.ContainsKey(functName))
        PhysicsFunctions.Add(functName, true);
    else
        PhysicsFunctions[functName] = true;
}
else
{
    if (!MoveTowardsFunctions.ContainsKey(functName))
        MoveTowardsFunctions.Add(functName, true);
    else
        MoveTowardsFunctions[functName] = true;
}

FirstCall = true;

return;
```

Appendice B

AdnExternalSimulatorTerrainManager

B.1 LoadTerrainAtStartup

```
NativeArray<byte> data = new NativeArray<byte>();
AdnTiffReader.ReadTiffData(DemInfo, ref data);
int2 terrainSize = new int2(DemInfo.Size);
float2 pixelSize = new float2(DemInfo.PixelSize.x,
    Math.Abs(DemInfo.PixelSize.y));
float range = DemInfo.HeightInterval.Max - DemInfo.HeightInterval.Min;
Center = new float2(terrainSize.x * pixelSize.x / 2,
    terrainSize.y * pixelSize.y / 2);

GenerateDemMatrix(data, range, pixelSize);

ObjectIndexInMatrix = new int2(DemMatrixSize.x / 2, DemMatrixSize.y / 2);

GenerateTerrains();

data.Dispose();

StartupCompleted = true;
```

B.2 GenerateDemMatrix

```
int xLen = DemInfo.Size.x / 2 - HeightmapResolution / 2;
int yLen = DemInfo.Size.y / 2 - HeightmapResolution / 2;

int xCells = xLen / HeightmapResolution;
int yCells = yLen / HeightmapResolution;

DemMatrixSize = new int2(xCells * 2 + 1, yCells * 2 + 1);
```

```
DemMatrix = new DemMatrixElement[DemMatrixSize.x, DemMatrixSize.y];

for (int i = 0; i < DemMatrixSize.x; i++)
    for (int j = 0; j < DemMatrixSize.y; j++)
    {
        int2 startIndex = MapMatrixIndexToDemIndex(i, j);
        DemMatrix[i, j] = new DemMatrixElement();
        DemMatrix[i, j].startIndex = startIndex;
        DemMatrix[i, j].slice = GetSubmatrix(startIndex,
            HeightmapResolution, data, range);
        DemMatrix[i, j].bounds = ComputeBounds(startIndex,
            HeightmapResolution, pixelSize);
    }

return;
```

B.3 IsObjectOutOfSliceBounds

```
float posX = ActiveSimulationObject.transform.position.x;
float posY = ActiveSimulationObject.transform.position.z;
WorldBounds bounds =
    DemMatrix[ObjectIndexInMatrix.x, ObjectIndexInMatrix.y].bounds;

int newX = ObjectIndexInMatrix.x;
int newY = ObjectIndexInMatrix.y;

if(posX < bounds.bottomLeft.x)
    newX -= 1;
else if(posX > bounds.topRight.x)
    newX += 1;

if(posY < bounds.bottomLeft.y)
    newY += 1;
else if(posY > bounds.topRight.y)
    newY -= 1;

newX = Math.Clamp(newX, 0, DemMatrixSize.x - 1);
newY = Math.Clamp(newY, 0, DemMatrixSize.y - 1);

return new int2(newX,newY);
```

B.4 GenerateTerrains

```
// Generate N x N submatrix of terrains
int startX = Math.Clamp(ObjectIndexInMatrix.x - SquareSubMatrixDim / 2,
    0, DemMatrixSize.x - 1);
int endX = Math.Clamp(ObjectIndexInMatrix.x + SquareSubMatrixDim / 2,
    0, DemMatrixSize.x - 1);
int startY = Math.Clamp(ObjectIndexInMatrix.y - SquareSubMatrixDim / 2,
    0, DemMatrixSize.y - 1);
int endY = Math.Clamp(ObjectIndexInMatrix.y + SquareSubMatrixDim / 2,
    0, DemMatrixSize.y - 1);

for (int i = startX; i<= endX; i++)
{
    for(int j = startY; j<= endY; j++)
    {
        if (DemMatrix[i,j].terrain == null)
        {
            // spawn new terrain
            SpawnTerrain(i, j);
        }
    }
}
```

B.5 SpawnTerrain

```
float[,] submatrix = DemMatrix[i, j].slice;

TerrainData newTerrainData = new TerrainData();
GameObject newTerrainObject = Terrain.
    CreateTerrainGameObject(newTerrainData);
DemMatrix[i,j].terrain = newTerrainObject;

newTerrainData.heightmapResolution = HeightmapResolution;
float range = DemInfo.HeightInterval.Max - DemInfo.HeightInterval.Min;
newTerrainData.size = new Vector3(HeightmapResolution * DemInfo.PixelSize.x,
    range, HeightmapResolution * Math.Abs(DemInfo.PixelSize.y));
newTerrainData.SetHeights(0, 0, submatrix);

