ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

ESTENSIONE DI UN'INFRASTRUTTURA PER AMBIENTI COOPERATIVI DI MIXED REALITY: INTEGRAZIONE DEL FRAMEWORK GOOGLE ARCORE E TECNOLOGIE MOBILE AVANZATE

Tesi in
PROGRAMMAZIONE AVANZATA E PARADIGMI

Relatore
Prof. ALESSANDRO RICCI

Co-relatore
Dott. Ing. ANGELO CROATTI

Presentata da
MATTIA CAPUCCI

Anno Accademico 2018 – 2019

Alla mia famiglia

Indice

Introduzione

La grande rivoluzione tecnologica che è avvenuta negli ultimi anni ha portato la realtà aumentata a essere uno degli argomenti di studio di maggior interesse. La convinzione di poter eseguire operazioni quotidiane coadiuvate da un insieme di elementi virtuali e digitali si è concretizzata nello sviluppo e rilascio di dispositivi fisici – quali visori – e kit di sviluppo software. Quest'ultimi, in particolare, hanno recentemente raggiunto una maturità tale da permettere a device di tutti i giorni, quali smartphone e tablet, di poter sfruttare tecniche avanzate di analisi delle immagini, consentendo di essere anch'essi il veicolo per esperienze di augmented e mixed reality. Ci stiamo avvicinando a una realtà in cui elementi digitali coesistono nel mondo reale e le persone sono in grado di percepirli e interagire con essi, in modo condiviso.

In questo contesto emergente, la ricerca è indirizzata nello studio di un modello e/o un'infrastruttura, che permetta di sfruttare le potenzialità delle tecnologie oggi disponibili e possa essere un punto di riferimento nello sviluppo di sistemi di augmented e mixed reality. Su questo fronte, spicca il modello di *augmented worlds*, punto di partenza del lavoro di questa tesi. Infatti, dall'analisi di detto modello e dell'infrastruttura a esso associata, si propone una possibile estensione che permetta a un qualsiasi utente, dotato di smartphone o tablet, di accedere a esperienze cooperative di mixed reality, incapsulando le funzionalità di comprensione dell'ambiente e motion tracking, che sono tipiche nelle tecnologie software mobili odierne. A tale fine, il lavoro di questa tesi si suddivide in otto capitoli.

Nel primo capitolo si presenta un'overview del percorso intrapreso, focalizzandosi sui concetti chiave che verranno trattati e presentando dettagliatamente l'obiettivo della tesi.

Nel secondo capitolo si descrive il contesto della realtà aumentata e mista, le applicazioni odierne e future nella vita quotidiana e i dispositivi hardware abilitanti a questa rappresentazione innovativa dell'ambiente. Si presenta, inoltre, una tematica centrale quando si tratta di applicazioni di augmented/mixed reality, ovvero la gestione della registrazione e sincronizzazione con il mondo fisico. Infine, si descrivono le principali tecnologie software che abilitano i device mobili all'AR.

Nel terzo capitolo si introduce il concetto di *augmented worlds*, come modello per produrre sistemi pervasivi di mixed reality basati sugli agenti e si definisce il concetto di *mirror worlds*, da cui trae le sue origini.

Nel quarto capitolo si propone MiRAgE, ovvero il primo framework che permette la creazione e l'esecuzione di mondo aumentati. Si fornisce una descrizione dell'architettura e dei principali componenti, focalizzandosi sugli elementi che saranno oggetto di aggiornamento o estensione durante lo svolgimento del progetto di tesi.

Nel quinto capitolo si fornisce un'analisi e una descrizione approfondita della piattaforma di realtà aumentata ARCore, sviluppata da Google. Essa rappresenta il caso di studio della tesi come tecnologia mobile abilitante all'AR e verrà integrata in MiRAgE al fine di migliorare la rappresentazione e l'ancoraggio degli ologrammi nell'ambiente fisico.

Nel sesto capitolo si entra nel vivo del progetto per la tesi, analizzando il problema proposto e fornendo, a livello di progettazione, le possibili soluzioni adottabili. Il focus è rivolto a come deve avvenire l'integrazione della piattaforma in MiRAgE, la strategia adottata per l'accesso e la rappresentazione del mondo digitale e, infine, come gestire l'interazione dell'utente.

Nel settimo capitolo si procede a descrivere i principali dettagli implementativi che sono stati applicati per raggiungere gli obiettivi prefissati, seguendo l'iter proposto nella fase di analisi e progettazione.

Nell'ottavo capitolo, infine, si descrive il processo di validazione del framework prodotto, mostrando i diversi test effettuati e gli esempi ad-hoc sviluppati. Viene presentato, inoltre, il caso di studio di Rocca delle Caminate, il cui contesto è stato utilizzato per realizzare un progetto che potesse valorizzare il patrimonio culturale, inserendo elementi e caratteristiche di mixed reality.

Capitolo 1

Dalla realtà aumentata a mondi aumentati multi-utente cooperativi

1.1 La realtà aumentata come argomento di ricerca

I recenti progressi nello sviluppo di tecnologie e dispositivi, che abilitano gli utenti a interagire con il mondo fisico, hanno aperto la strada a studi di ricerca nel campo dell'*augmented* e *mixed reality*.

Con augmented reality si intende la possibilità di sovrapporre al mondo fisico contenuti digitali. L'ambiente è, quindi, arricchito con elementi virtuali, visibili dall'utente in base alla propria posizione e prospettiva.

La mixed reality, invece, è un ulteriore raffinamento della realtà aumentata, in cui abbiamo una fusione tra mondo reale e virtuale. In questa visione, gli elementi digitali non sono solamente entità a sé stanti, bensì essi si integrano con l'ambiente, rafforzando l'illusione di coesistenza tra mondo fisico e digitale.

Gli utenti, immersi all'interno dell'ambiente, possono percepire e, al contempo, interagire sia con oggetti fisici sia con elementi virtuali, diventando quest'ultimi parte integrante dello spazio in cui sono localizzati.

Al fine di permettere all'utente di accedere a realtà di mixed reality, sono stati introdotti nel mercato diversi prodotti. Un esempio è dato da *Hololens* di Microsoft, un head-mounted display che consente, una volta indossato come un casco, di avere una rappresentazione del mondo virtuale direttamente sulla vista dell'utente.

Negli ultimi anni, inoltre, vi è stato un progresso rilevante a livello software, che ha portato al rilascio di tool e SDK che permettessero agli sviluppatori di

creare applicazioni che fornissero elementi di AR/MR. Tra questi, *ARCore* di Google e *ARKit* di Apple rivestono un ruolo centrale, in quanto hanno integrato meccanismi avanzati di tracciamento della posizione e comprensione dell'ambiente tali da permettere a dispositivi di tutti i giorni, quali tablet e smartphone, di essere il tramite per l'utente alla mixed reality.

Ci stiamo avvicinando a un nuovo concetto di ambiente, definibile *smart space*, in cui sia il livello fisico, che quello digitale e sociale sono strettamente intrecciati tra loro. Da questa punto di vista, l'obiettivo della ricerca è fornire un modello che possa esprimere questa tipologia di smart environment.

Tra i lavori in questo campo, è rilevante il concetto di *Mirror World* in cui gli oggetti fisici, percepibili nel mondo reale dagli umani, hanno un'estensione digitale in modo tale che possano essere osservati da entità quali gli agenti, che possono agire su di esse. In modo analogo, elementi appartenenti al livello digitale possono avere un'estensione fisica (per esempio, tramite AR) che possa essere visibile agli umani. Il mirror world definisce, quindi, un accoppiamento, uno specchio digitale del mondo in cui le persone vivono ed eseguono i loro compiti.

Ponendo le proprie radici sui mirror world, il primo tentativo di definire un modello concreto, nonché un supporto tecnologico di questo accoppiamento tra mondo fisico e digitale è dato dal concetto di *augmented worlds*.

1.2 Il modello di *augmented worlds*

Ispirandosi al modello di mirror world, un *augmented world* definisce un insieme di entità aumentate, che compongono il livello digitale del mondo. Esse posseggono un insieme di proprietà, che possono mutare nel corso del tempo. Queste entità possono essere create e modificate da agenti, i quali percepiscono lo stato e gli eventi associati.

Nella controparte fisica, ogni *augmented entity* può essere associata a un ologramma, ovvero una rappresentazione di AR che può essere percepita dagli utenti. Ogni cambiamento che avviene nell'entità aumentata si riflette, conseguentemente, nell'ologramma associato corrispondente.

Gli utenti all'interno del mondo fisico hanno un ruolo attivo: non solo percepiscono gli ologrammi ma possono anche interagirvi. A tal fine, anche gli utenti hanno una loro rappresentazione all'interno di un augmented world, che tiene traccia delle sue caratteristiche.

Così come accade per gli utenti, l'accoppiamento con il livello digitale può avvenire anche per gli oggetti fisici. Infatti, un'entità aumentata può rappresentarli, definendo uno stato che, se mutato, comporta la modifica corrispondente nell'oggetto.

Il supporto tecnologico al modello è fornito da **MiRAgE**, il primo framework adibito alla creazione ed esecuzione di mondi aumentati. Un componente fondamentale nell'architettura di MiRAgE è *Hologram Engine*, in esecuzione sia lato server che lato client. Nel primo caso, esso ha il compito di fornire supporto per il design delle geometrie di ogni ologramma e mantenere aggiornate le rappresentazioni in base alle proprietà delle rispettive entità aumentate. Nel secondo caso, invece, esso deve fornire tutte le funzionalità necessarie per visualizzare gli ologrammi nel mondo fisico, incapsulando le tecnologie software adibite a questo compito.

1.3 Obiettivo della tesi

Da una prima ricognizione condotta, si evidenzia una certa frammentazione nelle tecnologie per lo sviluppo di sistemi di realtà aumentata. Sebbene sia stata raggiunta una maturità, sia a livello hardware che software, tale da gestire elementi di augmented e mixed reality, essa non viene affiancata da un modello solido, che permetta di essere un supporto per lo sviluppo.

Augmented World si propone come il primo modello per la creazione di sistemi agent-based pervasivi di mixed reality, astraendo dalle tecnologie esistenti. Al contempo, MiRAgE è il primo framework che permette lo sviluppo e l'esecuzione di mondi aumentati.

Poiché la rappresentazione degli ologrammi è un elemento cruciale in MiRAgE, permettendo all'utente di immergersi in un mondo in cui elementi fisici e virtuali coesistono, è opportuno che essa sia affidata ad una tecnologia efficace nella visualizzazione e ancoraggio di elementi virtuali all'interno dell'ambiente.

Da questa analisi si pone l'obiettivo della tesi, ovvero integrare una tecnologia di AR/MR nel framework MiRAgE, che permetta una rappresentazione efficace degli ologrammi nel livello fisico del mondo. Tale tecnologia è stata individuata in ARCore.

1.3.1 Integrazione della piattaforma ARCore in MiRAgE

L'obiettivo della tesi mira a integrare la tecnologia software ARCore all'interno dell'architettura di MiRAgE, estendendo il componente Hologram Engine adibito alla rappresentazione degli ologrammi nel mondo fisico.

Da una prima analisi sulla piattaforma, si evince come ARCore fornisca sia caratteristiche *markerless* sia approcci *marker-based*, garantendo un grado di robustezza elevato nella rappresentazione e ancoraggio di elementi virtuali

nell'ambiente da parte di MiRAgE, migliorando l'attuale gestione affidata al framework Vuforia.

Estendendo il prototipo di MiRAgE, integrando ARCore come piattaforma per la rappresentazione degli ologrammi, si vogliono incapsulare le sue funzionalità e, in aggiunta, affrontare alcune tematiche chiavi inerenti al modello di augmented world, tra le quali il processo su come deve avvenire l'accesso al mondo all'interno di un ambiente fisico, gestire la corretta posizione e rotazione degli ologrammi rispetto alla posizione dell'utente, valutare più punti di accesso al mondo e gestire elementi dinamici.

Inoltre, l'attuale versione di MiRAgE non considera la *user interaction*, ovvero l'interazione che l'utente può avere con gli ologrammi e viceversa. Risulta necessario, a seguito dell'integrazione di ARCore, analizzare anche questo aspetto, in particolare considerando l'utente non singolarmente come individuo ma accoppiato a un'entità aumentata corrispondente, che tiene traccia della sua posizione e rotazione rispetto al mondo. In questo modo anche l'utente può essere percepito dagli ologrammi, i quali possono interagire con esso e viceversa.

L'integrazione della piattaforma, assieme all'estensione delle funzionalità sopra descritte, vuole produrre un framework che nasconda i dettagli tecnici e implementativi, permettendo a uno sviluppatore di focalizzarsi unicamente sul modello delle entità aumentate, definendo le loro proprietà, il loro comportamento e le geometrie dei corrispondenti ologrammi.

1.3.2 Validazione attraverso caso di studio

La validazione sarà una fase importante al termine del raggiungimento dell'obiettivo proposto in questa tesi, in quanto permetterà di verificarne la robustezza, l'efficacia di quanto prodotto ed eventuali limiti, nonché mettere in luce possibili sviluppi futuri.

Dopo un insieme di *unit test*, che permetteranno di verificare che il sistema produca gli output previsti a fronte di determinati input, si procederà a creare esempi ad-hoc che simuleranno le diverse funzionalità sviluppate e analizzeranno i diversi scenari possibili.

Al termine di questa fase, si procederà a validare il nostro framework attraverso un caso di studio. Nello specifico, si intende sfruttare il contesto di `Rocca delle Caminate`[1] per sviluppare un'applicazione che, utilizzando il framework prodotto, permetta a un visitatore di accedere a un mondo aumentato, visualizzando elementi virtuali sia statici sia dinamici dispersi nell'ambiente e interagendo con alcuni di essi.

[1] `http://www.roccadellecaminate.com`

Rocca delle Caminate permetterà di cimentare il framework in un contesto reale, sfruttando gli spazi ampi per verificare la robustezza e precisione della rappresentazione degli ologrammi e del motion tracking in ambienti outdoor.

Lo scenario tipico è un utente che, fornito di un tablet, possa accedere al mondo aumentato, tramite l'applicazione sviluppata. Successivamente, egli è libero di muoversi all'interno degli spazi della rocca e visualizzerà elementi virtuali che coesistono assieme al mondo fisico. Avvicinandosi ad alcuni di essi, potrà interagirvi e visualizzare il risultato di tale interazione.

L'applicazione sviluppata si collocherà nel contesto di valorizzazione del patrimonio culturale, permettendo di estendere ambienti di rilevanza storica con elementi moderni di mixed reality, nel tentativo di aumentare l'interesse e la partecipazione dei visitatori.

Capitolo 2

Augmented/Mixed Reality

Sebbene la realtà aumentata sia diventato un tema ricorrente negli ultimi anni, grazie ai recenti progressi sia a livello hardware che software, essa trova le sue origini già a metà degli anni Novanta. In questo capitolo si vuole condurre una panoramica sull'augmented e mixed reality, mostrandone vantaggi e limiti, nonché le applicazioni nella vita di tutti i giorni. Si propongono, inoltre, le principali tecnologie oggi disponibili.

2.1 Cenni di realtà aumentata

Un primo riferimento sulla realtà aumentata può essere datato intorno al 1950, quando il regista Morton Heilig vedeva il cinema come un'attività che comprendesse tutti i sensi, che proiettasse lo spettatore all'interno dello schermo. Da questa concezione egli sviluppò un prototipo meccanico, denominato *Sensorama*, che immergeva completamente l'utente in un'esperienza avvolgente, caratterizzata da immagini 3D, audio stereofonico, sensazione tattile di movimento, vibrazioni e, persino, vento. Possiamo considerare Sensorama uno dei primi esempi di realtà virtuale.

Nel 1994 Paul Milgram, in *Augmented Reality: A class of displays on the reality-virtuality continuum* [14], nel tentativo di individuare una relazione tra la realtà aumentata e quella virtuale, propose una prima classificazione che confluì in quello che egli stesso definì come *Reality-Virtuality Continuum*. Secondo questo modello, il *Real Environment*, ovvero un ambiente costituito solamente da oggetti reali e il *Virtual Environment*, un ambiente composto da soli componenti virtuali, si collocano rispettivamente agli antipodi sinistro e destro di una scala continua. L'intervallo compreso tra questi due estremi rappresenta la *Mixed Reality*, ovvero spazi in cui coesistono sia componenti fisici che virtuali. Avvicinandosi all'estremo dell'ambiente reale parleremo di *Augmented Reality*, in cui dati digitali e virtuali si sovrappongono al mondo reale mentre

accostandosi all'estremo opposto parleremo di *Virtual Reality*, ovvero ambienti prettamente virtuali all'interno dei quali vengono inseriti alcuni oggetti fisici.

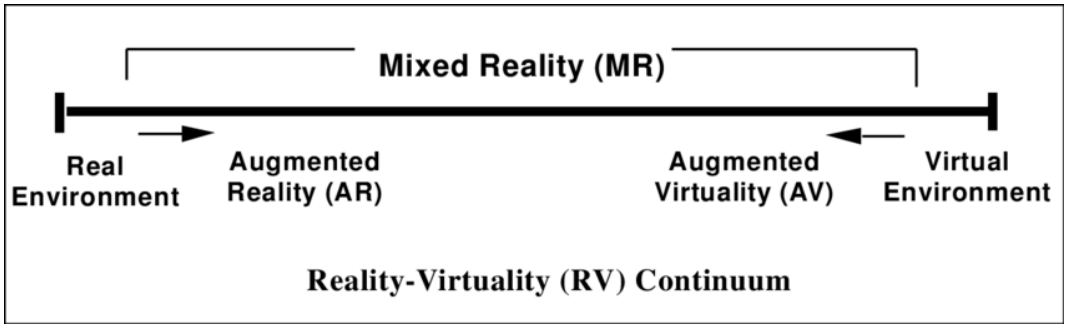

Figura 2.1: Rappresentazione del RV Continuum proposto da Paul Milgram in [14]

Nel 1997 Ronald Azuma, in [17], propose anch'egli una propria analisi, focalizzandosi sulle differenze tra realtà aumentata e virtuale. Infatti, mentre la seconda tende a immergere l'utente in un mondo completamente sintetico, in cui non si hanno riferimenti a spazi fisici reali, la prima permette di osservare il mondo fisico, con oggetti virtuali che si sovrappongono a esso. Al fine di definire la realtà aumentata in modo indipendente dalle singole tecnologie, Ronald Azuma identificò tre caratteristiche fondamentali che ogni sistema di questa tipologia dovrebbe integrare:

- Combinazione di reale e virtuale
- Interazione in real-time
- Registrazione nelle tre dimensioni

Negli ultimi anni vi è stato un notevole progresso nel campo della realtà aumentata, concretizzatosi con il rilascio di device mobili e wearable, nonché di piattaforme per lo sviluppo. Oggi, quando si parla di realtà aumentata, è importante distinguere le tre macro categorie:

- **Virtual Reality:** l'utente è completamente immerso in un ambiente virtuale ed, eventualmente, può muoversi in esso e udire i suoni.
- **Augmented Reality:** l'utente è immerso in un ambiente reale, al quale sono sovrapposti alcuni contenuti digitali.
- **Mixed Reality:** fusione di mondi reali e virtuali, che producono nuovi ambienti in cui gli oggetti fisici e digitali coesistono e l'utente può interagire con essi in real-time.

2.2 Applicazioni della realtà aumentata

Oggigiorno, diversi studi sono stati condotti per dimostrare l'utilità e l'efficacia della realtà aumentata in numerosi ambiti. Al contempo, essa è già stata applicata e utilizzata con successo in alcuni settori.

In questo paragrafo si vogliono descrivere i principali campi nei quali la realtà aumentata può essere un ottimo supporto alle attività delle persone, in particolare in ambito medico, educativo o di intrattenimento, industriale e nelle pubblicità.

2.2.1 Medicina

La realtà aumentata può essere applicata nel campo della medicina come supporto per la chirurgia. Infatti, molti esperti ritengono che l'AR migliorerà la precisione durante le operazioni, riducendo gli errori e fornendo a medici e pazienti una migliore comprensione dei problemi medici complessi. Basti pensare alle tecniche minimamente invasive che riducono la capacità del chirurgo di vedere all'interno del paziente, rendendo l'operazione più difficile. Le tecnologie moderne di realtà aumentata possono facilitare l'operazione, fornendo una vista interna del paziente senza la necessità di incisioni di grandi dimensioni.

Tra i primi tentativi in questo campo vi è sicuramente il lavoro svolto da Fuchs nel 1998 in [7], nel quale si presenta il prototipo di un sistema per la visualizzazione in 3D per assistere le operazioni chirurgiche di laparoscopia. Fuchs, innanzitutto, riconosce in questa operazione tre limiti principali:

- Le immagini fornite al chirurgo sono in 2D, obbligando quest'ultimo solamente a stimare la profondità delle strutture
- Il laparoscopio, ovvero lo strumento endoscopico per l'esame diretto della cavità addominale, ha un ridotto campo visivo, costringendo il chirurgo ad aggiustare frequentemente l'orientamento e la posizione della fotocamera
- La procedura richiede una coordinazione occhio-mano significativa, in quanto solitamente la direzione della fotocamera dello strumento non rispecchia quella nella quale il chirurgo sta guardando

Al fine di migliorare queste problematiche, il chirurgo viene dotato di un *head-mounted display*, ovvero un display posizionato sulla testa attraverso un casco ad hoc, il quale riceve un'immagine costituita sia da elementi reali, che derivano dalla piattaforma di *image-generation* e dal sistema di tracking sia

virtuali, che derivano dal laparoscopio 3D, il quale acquisisce informazioni sulla profondità delle strutture al fine di mostrare un'immagine tridimensionale.

Oltre a essere un ottimo strumento a supporto di un chirurgo, la realtà aumentata può essere applicata, inoltre, per scopi di apprendimento, mostrando istruzioni virtuali per un chirurgo novizio oppure oggetti che identificano organi e specificano la loro posizione.

2.2.2 Intrattenimento ed educazione

L'utilizzo della realtà aumentata può essere esteso ad ambiti culturali, quali guide turistiche o musei, ad ambiti di intrattenimento, come le applicazioni di *gaming* e, infine, per scopi educativi, ad esempio in ambito scolastico.

L'utilizzo di tecnologie di augmented o mixed reality all'interno di un museo permettono di aggiungere informazioni digitali a oggetti fisici in ivi presenti e, al tempo stesso, di mantenere alto l'interesse e il coinvolgimento del visitatore, creando ambienti nei quali l'utente possa interagire con elementi virtuali. Immaginiamo, per esempio, un turista in visita a una galleria d'arte. Grazie a uno specifico dispositivo, quale tablet od occhiali smart, lo spettatore potrebbe inquadrare ogni quadro della mostra e ottenere informazioni specifiche per ognuno di esso, nonché contenuti multimediali aggiuntivi con cui interagire.

Uno studio, condotto da Jung T., tom Dieck M. Claudia, Lee H. e Chung N. e confluito in [11], indaga l'impatto della realtà aumentata e virtuale sull'esperienza complessiva di un visitatore nel contesto di un museo. Effettuando alcune misurazioni su determinate esperienze, quali educativa (l'utente ha imparato qualcosa durante l'utilizzo di tecnologie di augmented e virtual reality?), estetica (l'utente ha trovato interesse nelle attività di augmented e virtual reality?) e di intrattenimento (l'utente si è divertito durante le attività di AR e VR proposte?), lo studio ha dimostrato che queste hanno avuto un impatto significativo sull'esperienza del visitatore, il quale è motivato a visitare nuovamente l'attrazione. Lo studio, infine, conclude affermando che, al fine di arricchire l'esperienza dell'utente, occorre creare ambienti di mixed reality in cui egli possa totalmente immergersi e, inoltre, è necessario considerare fattori accattivanti come animazioni 3D, che permettono di migliorare l'esperienza di intrattenimento del visitatore.

In ambito gaming, la realtà aumentata ha destato interesse nelle case produttrici di console e tra gli sviluppatori di applicazioni mobile. Un noto esempio è dato dal *Playstation VR*, visore sviluppato da Sony, che permette a tutti i giocatori di titoli compatibili per Playstation di immergersi in ambienti completamente virtuali e interagire con essi. Grazie all'evolversi delle tecnologie e al rilascio di opportuni SDK, anche gli sviluppatori di videogiochi per dispositivi mobili hanno ora la possibilità creare ambienti di mixed reality, con il quale

il giocatore può interagire semplicemente attraverso il proprio smartphone o tablet.

La realtà aumentata può essere di ausilio anche nel campo dell'istruzione. Immaginiamo, per esempio, un libro che permette, tramite opportune tecnologie di AR, di mostrare modelli 3D con i quali è possibile interagire. Nelle aule di scuola, in concomitanza con un approccio di insegnamento tradizionale, è possibile integrare una metodologia innovativa, in cui il materiale didattico è supportato da una controparte di esempi o concetti di realtà aumentata. In [12] si evidenziano cinque esempi di utilizzo:

- **Astronomia:** la relazione tra la Terra e il Sole può essere studiata in maniera interattiva, in cui il pianeta e la sua stella principale possono essere rappresentati in 3D e gli studenti possono modificare la prospettiva in modo tale da avere un maggior livello di dettaglio.

- **Chimica:** la struttura di un atomo o di una molecola può essere mostrata in modo interattivo tramite il rendering 3D di opportuni modelli.

- **Biologia:** la realtà aumentata può essere utilizzata per studiare l'anatomia e la struttura del corpo umano. Gli insegnanti possono usare tecnologie di AR per mostrare di quali organi si compone il nostro corpo e come appaiono attraverso modelli 3D generati da computer nelle aule di scuola.

- **Matematica e geometria:** tramite applicazioni di AR, insegnanti e studenti possono esplorare alcune proprietà matematiche di curve, superfici e altre forme geometriche.

- **Fisica:** la realtà aumentata può essere utilizzata nel campo della fisica per dimostrare diverse proprietà cinematiche.

2.2.3 Industria

In ambito industriale, le fasi di produzione e di riparazione sono i punti di interesse più promettenti, dove la realtà aumentata può essere utilizzata per migliorare le tecniche correnti e fornire nuove soluzioni. Con i recenti progressi nelle tecnologie informatiche e di produzione, vi è una tendenza crescente a consentire agli utenti di interagire direttamente con le informazioni associate ai processi di produzione. In quest'ottica l'AR ha la capacità di integrare queste modalità in tempo reale nell'ambiente di lavoro, fornendo all'utente un modo intuitivo per interagire direttamente con le informazioni. Nel tentativo di definire le diverse applicazioni della realtà aumentata nelle fasi di design e

di produzione industriale, l'articolo [15] evidenzia alcuni punti interessanti in cui l'AR può essere applicata:

- **Prototipazione:** l'insieme dei processi volti alla realizzazione fisica del prodotto può essere coadiuvato da tecnologie di AR. Un esempio è dato dall'industria automobilistica, dove la realtà aumentata è stata utilizzata per valutare l'interior design, sovrapponendo diversi interni di auto, che solitamente sono disponibili come modelli 3D nelle fasi iniziali dello sviluppo, su modelli di carrozzeria reali.

- **Annotazioni:** al fine di facilitare il processo di decision-making, alcune informazioni sono associate a prodotti come label virtuali e mostrate ai designer.

- **Manutenzione:** le tecnologie di AR mostrano benefici nelle applicazioni di manutenzione in due aspetti. In primo luogo, le interfacce utente possono essere rese in modo tale che il lavoratore percepisca le istruzioni con meno sforzo. In secondo luogo, le interazioni dell'utente nell'ambiente di AR possono facilitare la gestione dei dati di manutenzione e consentire la collaborazione remota in modo intuitivo.

- **Assemblaggio:** dato un insieme di componenti, è necessario ottenere una sequenza di operazioni di assemblaggio ottimale per ridurre il tempo e lo sforzo di completamento dell'assemblaggio. A questo scopo l'AR può essere sfruttata per creare oggetti virtuali che, combinati insieme ad ambienti reali, possono migliorare il design dell'assemblaggio e il processo di pianificazione, permettendo di analizzare il comportamento e le proprietà dei prodotti assemblati.

2.2.4 Pubblicità

Lo scopo principale della pubblicità è introdurre le persone a nuovi prodotti. Tuttavia, al fine di ottenere potenziali consumatori interessati a un prodotto, lo spettatore deve innanzitutto riconoscere che il bene esiste e ricordare i dettagli pertinenti su di esso. Secondo queste premesse, l'AR può fornire un ottimo supporto, permettendo a un possibile acquirente di controllare un'immagine virtuale 3D, raffigurante il prodotto che si intende pubblicizzare.

Un approccio tipico è fornire all'utente un marker, ovvero un'immagine 2D con un pattern specifico che, una volta inquadrato con una fotocamera, mostra un contenuto digitale sovrapposto a detto marker e con il quale l'utente può interagire.

Immaginiamo, per esempio, una casa automobilistica che voglia pubblicizzare le proprie autovetture. Essa potrebbe distribuire un'applicazione ad-hoc

che, utilizzando la fotocamera e inquadrando con quest'ultima un pattern di riferimento, mostri su di esso la rappresentazione 3D dell'automobile. L'utente, successivamente, può interagire con tale vettura, visualizzandone i dettagli e modificandone la configurazione.

2.3 Dispositivi per la realtà aumentata

Nel corso degli ultimi anni, la potenza di calcolo e dispositivi sempre più performanti hanno raggiunto una maturità tale da poter gestire elementi di augmented o mixed reality. Al fine di raggruppare la grande eterogeneità dei dispositivi abilitanti all'AR/MR, in [2] viene proposta la seguente suddivisione: display, device di input, dispositivi di tracking e computer.

2.3.1 Display

In questa categoria si inseriscono i principali display utilizzati nell'ambito della realtà aumentata, ovvero: head mounted displays (HDM), handheld displays e spatial displays.