// change terrain appearence
String layerName = DemInfo.Parent.Name + "Soil";
List<TerrainLayer> layers = new List<TerrainLayer>();
layers.Add((Resources.Load<TerrainLayer>("Textures/" + layerName)));
```

```
newTerrainData.terrainLayers = layers.ToArray();

float posX = DemMatrix[i, j].bounds.bottomLeft.x;
float posY = DemMatrix[i, j].bounds.bottomLeft.y;

DemMatrix[i, j].terrain.transform.position = new Vector3(posX, 0, posY);
DemMatrix[i, j].terrain.GetComponent<Terrain>().allowAutoConnect = true;
```

Acronimi

ADN Astra Data Navigator. 4, 5, 9, 39, 41, 43, 47, 49, 53, 55, 56, 66, 71, 72

ALTEC Aerospace Logistics Technology Engineering Company. 5, 9, 39, 45, 71

API Application programming interface. 10, 64, 65, 67, 69

AR Augmented Reality. 22, 28–31

ASI Agenzia Spaziale Italiana. 32

AV Augmented Virtuality. 22

CES Consumer Electronics Show. 25

CPU Central Processing Unit. 45, 63, 65, 67

CRT Cathod Ray Cube. 14

DEM Digital Elevation Model. 9, 39, 41, 52, 58, 59

DOF Degrees of Freedom. 21, 36

DOUG Dynamic On-board Ubiquitous Graphics. 26, 27

DST Dynamic Skills Trainer. 29

EAC European Astronaut Centre. 28, 29

ESA European Space Agency. 25, 27–30, 42, 53

EV Extra Vehicular. 28

EVA Extra Vehicular Activity. 26, 28

FoV Field of View. 12, 18, 20

GPU Graphics Processing Unit. 15, 45, 63, 65, 67, 71

HMD Head Mounted Display. 14, 17, 20, 21, 26, 29

IP Internet Protocol. 48

IPD Interpupillary Distance. 13, 20

ISS International Space Station. 26–30

JIVE Joint Investigation into VR for Education. 28, 29

JPL Jet Propulsion Laboratory. 32

LCD Liquid Crystal Display. 19, 20

LED Light Emitting Diodes. 24

MR Mixed Reality. 21, 22, 28

NASA National Aeronautics and Space Administration. 25, 28, 29, 31, 32

RAM Random Access Memory. 45, 65

ROAMS Rover Analysis, Modeling and Simulation. 32

SAFER The Simplified Aid for EVA Rescue. 26, 27

SDK Software Development Kit. 63

SGT Space Graphics Toolkit. 41

SPICE Spacecraft Planet Instrument C-matrix Events. 42, 43

TBD Temps Dynamique Barycentrique. 42

TGO Trace Gas Orbiter. 4, 51, 53, 65, 66, 71

VIVED Virtual Visual Environment Display. 25

VR Virtual Reality. 3–5, 9, 11–17, 21, 22, 25–30, 32, 36, 39, 45

VRLab Virtual Reality Laboratory. 26–28

XR eXtended Reality. 28

XRLab eXtended Reality Laboratory. 28

Glossario

3D Tridimensionale. 4, 13, 15, 17, 19, 20, 26, 29, 34, 40, 56

game engine Strumento in grado di gestire gli aspetti di basso livello, quali comunicazioni tra librerie software, inizialmente pensato per essere utilizzato durante la fase di sviluppo di videogiochi. 3–5, 10, 15, 23, 24, 33–35, 40, 63, 71

mesh In informatica grafica una mesh è un insieme di vertici, i quali compongono le facce di un oggetto e ne definiscono quindi la struttura nello spazio. 53, 54

microgravità Condizione per la quale il campo gravitazionale è molto basso o assente. Questa è tipicamente replicabile a bordo di veicoli spaziali in orbita attorno al pianeta, come la ISS. 27, 30

mockup Modello in scala, o a grandezza originale, di un dispositivo, utilizzato a scopi dimostrativi, educativi o di adestramento. 26, 33

multiplexing Processo di divisione dell'informazione visiva in flussi video distinti, poi percepiti dall'utente e ricostruiti dal cervello di nuovo in un unico flusso. 18

parallasse Spostamento angolare apparente di un oggetto quando osservato da due punti di vista differenti. 13, 19

physics engine Strumento in grado di eseguire calcoli semplificati per la gestione fisica degli elementi di un mondo virtuale. Generalmente integrato in un game engine. 3, 4, 10, 15, 23, 24, 33, 63, 67, 71

polarizzazione Proprietà delle onde elettromagnetiche che indica l'orientamento delle oscillazioni dell'onda nello spazio. 19

refresh rate Frequenza di aggiornamento di uno schermo misurata in hertz. 19, 45

rendering Processo di generazione dell'immagine bidimensionale a partire dagli elementi del mondo virtuale. 23, 40