Head mounted displays

Si tratta di display di device che vengono indossati sulla testa dell'utente (o che sono parte di un casco) e che posizionano sia immagini del mondo virtuale sia di quello reale sul campo visivo dell'utente. Gli HDM si suddividono, a loro volta, in due gruppi:

- **Video-see-through:** richiedono all'utente di indossare due camere e, al fine di ottenere una scena con elementi sia reali che virtuali, è necessario attendere che entrambe abbiano completato le operazioni. Essendo, in questa tipologia di sistemi, la scena già elaborata, si ha un maggiore controllo sul risultato.
- **Optical-see-through:** utilizzano una tecnologia a specchio semitrasparente per consentire alla vista del mondo fisico di passare attraverso l'obiettivo e sovrapporre graficamente le informazioni che si riflettono negli occhi dell'utente. La scena, così come il mondo fisico, è percepito in modo più naturale rispetto alla risoluzione del display.

Handheld displays

Gli handheld display rappresentano tutti quei display di dispositivi di dimensioni ridotte che possono essere tenuti in mano dall'utente. Utilizzano

tecniche di *video-see-through* per sovrapporre oggetti digitali nell'ambiente fisico e impiegano sensori per il tracciamento e sistemi per il riconoscimento di marker o tecniche di computer vision. Possiamo suddividerli in tre classi principali:

- **Smartphone:** particolarmente portatili e diffusi, i recenti modelli includono una CPU performante, fotocamera, accelerometro, GPS e altri sensori che lo rendono una piattaforma hardware adatta per applicazioni di AR.
- **PDA:** presentano all'incirca gli stessi vantaggi e svantaggi degli smartphone ma sono di gran lunga meno diffusi.
- **Tablet:** tipicamente hanno una potenza maggiore rispetto agli smartphone ma sono più costosi e necessitano un utilizzo a due mani a causa delle loro dimensioni.

Spatial displays

La *Spatial Augmented Reality* (SAR) utilizza videoproiettori, elementi ottici, ologrammi e altre tecnologie di tracking per visualizzare informazioni grafiche direttamente negli oggetti fisici, senza richiedere all'utente di indossare o avere con sé un display. A tal fine, gli spatial displays separano gran parte della tecnologia dall'utente per integrarla nell'ambiente.

Gli spatial displays si suddividono in tre categorie:

- **Video-see-through:** tipicamente screen-based, sono utilizzati se il sistema non deve essere mobile in quanto richiedono solo componenti hardware e apparecchiature PC standard.
- **Optical-see-through:** generano immagini che sono allineate all'interno dell'ambiente fisico. Schermi trasparenti e ologrammi sono componenti tipici di questi display. Tuttavia, non supportano applicazioni mobile a causa della tecnologia di visualizzazione.
- **Projector-based:** utilizza una proiezione frontale per rappresentare le immagini direttamente sulle superfici degli oggetti fisici.

2.3.2 Device di input

Questa categoria comprende numerosi dispositivi, che dipendono principalmente dal sistema che si intende sviluppare. Infatti, se un'applicazione richiede che l'utente debba utilizzare le mani liberamente, è necessario un device

di input che permetta questa libertà di movimento, senza obbligare l'utente a utilizzare gesture non naturali. Un esempio è dato dai braccialetti smart, che possono essere indossati come semplici braccialetti e includono alcuni sensori di interesse, come il Wi-Fi.

Viceversa, se un sistema fa utilizzo di un handheld display, allora lo stesso schermo touch dello smartphone può essere utilizzato come dispositivo di input.

2.3.3 Dispositivi di tracking

I dispositivi di tracking comprendono fotocamere o sensori ottici, GPS, accelerometri, sensori wireless e tutti quei device che permettono di tracciare la posizione in relazione a un ambiente. Le tecnologie elencate tipicamente hanno un grado di accuratezza che dipende in gran parte dal sistema nel quale è stata integrata. Nel campo della realtà aumentata si evidenziano:

- **Fotocamera:** strumento ottico utilizzato per apprendere l'ambiente circostante.
- **Accelerometro:** misura l'accelerazione subita da un device, identificandone l'orientamento
- **Giroscopio:** rileva i movimenti del device lungo gli assi X, Y e Z e lavora in sintonia con l'accelerometro per avere misurazioni più precise su tali movimenti.
- **Magnetometro:** permette di misurare l'intensità e la direzione dei campi magnetici.
- **GPS:** consente di conoscere le coordinate del device in termini di latitudine, longitudine e altitudine.

2.3.4 Computer

Comprendono tutti quei computer con un quantitativo di CPU e di RAM tali da processare le immagini della fotocamera.

Poiché oggi i dispositivi mobili, quali smartphone e tablet, hanno raggiunto una potenza di calcolo tale da permetter loro di processare immagini in tempo reale e di essere utilizzati direttamente in contesti di realtà aumentata, è opportuno integrare la suddivisione proposta da [2], aggiungendo in questa categoria tutti i device mobili compatibili.

2.4 Registrazione nei sistemi di realtà aumentata

Una tematica centrale quando si tratta di sistemi di realtà aumentata è la registrazione. Infatti, affinché sia solida l'illusione che il mondo reale e quello virtuale coesistano, è necessario che gli oggetti all'interno di essi siano allineati correttamente. Basti pensare a un'operazione di chirurgia assistita dalla realtà aumentata: è indispensabile che la registrazione sia accurata affinché si possa procedere senza complicazioni.

In questo paragrafo si vogliono descrivere le principali tecniche che permettono di sincronizzarsi con il mondo fisico e definire le cause che provocano un fallimento nel processo di registrazione.

2.4.1 Tecniche di registrazione e motion tracking

La precisione nel processo di registrazione è un elemento chiave nei sistemi di realtà aumentata. Le moderne applicazioni di AR cercano di identificare la posizione e la rotazione della fotocamera tramite diversi sensori, quali accelerometro, giroscopio, magnetometro, GPS e Wi-Fi al fine di avere una precisione accurata.

Il risultato della combinazione di queste informazioni è la *pose*, nota anche come 6DOF (Six-degrees of freedom). Essa definisce la posizione e l'orientamento della fotocamera nello spazio tridimensionale: avanti/indietro, su/giù, destra/sinistra combinati con la rotazione attorno ai tre assi perpendicolari.

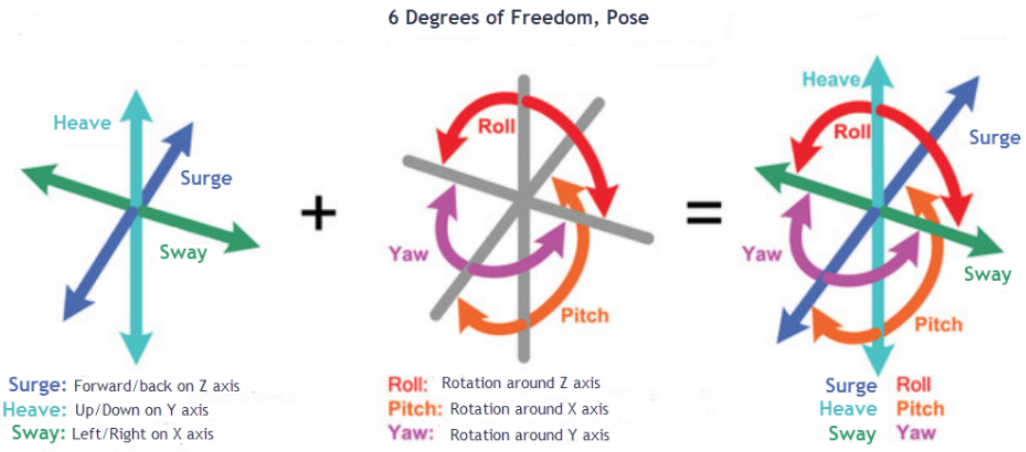

Figura 2.2: Rappresentazione grafica della *pose* proposta in [13]

Seguendo la suddivisione proposta in [10], esistono due tecniche principali per sincronizzarsi con il mondo fisico: marker-based e markerless.

Tecniche marker-based

Uno degli approcci più comuni e semplici per determinare la pose della fotocamera è basato sulla visione di marker noti, ovvero immagini tipicamente quadrate e con un pattern specifico.

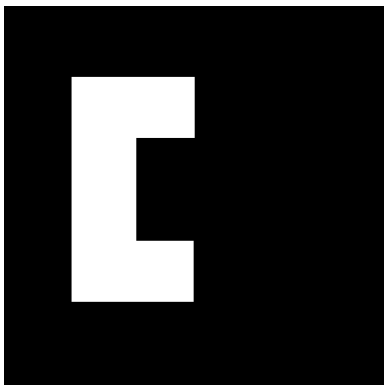

Figura 2.3: Esempio di marker

La posizione del marker, assieme alla calibrazione della fotocamera, viene utilizzata per sovrapporre con precisione elementi tridimensionali sul display. Grazie alla loro forma e pattern predefiniti, i marker sono facilmente rilevabili nell'ambiente e permettono il calcolo della pose in modo rapido.

I marker possono essere utilizzati per sovrapporre in esso un oggetto virtuale oppure per definire un punto di origine dal quale posizionare i diversi elementi.

La principale limitazione si evidenzia negli ambienti di grandi dimensioni, dove oggetti posizionati distanti dal marker possono perdere precisione nella locazione.

Tecniche markerless

A differenza degli approcci marker-based, le tecniche markerless non necessitano di riconoscere immagini predefinite all'interno dell'ambiente ma identificano in automatico delle *features*, che possono essere d'interesse per il tracking. Tra le tecniche più note risalta SLAM.

SLAM (Simultaneous localization and mapping) cerca di generare una mappa in un ambiente sconosciuto e localizza il device all'interno di essa, attraverso

una serie di computazioni e algoritmi complessi che sfruttano i dati ricevuti dai sensori IMU.

Esistono diversi algoritmi per stabilire la pose utilizzando SLAM. La tecnica principale esegue un'ottimizzazione computazionale non lineare e intensiva, chiamata *bundle adjustment*, per assicurarsi che vengano creati modelli con un livello alto di accuratezza. Questa ottimizzazione è migliorata in modo significativo utilizzando parallelamente più processori e GPU.

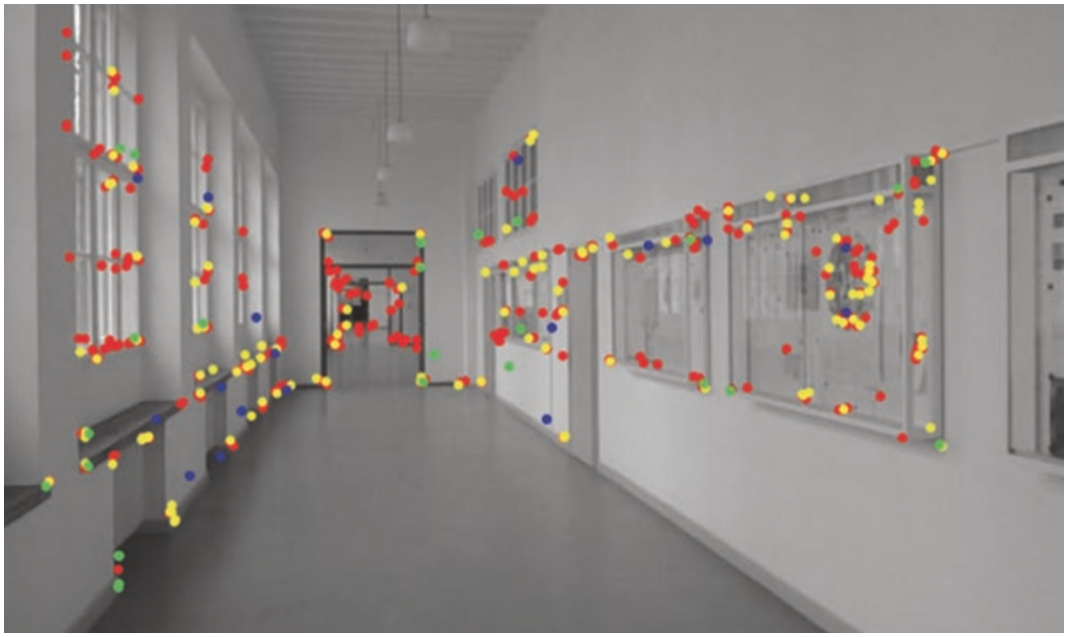

Figura 2.4: Identificazione di features tramite tecnica SLAM, tratta da [10]

2.4.2 Errori di registrazione

A causa dei requisiti di accuratezza richiesti, come il tempo di risposta del sistema al cambiare del punto di vista dell'utente, durante la registrazione possono insorgere diversi errori, che in [17] vengono raggruppati in due classi principali: errori statici e dinamici.

Errori statici

Gli errori statici comprendono tutti quegli errori che avvengono nella registrazione, nonostante il punto di vista dell'utente e gli oggetti, immersi nell'ambiente, rimangano fermi.

Le cause principali sono da attribuire a:

- **Distorsione ottica:** l'immagine tende a diventare di gran lunga più distorta all'aumentare della distanza dal centro del campo visivo. Tendono a soffrire di questo errore principalmente i display con campo visivo ampio, in quanto le distorsioni sono in funzione della distanza radiale dall'asse ottico.

- **Errori nel sistema di tracking:** i *tracking system* permettono di determinare e seguire la posizione e l'orientamento di un oggetto nel tempo rispetto a un sistema di riferimento. Spesso questi sistemi non sono precisi a sufficienza da poter soddisfare i requisiti di sistemi di realtà aumentata, causando errori non sistematici nei valori di output.

- **Disallineamenti meccanici:** si tratta di discrepanze tra le specifiche hardware e le proprietà fisiche effettive del sistema, che possono causare cambiamenti nella posizione e nell'orientamento degli elementi virtuali.

- **Parametri di visualizzazione non corretti:** i parametri di visualizzazione specificano come convertire le posizioni della fotocamera in matrici di visualizzazione, in modo tale che un generatore di scene possa disegnare correttamente immagini digitali. Valori non corretti nei parametri di visualizzazione sono la causa principale di errori sistematici di allineamento.

Errori dinamici

Gli errori dinamici comprendono tutti quegli errori che avvengono a causa di ritardi nel sistema, meglio noti come *lag*. Questi ritardi sono causati dal fatto che ogni componente in un sistema di realtà aumentata richiede un certo tempo per completare il suo compito. Infatti, ritardi nel sistema di tracciamento, ritardi nella comunicazione e il tempo necessario al generatore di scene per disegnare immagini virtuali contribuiscono insieme al verificarsi di lag.

Per evitare fallimenti nella registrazione, dovuti a questa tipologia di errori, si possono applicare i seguenti metodi:

- **Riduzione del lag di sistema:** si basa sull'idea che, eliminando completamente i ritardi, si annullino gli errori dinamici di conseguenza. La riduzione dei ritardi del sistema al punto che questi non siano più un problema risulta, però, molto difficile. Per esempio, i generatori di scene sono progettati per massimizzare il throughput, a discapito della latenza. Una possibile soluzione è la riconfigurazione del software, in modo tale da minimizzare la latenza. Una seconda possibilità è una sincronizzazione accurata dei task nella pipeline, che permette così una riduzione del lag.

- **Riduzione del lag apparente:** qualora si tratti di sistemi che utilizzano solamente l'orientamento della testa, è possibile applicare tecniche di *image deflection* per ridurre il lag apparente. Si tratta di metodi che permettono di incorporare le più recenti misurazioni di orientamento all'interno della pipeline di rendering. Viceversa, è possibile impiegare tecniche di *feed-forward*, per esempio facendo il render di un'immagine più grande del necessario per riempire il display.

- **Combinazione di flussi temporali:** poiché il generatore di scene produce del ritardo nel generare le immagini virtuali, è possibile imporre un ritardo anche al flusso video, originato dalle videocamere, in modo tale da sincronizzare i flussi temporali. Nonostante si riduca il conflitto tra mondo reale e virtuale, sia gli oggetti reali sia quelli virtuali vengono sottoposti a ritardo. Di conseguenza, questa tecnica è applicabile con successo solamente con ritardi di lieve entità mentre produce un'esperienza negativa con ritardi di lunga durata.

- **Previsione di posizioni future:** tecnica che si basa sulla previsione futura delle posizioni degli oggetti. Infatti, conoscendo tali informazioni, è possibile fare il render della scena con le posizioni previste anziché quelle misurate. Tuttavia, questo metodo richiede un sistema per le misurazioni e computazioni real-time accurato e preciso.

2.5 Tecnologie software di realtà aumentata

Negli ultimi anni, gli sviluppatori hanno avuto accesso a diversi tools che hanno permesso loro di approfondire i concetti di augmented e mixed reality. In particolare, sono stati resi disponibili piattaforme e SDK per diversi dispositivi, tra i quali smartphone e tablet, rendendo la platea di fruitori più ampia.

In questa sezione si intende descrivere una selezione di tecnologie software, che si ritengono rilevanti anche in vista del lavoro da svolgere per raggiungere l'obiettivo della tesi.

2.5.1 Piattaforme di AR/MR

Tra le diverse piattaforme, che permettono lo sviluppo di applicazioni di augmented e mixed reality, si procede a descriverne tre principali: ARCore, ARKit e Vuforia. Le prime due hanno introdotto caratteristiche avanzate di motion tracking e comprensione dell'ambiente, permettendo a dispositivi mobili di eseguire applicazioni di mixed reality utilizzando sia approcci marker-based sia markerless. Il terzo SDK, meno recente dei precedenti, è attualmente

utilizzato nella rappresentazione degli ologrammi in MiRAgE, framework che rappresenta il punto di partenza sui cui si baserà il lavoro della seguente tesi.

ARCore

Sviluppata da Google, ARCore[1] è una piattaforma per lo sviluppo di applicazioni di realtà aumentata. Consente ai device compatibili di rilevare l'ambiente circostante, comprendere il mondo reale e interagire con le informazioni a esso associate.

Attraverso la fotocamera, il contenuto virtuale viene integrato nel mondo reale tramite la combinazione di tre tecniche principali: motion tracking, environmental understanding e light estimation. La prima permette di tenere traccia della posizione dello smartphone rispetto al mondo; la seconda permette di identificare la dimensione e la posizione di tutte le tipologie di superfici (orizzontali, verticali e ad angolo); l'ultima tecnica, infine, permette di stimare le condizioni di luminosità dell'ambiente.

La fotocamera ha un ruolo chiave in quanto viene utilizzata da ARCore per identificare e tenere traccia di punti di interesse, detti *features*. Grazie ai movimenti di questi punti, combinati con le letture dei sensori del dispositivo, ARCore riesce a determinare sia la posizione che l'orientamento del device mentre si muove nell'ambiente che lo circonda.

Grazie alle librerie fornite, lo sviluppatore è in grado di identificare piani e pattern di immagini 2D nell'ambiente, interagire con essi e inserire nuovi elementi nella scena, la cui posizione sarà tracciata e aggiornata da ARCore. Inoltre, grazie agli ultimi aggiornamenti, ARCore è in grado di utilizzare la fotocamera frontale e riconoscere i volti.

ARKit

Sviluppato da Apple, ARKit[2] è, anch'esso, una piattaforma per creare esperienze di realtà aumentata. Così come ARCore, ARKit combina tecniche di motion tracking, comprensione della scena e stima della luminosità per semplificare lo sviluppo di applicazioni. Introdotto assieme a iOS11, è compatibile con tutti i dispositivi iPhone che abbiano un processore della famiglia A9 o superiore.

Per creare una corrispondenza tra mondo reale e virtuale, ARKit utilizza una tecnica chiamata *visual-intertial-odometry*. Questo processo combina le informazioni dai sensori di movimento del dispositivo iOS con analisi di

[1] `https://developers.google.com/ar/`
[2] `https://developer.apple.com/arkit/`

computer vision della scena visibile alla fotocamera del dispositivo. ARKit riconosce alcune caratteristiche chiave (features) nell'immagine della scena, tiene traccia delle differenze nelle posizioni di tali caratteristiche tra i fotogrammi video e confronta tali informazioni con i dati di rilevamento del movimento. Il risultato è un modello preciso della posizione e del movimento del dispositivo.

Oltre a identificare piani e pattern 2D di immagini, ARKit integra tecniche di *object detection*, al fine di riconoscere oggetti 3D noti e può utilizzare la fotocamera frontale del dispositivo per fornire informazioni in tempo reale sull'espressione del volto dell'utente.

Vuforia

Acquisito da PTC Inc. nel 2015, Vuforia[3] è un SDK per lo sviluppo nell'ambito della realtà aumentata, con supporto per smartphone, tablet e occhiali smart. Il core di Vuforia è *Vuforia Engine Library*, che si compone di tre elementi principali che è in grado di gestire:

- **Images:** rappresentano l'insieme delle immagini che Vuforia è in grado di riconoscere. Si suddividono a loro volta in *image targets*, che rappresentano immagini flat 2D; *multi targets*, che definiscono forme geometriche regolari o disposizioni arbitrarie di superfici planari composte da più image targets; infine, i *cylinder targets*, che definiscono immagini avvolte su oggetti di forma approssimativamente cilindrica.

- **Objects:** Vuforia è in grado di riconoscere oggetti tramite *VuMarks*, ovvero marker personalizzati che possono codificare un range di formati di dati oppure tramite *model targets*, che permettono di riconoscere e tracciare oggetti particolari nel mondo reale, in base alla forma dell'oggetto.

- **Environments:** Vuforia analizza l'ambiente tramite *extended tracking*, al fine di mantenere disponibili le informazioni sulla pose di un target anche quando questo non è nel campo visivo della fotocamera e tramite *ground plane*, che abilita l'inserimento di contenuti digitali su superfici orizzontali.

[3] `https://library.vuforia.com/getting-started/overview.html`

Capitolo 3

I mondi aumentati come modello per ambienti di Mixed Reality

Dopo aver fornito un background sulle caratteristiche dell'augmented e mixed reality, descrivendo gli ambiti di utilizzo e le principali tecnologie hardware e software, si procede ora a definire il modello di *augmented worlds* per lo sviluppo e l'esecuzione di sistemi agent-based pervasivi di mixed reality.

3.1 Definizione e struttura di un mondo aumentato

Con il termine *augmented world* si definisce un modello per lo sviluppo di sistemi agent-based pervasivi di mixed reality.

Esso specifica un insieme di oggetti computazionali, localizzati in un certo spazio, che prendono il nome di *augmented entities* (AEs). Ogni AE fornisce un'interfaccia, contenente un insieme di azioni che gli agenti possono eseguire, modificando lo stato delle entità. Inoltre, ognuna di esse propone un set di proprietà osservabili, che rappresentano informazioni sullo stato dell'agente e sono percepibili dagli agenti.

Nel mondo fisico, ogni entità aumentata può avere un ologramma associato, al fine di essere percepibile dagli utenti come una figura bidimensionale o tridimensionale, definita rispetto a un sistema di riferimento locale. Allo stesso tempo, un oggetto fisico può essere accoppiato con un'entità aumentata corrispondente, così che cambiamenti che avvengono allo stato dell'oggetto possono essere notificati al livello aumentato e viceversa.

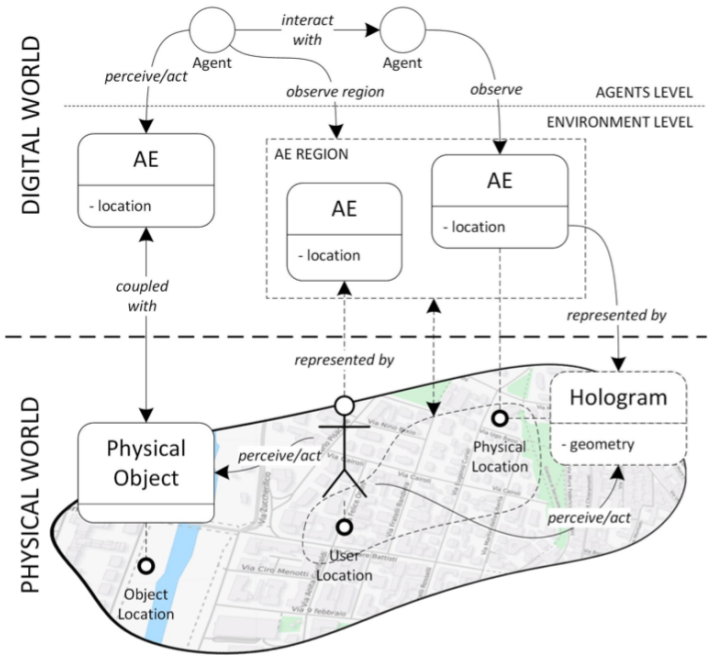

Figura 3.1: Rappresentazione di un Augmented World, tratto da [4]

Un augmented world è tipicamente multi-utente, ognuno dei quali può percepire e interagire con le stesse entità. In un mondo aumentato, un utente può essere modellato associandogli un'entità aumentata, con caratteristiche specifiche che permettono di controllare lo user input/output e con uno stato computazionale, che rappresenta le informazioni dinamiche dell'utente.

Riassumendo, si possono identificare le seguenti caratteristiche chiave di un augmented world:

- **Shared/multi-user worlds:** più utenti sono immersi nello stesso mondo aumentato, condividendo e interagendo con le stesse entità virtuali
- **Bidirectional augmentation:** viene integrata la prospettiva di *pervasive computing*, dove l'ambiente fisico viene arricchito con capacità computazionali
- **Open systems:** gli agenti possono accedere in modo dinamico a un mondo aumentato e possono cambiarne la struttura a tempo di esecuzione
- **Heterogeneous agent (programming) technology:** è permesso l'utilizzo di tecnologie ad agenti differenti ed eterogenei per la programmazione degli agenti all'interno del mondo aumentato

- **Cognitive agency:** nonostante il supporto alle diverse tecnologie ad agenti, il framework risulta particolarmente efficace con agenti con abilità cognitive, ad esempio basati sul modello BDI (Belief - Desire - Intention)

- **Enabling AR technologies:** il design del framework permette il pieno utilizzo delle tecnologie abilitanti all'AR esistenti, mantenendo una separazione netta tra questo livello e quello ad agenti

- **Generality:** il framework mira ad essere sufficientemente generale per supportare lo sviluppo di un augmented world in domini applicativi differenti, includendo sia scenari indoor che outdoor

3.1.1 Il modello concettuale

I concetti principali di un augmented world sono formalizzati nel seguente diagramma. Segue una descrizione delle entità e caratteristiche più rilevanti.

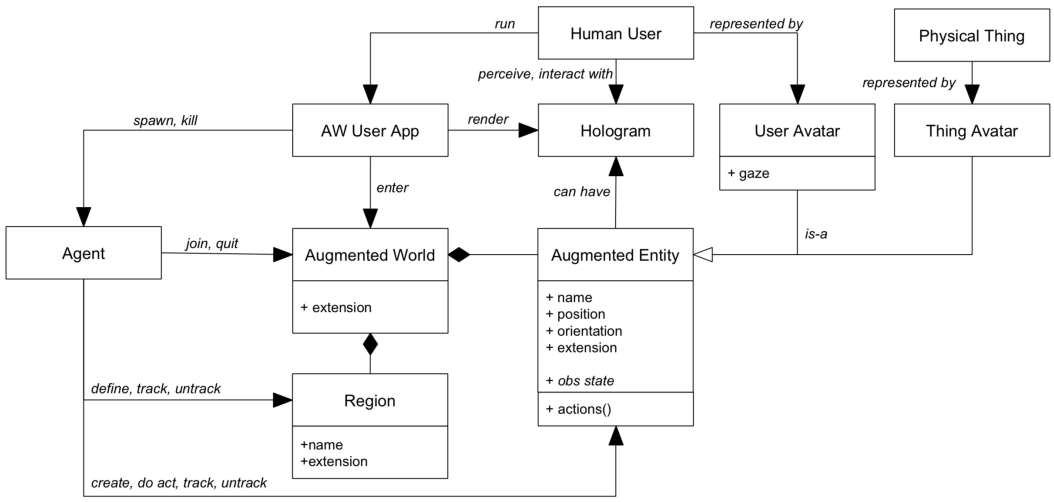

Figura 3.2: Modello concettuale di un Augmented World, tratto da [6]

Augmented Entities

Rappresentano oggetti virtuali, stratificati sopra al livello del mondo fisico. Esse hanno una specifica posizione, orientamento ed estensione rispetto al sistema di riferimento definito dal mondo aumentato. Forniscono uno stato osservabile, ovvero un insieme di proprietà che possono cambiare nel tempo e un'interfaccia, ovvero un insieme di operazioni che possono essere eseguite dagli agenti che le controllano.

Agents

Rappresentano entità che incapsulano uno stato, un comportamento e un flusso di controllo. Per entrare in un mondo aumentato devono necessariamente effettuare la *join*. Successivamente, possono agire sulle entità aumentate attraverso le azioni che esse stesse rendono disponibili tramite un'interfaccia. Inoltre, possono percepire lo stato dell'entità ed eventi. Oltre a tracciare specifiche entità aumentate, gli agenti possono tener traccia di *regions*, ovvero porzioni all'interno dello spazio fisico nel quale il mondo aumentato è mappato.

Holograms

Un'entità aumentata può essere associata a un ologramma, che definisce una rappresentazione di AR di tale entità. Questa rappresentazione è posizionata nel mondo fisico ed è percepibile dagli utenti. Lo stato dell'ologramma rispecchia lo stato dell'entità aumentata: ogni volta che quest'ultimo viene modificato, i cambiamenti si propagano anche all'ologramma corrispondente.

Users

Un utente può essere partecipe del mondo aumentato tramite una specifica applicazione (AW User App). Quest'ultima deve essere in grado di riprodurre un ambiente di realtà aumentata in base alla posizione e all'orientamento dell'utente, nonché di riconoscere i comandi di input.

Physical World Coupling

Le entità aumentate possono essere utilizzate anche per rappresentare oggetti fisici, ovvero oggetti che fanno parte dell'ambiente fisico. Lo stato osservabile della augmented entity rappresenta, quindi, un modello dello stato dell'oggetto fisico.