riflessione Capacità di un processo di elaborare il programma stesso e la propria struttura. 48, 50

script Tipo particolare di programma, che tipicamente sfrutta un linguaggio interpretato. 35, 47–49, 51, 57, 58

stereopsi Processo di fusione delle delle immagini provenienti dagli occhi in un unica immagine ad opera del cervello. L'immagine risultante permette di percepire la profondità. 13

Bibliografia

[1] Neanias. https://www.neanias.eu/. Visitato il 10 Giugno 2024.

[2] Grigore C Burdea and Philippe Coiffet. *Virtual reality technology*. John Wiley & Sons, 2003.

[3] D. Broderick. *The Judas Mandala*. Fantastic Books, 2009.

[4] Persistenza su enciclopedia treccani. https://www.treccani.it/enciclopedia/persistenza/. Visitato il 24 Aprile.

[5] Milgram et al. Merging real and virtual worlds. In *Proceedings of IMAGINA*, volume 95, pages 218–230, 1995.

[6] Forging new paths for filmmakers on "the mandalorian". https://www.unrealengine.com/fr/blog/forging-new-paths-for-filmmakers-on-the-mandalorian. Visitato il 7 Maggio 2024.

[7] Garcia et al. Training astronauts using hardware-in-the-loop simulations and virtual reality. pages 1–7, 01 2020.

[8] Better SAFER then sorry. https://www.esa.int/ESA_Multimedia/Images/2023/07/Better_SAFER_than_sorry. Visitato il 29 Aprile 2024.

[9] Exploring new realities: ESA's XR lab. https://www.esa.int/About_Us/EAC/Exploring_new_realities_ESA_s_XR_lab. Visitato il 30 Aprile 2024.

[10] Stephan Ghiste, Christopher Scott, Sommy Khalaj, Tommy Nilsson, Hanjo Schnellbächer, Leonie Bensch, and Andrea Casini. Using the method of loci in virtual reality to reduce robotic operations training time of astronauts. 09 2022.

[11] Ready, set, go for COVID-conscious astronaut training. https://www.esa.int/About_Us/EAC/Ready_set_go_for_COVID-conscious_astronaut_training. Visitato il 30 Aprile 2024.

[12] Nine Ways We Use AR and VR on the International Space Station. https://www.nasa.gov/missions/station/nine-ways-we-use-ar-and-vr-on-the-international-space-station/. Visitato il 30 Aprile 2024.

[13] Better than reality: Nasa scientists tap virtual reality to make a scientific discovery. https://www.nasa.gov/technology/better-than-reality-nasa-scientists-tap-virtual-reality-to-make-a-scientific-discovery/.

[14] PointCloudsVR. https://github.com/nasa/PointCloudsVR/tree/master. Visitato il 30 Aprile 2024.

[15] A. Jain, J. Balaram, J. Cameron, J. Guineau, C. Lim, M. Pomerantz, and G. Sohl. Recent developments in the roams planetary rover simulation environment. In *2004 IEEE Aerospace Conference Proceedings (IEEE Cat. No.04TH8720)*, volume 2, pages 861–876 Vol.2, 2004.

[16] Lucio Pinello, Lorenzo Brancato, Marco Giglio, Francesco Cadini, and Giuseppe Luca. Enhancing planetary exploration through digital twins: A tool for virtual prototyping and hums design. *Aerospace*, 11:73, 01 2024.

[17] Shih-Chung Kang, Jhih-Ren Juang, and W. Hung. Using game engine for physics-based simulation - a forklift. *Journal of Information Technology in Construction*, 16:3–22, 01 2011.

[18] Jonas Sørensen, Zheng Ma, and Bo Jørgensen. Potentials of game engines for wind power digital twin development: an investigation of the unreal engine. *Energy Informatics*, 5:39, 12 2022.

[19] Ziran Wang, Kyungtae Han, and Prashant Tiwari. Digital twin simulation of connected and automated vehicles with the unity game engine. 07 2021.

[20] Yuzhen Xie, Zihan Tang, and Aiguo Song. Motion simulation and human–computer interaction system for lunar exploration. *Applied Sciences*, 12:2312, 02 2022.

[21] Gaia Archive. `https://gea.esac.esa.int/archive/`. Visitato il 17 Giugno 2024.

[22] ESA. `https://www.esa.int/`.

[23] Unity Manual. `https://docs.unity3d.com/Manual/Coroutines.html`. Visitato il 15 Maggio 2024.

[24] Open Source Simulation Expands with NVIDIA PhysX 5 Release. `https://developer.nvidia.com/blog/open-source-simulation-expands-with-nvidia-physx-5-release/`. Visitato il 22 Maggio 2024.