3.2 Alla base dei mondi aumentati: Mirror World

Il modello dei mondi aumentati può essere considerato un'estensione dell'idea dei Mirror Worlds, proposta in [16].

Nella visione di mirror world gli *smart spaces*, ovvero luoghi in cui i livelli fisici, digitali e sociali sono fortemente intrecciati, sono definiti in termini di città digitali modellate dal mondo fisico con cui sono accoppiate, abitate da

società aperte e organizzazioni di agenti software, che svolgono il ruolo di abitanti.

Il *mirroring* avviene quando oggetti fisici, che possono essere percepiti dagli umani e con i quali è possibile interagire nel mondo fisico, hanno una controparte digitale nel mirror world, in modo tale che essi possano essere osservati da agenti software, che hanno un'estensione nel mondo fisico (per esempio, tramite AR) e possono, di conseguenza, essere osservati dagli umani.

Abbiamo, quindi, una forma di accoppiamento, in cui un'azione su un oggetto del mondo fisico può causare alcuni cambiamenti nelle entità del mirror world, percepibili da agenti software; allo stesso tempo, un'azione eseguita da un agente su un'entità del mirror world può riflettersi negli oggetti del mondo fisico, percepibili dagli utenti.

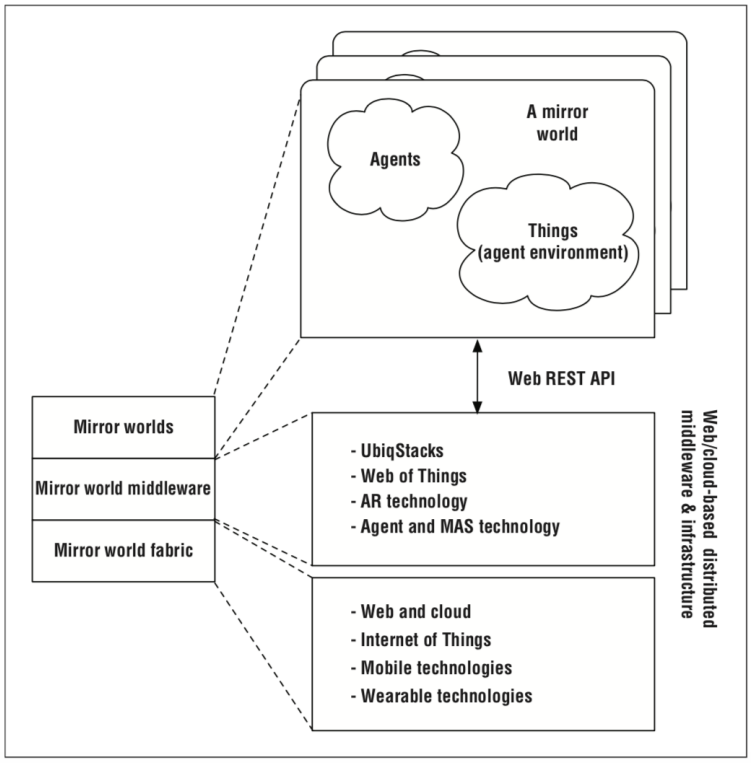

Figura 3.3: Stack di tecnologie abilitanti ai mirror worlds, proposto in [16]

Lo stack di tecnologie, che abilitano ai mirror worlds, comprendono tre livelli principali:

- **Mirror world app layer:** rappresenta il mirror world come applicazioni individuali

- **Mirror world middleware:** infrastruttura che fornisce le API e gestisce l'esecuzione di mirror worlds
- **Mirror world fabric layer:** infrastruttura distribuita low-level, che include la rete e qualsiasi risorsa hardware e software necessaria a eseguire l'infrastruttura mirror world

3.3 Agenti e MAS in sistemi di Mixed Reality

Come abbiamo visto, il modello di augmented worlds si propone di costruire sistemi AR/MR in cui gli agenti hanno un ruolo chiave. Essi, infatti, possono creare e disporre le entità aumentate, nonché tener traccia del loro stato osservabile e degli eventi a esse associati. Possono, inoltre, definire regioni all'interno del mondo aumentato e percepirne gli eventi, quali entrata e uscita di un'entità aumentata.

Solitamente, quando si introduce il concetto di agente si differenziano due definizioni principali:

- **Weak definition:** l'agente è un'entità computazionale caratterizzata da `autonomia` (non è controllata esternamente e ha il controllo sulle proprie azioni e il suo stato interno), `abilità sociale` (può interagire con altri agenti così come con gli utenti), `reattività` (può percepire l'ambiente e reagire a cambiamenti che avvengono al suo interno) e `proattività` (può prendere iniziativa, mostrando un comportamento *goal-directed*)
- **Strong definition:** l'agente è un'entità composta da particolari abilità mentali o cognitive. Questa definizione tende verso un modello BDI, che definisce l'agente in termini di `beliefs` (informazioni riguardo il mondo), `desires` (obiettivi a lungo termine) e `intentions` (azioni a breve termine).

Benché il modello di augmented world si proponga di supportare qualsiasi tecnologia ad agenti, esso è stato concepito principalmente per l'utilizzo di agenti cognitivi, seguendo un approccio BDI.

Nella progettazione di un sistema è utile immaginare agenti che abitano in un ambiente, il quale contiene altri agenti, definendo quello che più comunemente viene chiamato *multi-agent system*.

Come mostra la figura, in un MAS gli agenti occupano uno spazio all'interno di un ambiente condiviso. In particolare, ogni agente ha una "sfera di influenza", ovvero una porzione di ambiente che è in grado di controllare, in modo parziale o meno.

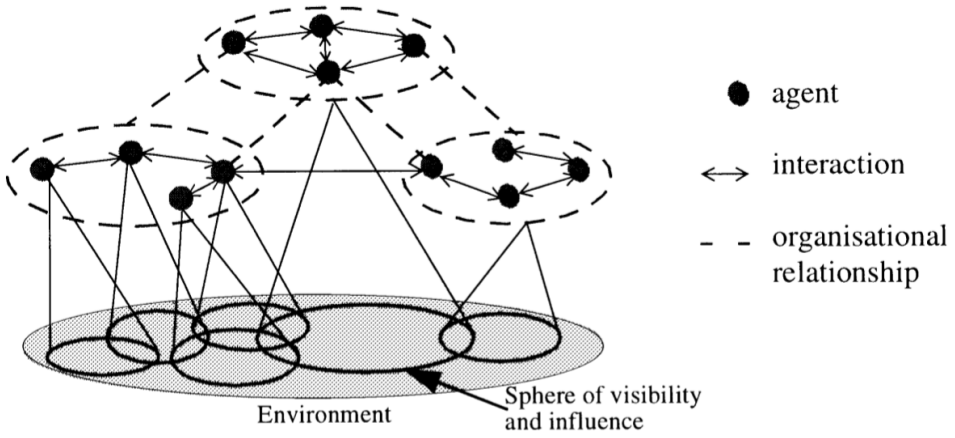

Figura 3.4: Struttura di un multi-agent system, tratta da [9]

Nell'ottica di augmented world, i MAS e l'organizzazione degli agenti possono essere sfruttati per modellare sistemi complessi di entità aumentate, che sono caratterizzate da un certo comportamento sociale e/o cooperativo. Gli agenti, inoltre, hanno una sfera d'influenza all'interno del mondo aumentato, specifica a una o più entità aumentate o a una regione, che identifica una porzione di spazio.

Lo studio degli agenti e le loro relazioni all'interno di un contesto di mixed reality è stato condotto, inoltre, in MiRA e AuRA, rispettivamente definiti in [8] e [1].

Si definisce MiRA un agente incorporato in un ambiente di mixed reality. Sulla base di questa definizione, viene proposta una tassonomia che classifica i MiRA lungo tra assi:

- **Agency:** basata sulle definizioni weak e strong di agenti, sopra illustrati
- **Corporeal presence:** descrive il grado di rappresentazione grafica o virtuale di un MiRA
- **Interactive capacity:** abilità di un MiRA di percepire e agire in un ambiente virtuale e fisico

Tale tassonomia può essere rappresentata da un cubo, che prende il nome di *MiRA Cube*.

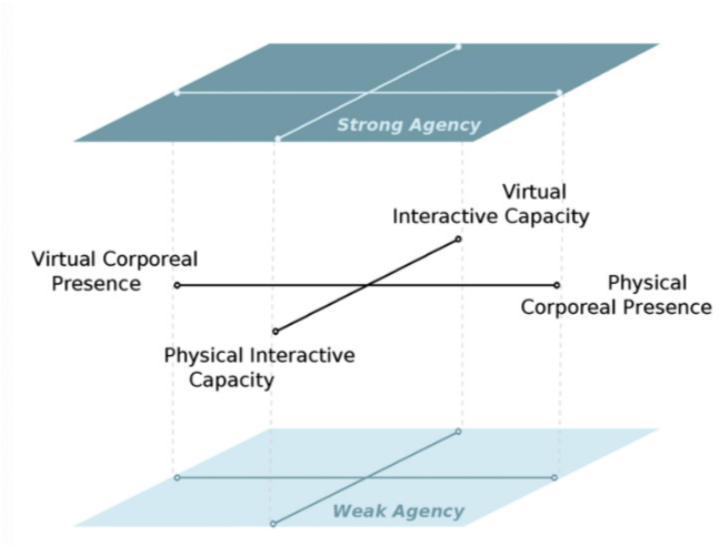

Figura 3.5: Rappresentazione di MiRA Cube proposta in [8]

Per quanto riguarda la presenza corporea, i MiRA si differenziano per il loro grado di rappresentazione nel dominio fisico e virtuale, evidenziando tre categorie distinte:

- Agenti che hanno una forte presenza corporea nel dominio virtuale rispetto a quello fisico del loro ambiente di mixed reality
- Agenti che hanno una forte presenza corporea nel dominio fisico rispetto a quello virtuale del loro ambiente di mixed reality
- Agenti che hanno una presenza corporea equamente forte sia nel dominio fisico sia in quello virtuale del loro ambiente di mixed reality

Una distinzione simile può essere applicata per la capacità interattiva, dove i MiRA si differenziano per le loro capacità di percepire e agire nel dominio fisico e virtuale. La maggior parte degli agenti che hanno una presenza corporea più forte in uno dei domini sono in grado sia di percepire che di agire in quel particolare dominio mentre, il più delle volte, sono in grado solamente di percepire l'altro dominio.

Infine, come visibile dalla figura di MiRA Cube, sono presenti i piani *Weak Agency* e *Strong Agency*. Gli agenti che presentano le capacità di autonomia, abilità sociale, reattività e proattività si posizionano nel primo piano mentre gli agenti che incorporano abilità mentali o cognitive si localizzano nel secondo.

Una tassonomia basata su questi tre assi facilita la classificazione degli agenti in termini del loro grado di incorporamento all'interno dell'ambiente di mixed reality condiviso.

A partire dalla definizione di Mixed Reality Agent proposta in [8], in [1] si definisce Augmented Reality Agent (AuRA) un MiRA che è in grado sia di percepire sia di agire nella componente virtuale della propria realtà ma, allo stesso tempo, è in grado solo di percepire la componente fisica.

Gli AuRA, quindi, esistono in un spazio condiviso di mixed reality, sono virtualmente incorporati e possono interagire con gli elementi dell'ambiente. In aggiunta, possono percepire il mondo fisico.

Nel delineare i vantaggi dell'utilizzo di paradigmi che applicano il concetto di AuRA, gli autori di [1] propongono il toolkit AFAR (Agent Factory Augmented Reality), un modulo per l'ambiente di sviluppo NetBeans che abilita gli sviluppatori alla creazione di agenti intenzionali per applicazioni di AR, sfruttando il framework NeXuS Mixed Reality.

L'obiettivo dell'architettura proposta è facilitare la prototipazione rapida e il deployment di applicazioni che combinano contenuti virtuali e fisici in un ambiente di AR attraverso l'uso di tecnologie ad agenti intenzionali.

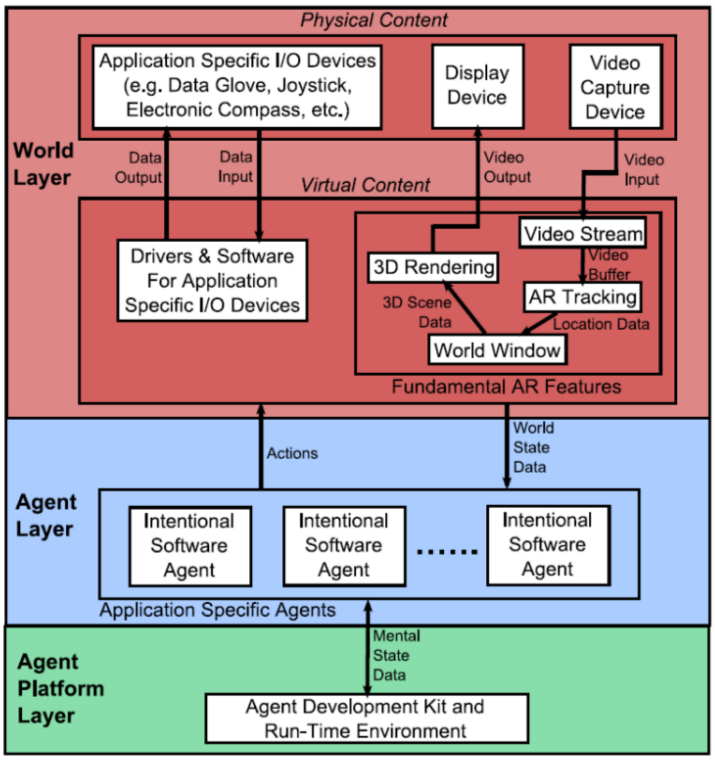

Figura 3.6: Architettura a tre livelli del toolkit AFAR, proposto in [1]

L'architettura si può dividere in tre livelli distinti:

- **Agent Platform Layer:** comprende il set di strumenti per lo sviluppo degli agenti e l'ambiente di runtime su cui distribuire detti agenti
- **Agent Layer:** contiene gli agenti specifici dell'applicazione
- **World Layer:** definisce l'ambiente stesso di AR, suddiviso a sua volta in componenti virtuali e fisici

Capitolo 4

Il framework MiRAgE per lo sviluppo ed esecuzione di mondi aumentati

Sulla base del modello concettuale di Augmented World, in [5] viene presentato MiRAgE, il primo framework che permette lo sviluppo di sistemi pervasivi agent-based di mixed reality. Si procede, quindi, nell'analisi e descrizione di detto framework, che sarà il punto di partenza, assieme al modello di AW, per realizzare l'obiettivo della tesi.

4.1 Architettura logica

Da un punto di vista strutturale, MiRAgE è caratterizzato da tre componenti principali:

- **AW Runtime:** fornisce l'infrastruttura per eseguire istanze di mondi aumentati, gestendo l'esecuzione delle entità aumentate e fornendo un'interfaccia comune agli agenti per interagire con esse e osservarle.

- **Hologram Engine:** permette la visualizzazione degli ologrammi e mantiene aggiornate le loro geometrie in tutti i device degli utenti, sfruttando le tecnologie di AR che incapsula. È eseguito sia lato server sia in ogni device degli utenti: nel primo caso fornisce supporto per il design di ogni ologramma e mantiene aggiornata la sua rappresentazione in base alle proprietà dell'entità; nel secondo caso, invece, fornisce tutte le features necessarie per visualizzare gli ologrammi, gestire la fisica e gli input e le interazioni degli utenti con gli ologrammi.

- **WoAT Layer:** basato sull'idea di *Web of Augmented Things*, proposta in [4, 3], permette all'augmented world e alle entità aumentate di essere risorse raggiungibili attraverso la rete, fornendo delle REST API per interagire con essi. Nello specifico, le operazioni GET possono essere utilizzate per ottenere informazioni sulle proprietà mentre le operazioni POST consentono di inviare azioni da eseguire sulle entità aumentate.

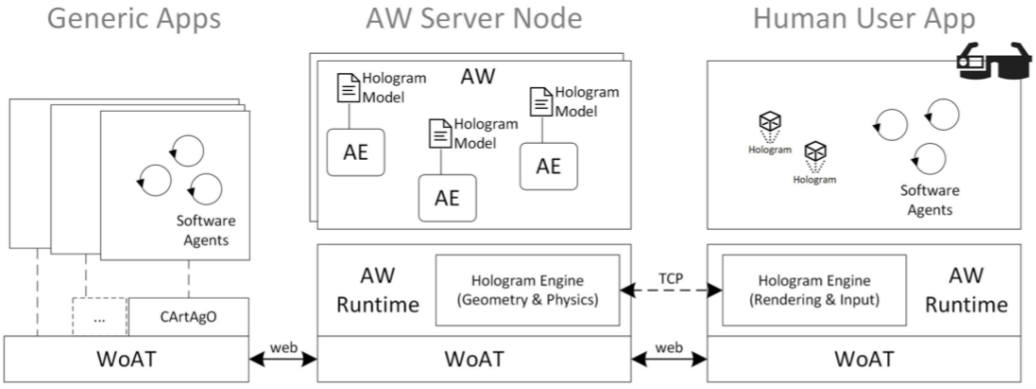

Figura 4.1: Architettura logica di MiRAgE, tratta da [5]

4.2 Hologram Engine e rappresentazione degli ologrammi

Un componente fondamentale nell'architettura di MiRAgE è `Hologram Engine`. Esso è in esecuzione sia lato server, con il compito di supportare il design delle geometrie degli ologrammi e mantenere aggiornate le loro rappresentazioni su tutti i client, sia nelle user app, fornendo tutte le features necessarie alla visualizzazione degli ologrammi e alla gestione dell'interazione con l'utente. Entrambi utilizzano un bridge, detto `Hologram Engine Bridge`, che permette di gestire la connessione TCP tra i due componenti.

Tutti gli aggiornamenti che avvengono alle entità aumentate lato server, vengono propagati a tutti i client tramite invio di messaggio sulla connessione TCP stabilita. Attualmente, Hologram Engine è in grado di ricevere i seguenti messaggi:

- **Create augmented world:** permette di creare un'istanza del mondo aumentato
- **New hologram:** permette di creare un nuovo ologramma all'interno del mondo aumentato

- **Update hologram property:** permette di aggiornare una specifica proprietà di un ologramma

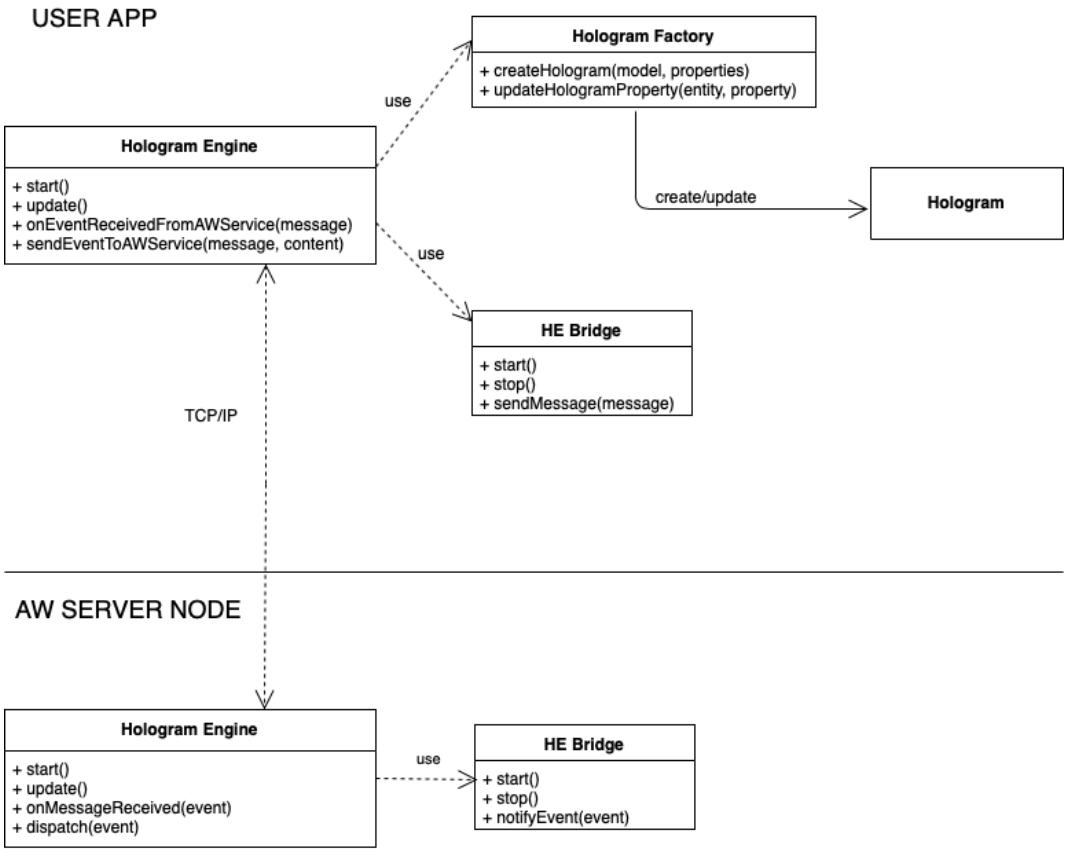

Figura 4.2: Focus su Hologram Engine lato client e server

Ricevuto un messaggio relativo agli ologrammi, Hologram Engine delega a `Hologram Factory` la creazione o aggiornamento di proprietà. In quest'ultimo caso, Hologram Factory è in grado di gestire autonomamente tre proprietà:

- **Location:** aggiorna la posizione dell'ologramma
- **Orientation:** aggiorna l'orientamento dell'ologramma
- **Extension:** aggiorna l'estensione dell'ologramma

Nel caso di proprietà *custom*, invece, viene richiamato il metodo **onCustomPropertyUpdateReceived** del Hologram Controller associato al relativo ologramma.

4.3 Il ruolo degli agenti

Gli agenti rivestono un ruolo chiave all'interno di un mondo aumentato, creando, tenendo traccia e agendo su una o più entità aumentate. Il set di azioni che possono eseguire in MiRAgE si suddivide in due gruppi principali: le primitive, ovvero funzionalità predefinite e l'insieme di azioni fornite dall'interfaccia delle entità aumentate.

All'interno del primo gruppo si evidenziano:

- **joinAW(name, location) → awID:** permette all'agente di effettuare la *join*, al fine di poter agire all'interno di uno specifico augmented world.
- **quitAW(awID):** termina la sessione di lavoro dell'agente, uscendo da uno specifico augmented world
- **createAE(awID, name, template, args, config) → aeID:** permette di creare una nuova entità aumentata in un preciso augmented world, specificando il suo nome, il suo template, i parametri e la configurazione iniziale
- **disposeAE(aeID):** permette di eliminare un'entità aumentata esistente
- **trackAE(aeID):** permette di tener traccia di un'entità aumentata esistente, ricevendo informazioni sul suo stato osservabile e sugli eventi generati
- **stopTrackingAE(aeID):** termina il tracking di un'entità aumentata esistente
- **moveAE(aeID, pos, orientation):** cambia la posizione e l'orientamento di un'entità aumentata
- **defineRegion(awID, name, region):** definisce una nuova regione, specificando nome ed estensione
- **trackRegion(awID, name):** permette di tener traccia di una regione, ricevendo informazioni sugli eventi collegati all'entrata o uscita delle entità aumentate da essa
- **stopTrackingRegion(awID, name):** termina il tracking di una regione

Nel secondo gruppo, invece, data un'entità aumentata *aeID*, che fornisce un'operazione *op*, allora l'agente avrà un'azione `doAct(aeID, op, args)`, che innesca l'esecuzione dell'operazione sull'entità aumentata.

4.4 Tecnologie software

Riferendoci all'architettura logica definita poc'anzi, la versione corrente di MiRAgE è sviluppata con un AW-Runtime scritto in Java, utilizzando *Vert.x*[1] per implementare parte del WoAT Layer.

Vert.x è un framework event-driven e non bloccante, che gira su Java Virtual Machine. Permette di gestire elementi di concorrenza, utilizzando un numero ridotto di kernel threads e contiene diversi componenti progettati per rendere più facile la scrittura di applicazioni reattive.

Le gestione degli ologrammi, invece, è affidata a Unity, equipaggiato con:

- Plugin Vuforia, per gestire aspetti di AR
- Script C#, per fornire features specifiche all'augmented world

Inoltre, sono fornite API, anch'esse scritte in Java, per sviluppare template di entità aumentate, permettendo la creazione di più istanze dello stesso template.

Infine, lato agenti viene garantita interoperabilità, permettendo di utilizzare qualsiasi piattaforma. Ad esempio, è possibile utilizzare *JaCaMo*[2], dove agenti – basati sul modello BDI – sono implementati in Jason e possono usufruire di un insieme di artefatti, basati su CArtAgO.

[1] `https://vertx.io`
[2] https://github.com/jacamo-lang/jacamo

Capitolo 5

La piattaforma ARCore

Al fine di raggiungere l'obiettivo della tesi proposto, è necessario uno studio analitico di ARCore. Questo approfondimento ci permetterà di avere una rappresentazione chiara delle funzionalità e caratteristiche della piattaforma, consentendoci già di identificare gli elementi chiave da includere nella successiva integrazione con MiRAgE, al fine di fornire un sistema robusto per la rappresentazione degli ologrammi.

5.1 Introduzione alla piattaforma

ARCore è un SDK sviluppato da Google che permette agli sviluppatori di realizzare applicazioni di augmented e mixed reality. Tra le sue origini in Tango, piattaforma anch'essa di proprietà di Google, il cui supporto è terminato nel 2018.

Tango, grazie alla fotocamera e al rilevamento del movimento e della profondità, era in grado di analizzare l'ambiente e acquisire informazioni spaziali. A causa della necessità di utilizzare sensori speciali e al ridotto numero di device compatibili, il progetto è stato abbandonato in favore di ARCore.

L'obiettivo di ARCore è, quindi, superare i limiti imposti da Tango, rendendo la realtà aumentata disponibile a tutti gli sviluppatori e supportando un maggior numero di dispositivi mobili.

Grazie a una combinazione di tecniche, quali motion tracking, comprensione dell'ambiente e stima della luminosità, insieme alle informazioni ricevute dai sensori, ARCore è in grado di tracciare la posizione e l'orientamento del device, nonché di identificare e tener traccia di punti di interesse nell'ambiente.

La continua comprensione di ARCore del mondo reale consente di posizionare oggetti, annotazioni o altre informazioni in modo tale che si integrino perfettamente con il mondo stesso.

Attualmente, ARCore è disponibile per sistemi Android e per gli ambienti di sviluppo Unity e Unreal.

5.2 Caratteristiche fondanti della piattaforma

Come abbiamo accennato, ARCore utilizza una combinazione di tecniche (ad esempio, il motion tracking) al fine di comprendere l'ambiente che lo circonda e tracciare la posizione e l'orientamento del nostro dispositivo.

In questa sezione si vuole entrare nel dettaglio della piattaforma, descrivendone le caratteristiche e i metodi che la caratterizzano.

5.2.1 Motion tracking

Una delle attività più importanti eseguite dai sistemi come ARCore è il motion tracking. Infatti, le piattaforme di realtà aumentata necessitano di sapere quando ci muoviamo e la tecnologia utilizzata a questo scopo è definita Simultaneous Localization and Mapping (SLAM).

Questo processo permette a robot o smartphone di analizzare, comprendere e orientare se stessi nel mondo fisico. SLAM necessita di diversi dati, provenienti da dispositivi hardware come la fotocamera, sensore di profondità, sensore di luminosità, giroscopio e accelerometro. ARCore utilizza queste tecnologie hardware per comprendere l'ambiente circostante e utilizza le informazioni da esse ricevute per fare il render corretto di esperienze di realtà aumentata, identificando piani o punti di interesse dove impostare delle opportune ancore.

Nello specifico, ARCore utilizza un processo chiamato Concurrent Odometry and Mapping (COM). Esso specifica allo smartphone la sua posizione nello spazio in relazione al mondo che lo circonda, catturando feature visivamente distinte nell'ambiente, meglio note come feature points. Quest'ultimi comprendono tutti quegli elementi che rimangono visibili e posizionati in modo costante nell'ambiente.

I feature point, insieme a tutti i dati relativi al movimento del nostro smartphone, permettono ad ARCore di combinarli e identificare la nostra pose. La pose, come abbiamo accennato, rappresenta la posizione e l'orientamento di un oggetto rispetto al mondo che lo circonda ed è nota anche come 6 Degrees of Freedom (6DOF). Una volta appresa la pose dello smartphone, ARCore sa dove devono essere posizionati gli elementi virtuali affinché siano coerenti con l'ambiente circostante.

Possiamo vedere l'applicazione del motion tracking con un frammento di codice C#. Come mostrato di seguito, è sufficiente mostrare a video i valori della posizione e della rotazione della fotocamera. Si noterà come ogni spostamento verrà rilevato e i nuovi valori corrispondenti saranno visualizzati.

```
void Update()
{
  if (Session.Status != SessionStatus.Tracking)
  {
     return;
  }
  var pos = Camera.main.transform.position;
  var rot = Camera.main.transform.rotation;

  Debug.Log("Camera position x: " + pos.x + " y: " + pos.y + " z: "
      + pos.z);
  Debug.Log("Camera rotation pitch: " + rot.x + " yaw: " + rot.y + "
      roll: " + rot.z);
}
```

Listato 5.1: Codice C# che mostra per ogni frame la posizione e la rotazione della fotocamera rispetto al mondo.

5.2.2 Environmental understanding

Un'altra caratteristica fondamentale di ARCore è la sua capacità di migliorare continuamente la conoscenza dell'ambiente che lo circonda, individuando nuovi punti di interesse. In particolare, ARCore identifica in automatico piani o superfici tramite una tecnica chiamata *meshing*.

Meshing è il processo di collezionare feature points al fine di costruire da essi una maglia, detta mesh. Quest'ultima, successivamente, viene rappresentata nella scena.

ARCore utilizza la fotocamera per individuare cluster di feature points lungo una superficie al fine di creare piani, permettendo l'interazione dell'utente e il posizionamento di elementi virtuali.

Nel frammento di codice proposto si mostra come ARCore identifichi nuovi piani e, per ognuno di essi, venga associato un prefab, che permette di rappresentarlo graficamente nella scena, fornendo un feedback visivo all'utente.

```
void Update()
{
  if (Session.Status != SessionStatus.Tracking)
  {
    return;
  }

  /*Iterate over planes found in this frame and instantiate
```

```
  corresponding GameObjects to visualize them.*/
  Session.GetTrackables<DetectedPlane>(m_NewPlanes,
      TrackableQueryFilter.New);

  for (int i = 0; i < m_NewPlanes.Count; i++)
  {
     /*Instantiate a plane visualization prefab and set it
     to track the new plane.*/
     GameObject planeObject = Instantiate(DetectedPlanePrefab,
         Vector3.zero, Quaternion.identity, transform);

     planeObject.GetComponent<DetectedPlaneVisualizer>()
       .Initialize(m_NewPlanes[i]);
  }
}
```

Listato 5.2: Codice C# che mostra l'identificazione e rappresentazione di nuovi piani tramite ARCore.

5.2.3 Light estimation

Light estimation è una tecnica utilizzata per replicare le condizioni di illuminazione del mondo reale e applicarle agli oggetti virtuali.

ARCore utilizza un algoritmo di analisi dell'immagine per determinare l'intensità di luce nel frame corrente, catturato dalla fotocamera e applica tale valore come luce globale per gli elementi virtuali della scena.

Come si evince dal codice, mostrato di seguito, la prima operazione che ARCore esegue è ottenere l'intensità dei pixel, analizzando l'immagine ricevuta dalla fotocamera. *PixelIntensity* restituisce un valore compreso tra 0 e 1: l'antipodo sinistro corrisponde a un'immagine totalmente nera; viceversa, la controparte a destra identifica un'immagine completamente bianca. Tale valore viene, successivamente, normalizzato con la costante *middleGray* e, infine, viene impostato come correzione di colore, moltiplicando tale valore per la *colorCorrection* precedente.

```
void Update()
{
  [...]

  if (Frame.LightEstimate.State != LightEstimateState.Valid)
  {
    return;
  }
```

```
  // Normalize pixel intensity by middle gray in gamma space.
  const float middleGray = 0.466f;
  float normalizedIntensity = Frame.LightEstimate.PixelIntensity /
      middleGray;

  /* Apply color correction along with normalized
     pixel intensity in gamma space. */
  Shader.SetGlobalColor("_GlobalColorCorrection",
      Frame.LightEstimate.ColorCorrection * normalizedIntensity);

  // Set _GlobalLightEstimation for backward compatibility.
  Shader.SetGlobalFloat("_GlobalLightEstimation",
      normalizedIntensity);
}
```

Listato 5.3: Codice C# che mostra la correzione della luminosità effettuata da ARCore.

5.2.4 Anchors e Trackables

Durante la comprensione dell'ambiente, ARCore è in grado di identificare feature point e piani, che entrambi prendono il nome di *trackable*. I trackable sono un tipo speciale di oggetti che ARCore traccerà nel tempo.

Una volta riconosciuti, l'utente è in grado di impostare delle opportune *ancore* per oggetti virtuali che intendiamo inserire nel mondo fisico. Le ancore, dette anche *anchor points*, sono punti di interesse sui quali è possibile posizionare oggetti virtuali, in particolare oggetti statici.

Supponiamo, per esempio, di voler posizionare una lampada virtuale su un tavolo. Innanzitutto, è necessario inquadrare il tavolo con il nostro dispositivo in modo tale che ARCore possa identificarlo come piano.

Successivamente, è possibile definire un punto nel quale inserire un'ancora e posizionare il nostro oggetto statico. Riferendoci alle librerie disponibili per l'ambiente di sviluppo Unity, il tracciamento dei piani è possibile associando a un GameObject lo script *Detected Plane Generator*.

In seguito, come mostrato nel frammento di codice, è possibile ottenere la lista dei piani identificati tramite il metodo *GetTrackables*, al quale si specifica il tipo di trackable desiderato (in questo caso DetectedPlane). Definito un piano, è possibile creare un'ancora in una specifica posizione tramite il metodo *CreateAnchor* (nell'esempio è stata scelta la pose centrale del piano).

Infine, si visualizza l'oggetto desiderato tramite il metodo *Instantiate*, agganciandolo a quella specifica ancora.

```
void Update()
{
  Session.GetTrackables<DetectedPlane>(m_AllPLanes);
  List<DetectedPlane> planesInTracking = m_AllPlanes.Where(p =>
      p.TrackingState == TrackingState.Tracking).ToList();

  if (planesInTracking.Count > 0)
  {
    var anchors;
    DetectedPlane plane = planesInTracking.First();

    plane.GetAllAnchors(anchors);

    if (anchors.Count <= 0)
    {
      var anchor = plane.CreateAnchor(plane.CenterPose);
      Instantiate(lampPrefab, anchor.transform);
    }
  }
}
```

Listato 5.4: Frammento di codice C# che identifica un piano e posiziona un oggetto virtuale su di esso.

Nel caso di oggetti dinamici, invece, l'ancoraggio descritto sopra non è applicabile. Tuttavia, è possibile utilizzare le ancore come punto di riferimento per gli oggetti in movimento all'interno di un ambiente. Supponiamo, infatti, di voler far ruotare un oggetto (per esempio, una sfera) attorno a un punto specifico. Una volta identificato un piano e impostata un'ancora, è possibile richiamare il metodo *RotateAround* a cui è possibile specificare la posizione e le modalità di rotazione. Nell'esempio sottostante, la sfera effettua una rotazione attorno all'ancora, definita in precedenza, a 20 gradi/secondo.

```
void Update()
{
  Session.GetTrackables<DetectedPlane>(m_AllPLanes);
  List<DetectedPlane> planesInTracking = m_AllPlanes.Where(p =>
      p.TrackingState == TrackingState.Tracking).ToList();

  if (planesInTracking.Count > 0)
  {
    var anchors;
    DetectedPlane plane = planesInTracking.First();
```

```
    plane.GetAllAnchors(anchors);

    if (anchors.Count <= 0)
    {
      var anchor = plane.CreateAnchor(plane.CenterPose);
      Instantiate(lampPrefab, anchor.transform);
    }
  }
}
```

Listato 5.5: Frammento di codice C# che identifica un piano e posiziona un oggetto virtuale su di esso.

Le ancore assumono una grande importanza nei sistemi di realtà aumentata. Infatti, il motion tracking non è esente da errori che, accumulandosi, alterano la pose del device. Le ancore permettono al sistema di correggere questi errori, indicando quali punti di riferimento sono importanti.

5.2.5 User interaction

Come abbiamo visto, la fotocamera è uno dei componenti principali nei sistemi di realtà aumentata. Negli smartphone o nei tablet essa permette di proiettare oggetti tridimensionali in un'immagine 2D, che viene mostrata all'utente.

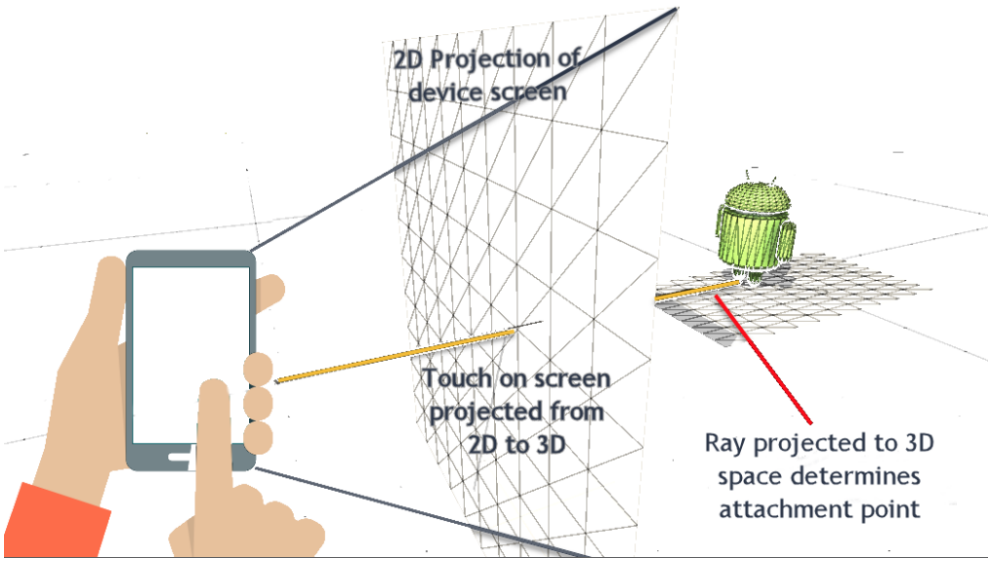

Figura 5.1: Rappresentazione della tecnica Raycast dallo schermo del device allo spazio tridimensionale (tratto da [13])

L'utente può interagire con l'immagine mostrata e, in particolare, ARCore permette di selezionare ed eseguire azioni sui trackable riconosciuti dalla piattaforma. La tecnica che permette all'utente di interagire con un particolare oggetto trackable della scena è chiamata *Raycast*.

Raycast, definito un punto bidimensionale dello schermo del device, proietta tale punto nella scena come un raggio, che viene utilizzato per identificare eventuali collisioni. Questa tecnica permette all'utente di interagire, per esempio, con i piani identificati da ARCore, selezionando alcuni punti di interesse dove posizionare ancore e, eventualmente, oggetti.

Nel frammento di codice possiamo vedere un esempio di funzionamento. Innanzitutto, viene identificata la posizione del touch dell'utente nello schermo del device.

Successivamente, viene applicata la tecnica sopra descritta tramite il metodo *Raycast* che, presi in ingresso la posizione x e y dello schermo e un filtro che identifica le categorie considerate nella collisione, scrive in una variabile di tipo *TrackableHit* il risultato.

Questa variabile contiene informazioni riguardo al trackable identificato, tra cui la sua pose e la sua distanza dall'origine del raggio.

```
void ProcessTouches()
{
  Touch touch;

  if (Input.touchCount != 1 || (touch = Input . GetTouch (0)).phase
      != TouchPhase.Began)
  {
    return;
  }

  TrackableHit hit ;
  TrackableHitFlags raycastFilter =
      TrackableHitFlags.PlaneWithinBounds |
      TrackableHitFlags.PlaneWithinPolygon ;

  if (Frame.Raycast (touch. position .x, touch. position .y,
      raycastFilter , out hit))
  {
    SetSelectedPlane (hit.Trackable as DetectedPlane);
  }
}
```

Listato 5.6: Frammento di codice C# che applica la tecnica Raycast per selezionare un piano precedentemente identificato da ARCore.

5.2.6 Augmented images

Le *Augmented Images* sono immagini 2D con le quali le applicazioni sviluppate con ARCore possono interagire, puntando verso di esse la fotocamera del dispositivo. A differenza dei classici marker, le augmented images non devono necessariamente essere di forma quadrata. È consigliato, tuttavia, che le features caratteristiche di queste immagini non siano scarse o ripetitive, al fine di facilitarne il riconoscimento.

L'utente può definire le immagini che devono essere riconosciute da ARCore, inserendole in un opportuno database oppure aggiungendole in real-time durante l'esecuzione dell'applicazione. Una volta che un'augmented image viene riconosciuta, le viene assegnata una pose.

Poichè le augmented images sono considerate come trackable, è possibile impostare anche in esse una o più ancore. Analogamente ai piani, per identificare una o più augmented image si utilizza il metodo GetTrackables, al quale andrà specificato il tipo di trackable che, in questo caso, è *AugmentedImage*.

L'utilizzo di una augmented image risulta utile quando si necessita di inserire contenuti in una specifica posizione oppure qualora fosse necessario definire un'origine dalla quale posizionare diversi oggetti virtuali.

```
void Update()
{
  Session.GetTrackables<AugmentedImage>(augmentedImages);

  foreach (var image in augmentedImages)
  {
    if(image.TrackingState == TrackingState.Tracking &&
        !instances.ContainsKey(image.Name))
    {
      var anchor = image.CreateAnchor(image.CenterPose);
      var hologram = Instantiate(prefabs[image.name],
          anchor.transform);
      instances.add(image.name, hologram);
     }
  }
}
```

Listato 5.7: Frammento di codice C# che identifica una o più augmented images e mostra uno specifico prefab in ognuna di esse.

5.3 SDK per Unity

Il SDK di ARCore per Unity si compone principalmente di un insieme di classi e strutture. In questa sezione si vogliono descrivere gli elementi principali di questa piattaforma, nello specifico:

- Trackable
- Anchor
- Session
- Frame

5.3.1 Trackable

Rappresenta un oggetto di cui ARCore sta tenendo traccia. Se ne distinguono tre:

- **FeaturePoint:** definisce un punto del mondo reale
- **DetectedPlane:** rappresenta una superficie piana del mondo fisico
- **AugmentedImage:** descrive un'immagine 2D immersa nell'ambiente

I trackable sopra elencati forniscono i seguenti metodi:

- **CreateAnchor(Pose pose):** permette di creare un'ancora nella pose fornita in input
- **GetAllAnchors(List<Anchor> anchors):** scrive in anchors la lista di tutte le ancore collegate al trackable

Inoltre, ogni tipologia di trackable fornisce delle proprietà specifiche che vengono descritte nelle sottosezioni seguenti.

FeaturePoint

Ogni FeaturePoint include:

- **Pose**: restituisce la pose del feature point
- **OrientationMode:** restituisce la modalità di orientamento del feature point

DetectedPlane

Per ogni DetectedPlane è possibile ottenere le seguenti informazioni:

- **CenterPose:** restituisce la posizione e l'orientamento del centro del piano
- **ExtentX:** restituisce l'estensione del piano nella dimensione X
- **ExtentZ:** restituisce l'estensione del piano nella dimensione Z
- **PlaneType:** restituisce la tipologia di piano (orizzontale rivolto verso il basso, orizzontale rivolto verso l'alto o verticale)

AugmentedImage

Ogni AugmentedImage include le seguenti proprietà:

- **CenterPose:** restituisce la posizione e l'orientamento del centro dell'immagine
- **DatabaseIndex:** restituisce l'indice dell'immagine all'interno del database
- **ExtentX:** restituisce la lunghezza stimata in metri della corrispondente immagine fisica
- **ExtentZ:** restituisce l'altezza stimata in metri della corrispondente immagine fisica
- **Name:** restituisce il nome dell'immagine
- **TrackingMethod:** restituisce il metodo che si sta utilizzando per tenere traccia dell'immagine

5.3.2 Anchor

Lega un GameObject a un trackable e definisce un punto di interesse nel quale è possibile posizionare oggetti. Fornisce la proprietà *TrackingState*, che rappresenta lo stato del tracking dell'ancora e può assumere i seguenti valori:

- **Paused:** il tracking dell'entità è stato messo in pausa ma potrebbe riprendere successivamente
- **Stopped:** il tracking dell'entità è stato interrotto e non verrà ripreso
- **Tracking:** il tracking dell'entità è in corso

5.3.3 Session

Rappresenta la sessione di ARCore, ovvero un punto di collegamento dell'applicazione al servizio di ARCore. Tra le proprietà di interesse si evidenziano:

- **Status**: restituisce lo stato corrente della sessione
- **LostTrackingReason:** restituisce il motivo a causa del quale ARCore ha perso il tracking

Inoltre, possono essere utilizzati i seguenti metodi:

- **CreateAnchor(Pose pose, Trackable trackable):** crea una nuova ancora nella pose legata al trackable fornito in input
- **GetCameraConfig():** restituisce la configurazione della fotocamera con la quale la sessione di ARCore è in esecuzione
- **GetTrackables<T>(List<T> trackables, TrackableQueryFilter filter):** restituisce i trackable dei quali ARCore sta tenendo traccia

5.3.4 Frame

Fornisce uno snapshot dello stato di ARCore a uno specifico timestamp associato con il frame corrente. Contiene informazioni quali stato del tracking, la posizione della fotocamera rispetto al mondo, i parametri di illuminazione stimati e informazioni sugli aggiornamenti degli oggetti che ARCore sta monitorando.

Mette a disposizione le seguenti proprietà:

- **LightEstimate:** restituisce la stima della luminosità per il frame corrente
- **Pose:** restituisce la pose del device per il frame corrente

Propone, inoltre, diverse funzioni per effettuare *Raycast*, ovvero convertire una posizione dello schermo del device in un raggio per verificare se esso intersechi un trackable.

5.4 I sistemi di riferimento

In qualsiasi applicazione 3D o di realtà aumentata, è inevitabile imbattersi in più sistemi di riferimento. Infatti, affinché il device rappresenti graficamente un oggetto virtuale attraverso le coordinate del proprio schermo, è necessario che siano eseguite diverse trasformazioni successive.

Le trasformazioni sono definite come prodotto di matrici, le quali definiscono le traslazioni e rotazioni che devono essere eseguite affinché si modifichi il sistema di riferimento a cui ci si riferisce.

Tipicamente, i principali sistemi di riferimento (descritti in seguito) sono quattro:

- Model space
- World space
- View space
- Screen space

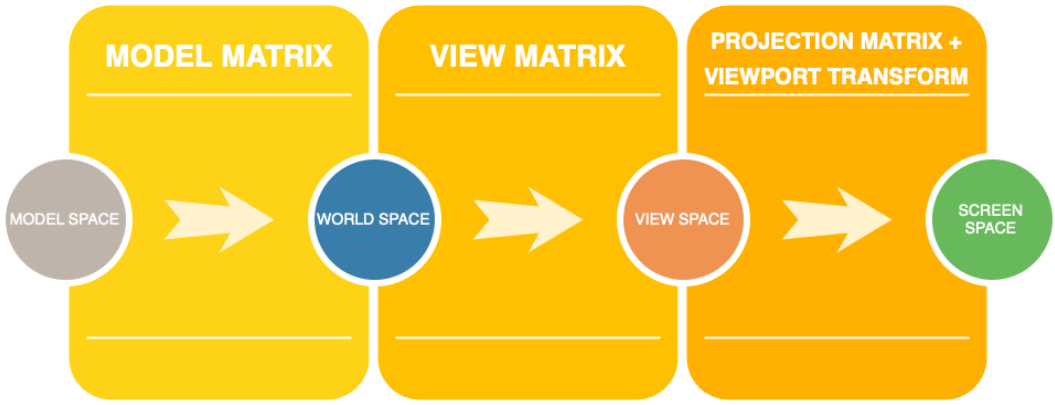

Figura 5.2: Transazione da model space a screen space tramite serie di trasformazioni consecutive

5.4.1 Model space

Definisce l'insieme delle coordinate locali a un oggetto di interesse. Supponiamo, per esempio, di avere un cubo virtuale posizionato sopra un tavolo del mondo fisico. Sia il cubo che il tavolo hanno un sistema di riferimento locale, ognuno dei quali ha un determinato punto di origine O (0, 0, 0). I punti degli oggetti, ad esempio i vertici, hanno una posizione che è relativa a tale origine.

Consideriamo un tavolo fisico di dimensioni 90 cm di lunghezza, 50 cm di profondità e 85 cm di altezza. Ipotizziamo che su detto tavolo sia stato rappresentato un cubo virtuale, di dimensioni 20x20x20 cm, posizionato a (0.20, 0.1, 0.15) rispetto al centro della base del tavolo.

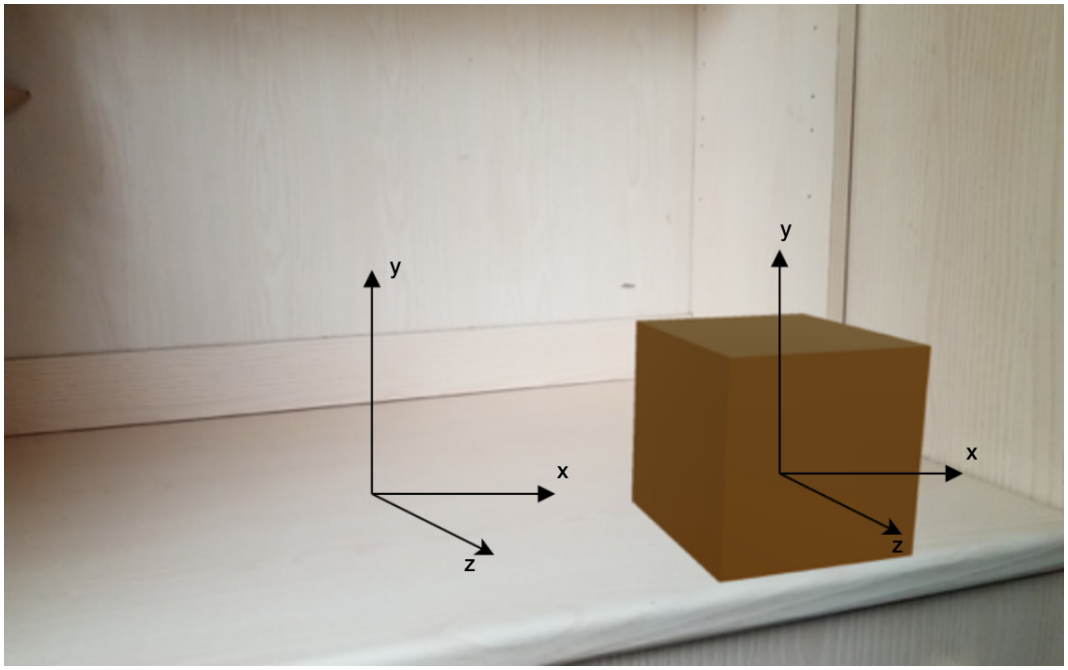

Figura 5.3: Model space: ogni oggetto ha un suo sistema di riferimento locale

La rappresentazione del cubo è data dal corrispondente prefab definito in Unity, dal quale è possibile visualizzare il sistema di riferimento locale. Supponiamo di voler conoscere la posizione di un vertice rispetto al sistema di riferimento locale del cubo. Come mostrato nel codice, è sufficiente ottenere il componente `MeshFilter` e selezionare il vertice di interesse, le cui coordinate saranno espresse localmente all'origine del cubo.

```
void Update()
{

  [...]

  MeshFilter mf = cube.GetComponent<MeshFilter>();
  Debug.Log("First vertex local position: " + mf.mesh.vertices[0] *
      scale);
```

```
  //Output
  // First vertex local position: (0.1, -0.1, 0.1)
}
```

Listato 5.8: Frammento di codice C# che mostra le coordinate locali di un vertice di un cubo

Per transitare da un sistema di riferimento prettamente locale a un sistema di riferimento del mondo, è necessaria una `model matrix`, ovvero una matrice di trasformazione che trasla, ridimensiona e/o ruota l'oggetto per posizionarlo nella locazione del mondo che gli appartiene. In ARCore, detta matrice corrisponde alla pose di un oggetto e le traslazioni definite sono espresse in metri.

5.4.2 World space

Definisce la posizione relativa di tutti gli oggetti nell'applicazione, definendo un punto di origine O (0, 0, 0) comune.

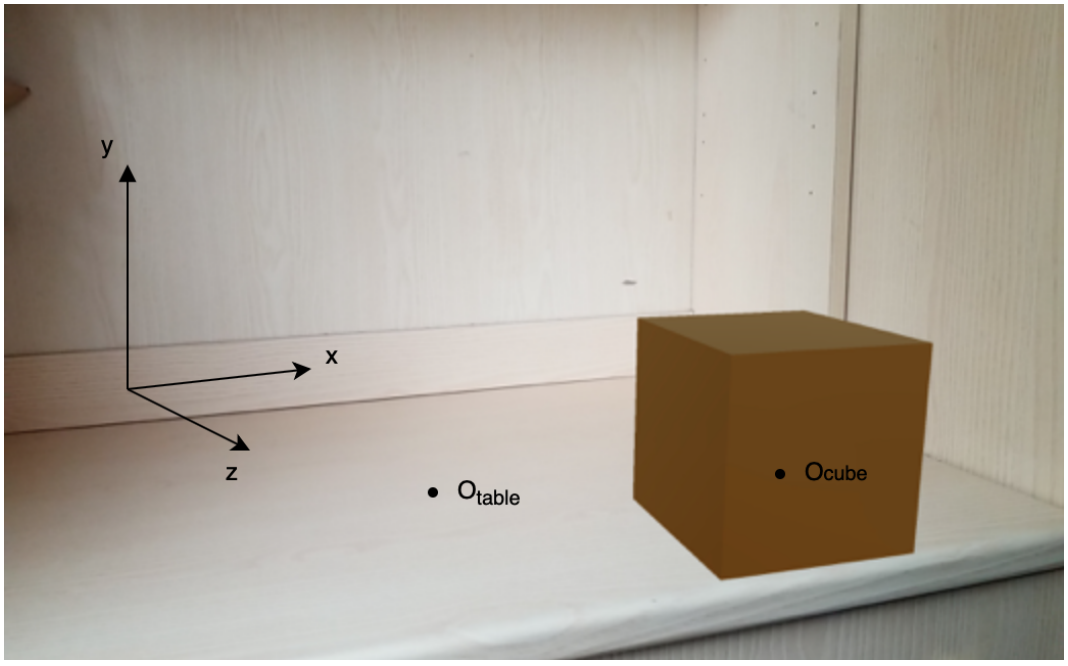

Figura 5.4: World space: le posizioni degli oggetti sono relative all'origine del mondo

Consideriamo il medesimo tavolo fisico e cubo virtuale dell'esempio precedente. Supponiamo, inoltre, di definire l'origine del mondo a un'altezza uguale

all'origine del cubo e la distanza dell'origine del mondo rispetto al centro del tavolo sia di -18 cm sull'asse delle x e -15 cm sull'asse delle z. Avremo, di conseguenza, che l'origine del tavolo rispetto a quella del mondo sarà nella posizione O_{table} (0.18, -0.1, 0.15) mentre l'origine del cubo sarà nella posizione O_{cube} (0.38, 0.0, 0.30).

Tali coordinate sono ottenibili in ARCore grazie alla pose degli oggetti, come si evince dal frammento di codice mostrato.

```
void Update()
{
  [...]

  Debug.Log("Plane position in world space: " +
      plane.CenterPose.position);
  Debug.Log("Cube position in world space: " +
      cube.transform.position);

  // OUTPUT
  // Plane position in world space: (0.18, -0.1, 0.15)
  // Cube position in world space: (0.38, 0.0, 0.30)
}
```

Listato 5.9: Frammento di codice C# che mostra le coordinate degli oggetti rispetto all'origine del mondo

In ARCore l'origine del mondo viene impostata al momento dell'avvio della sessione. Supponendo che la sessione cominci in un istante di tempo t_0, la posizione del dispositivo nel medesimo istante di tempo t_0 corrisponde all'origine del mondo. Il sistema di riferimento sarà così impostato: l'asse delle x è positivo procedendo verso destra rispetto al dispositivo; l'asse delle y è positivo spostandosi verso l'alto; infine, l'asse delle z è positivo muovendosi di fronte al device.

Conseguentemente, iniziando la sessione con due device in posizioni differenti, avremo due sistemi di riferimento diversi. La posizione del cubo, ad esempio, avrà una certa posizione per il primo device e un'altra per il secondo. In ambedue i casi, però, transitando al sistema di riferimento locale del cubo, avremo che le posizioni dei punti rispetto alla sua origine sono gli stessi.

Supponiamo di avere due smartphone, s e t, che eseguono la stessa applicazione, sviluppata con ARCore e visualizzano lo stesso cubo virtuale sopra al tavolo, come definito in precedenza.

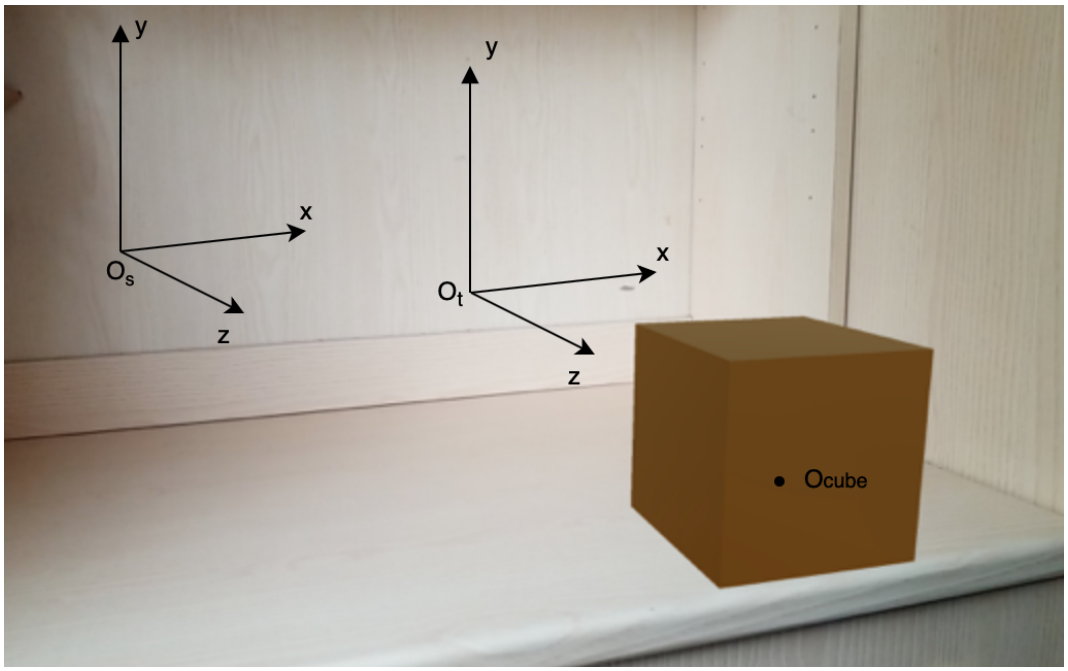

Figura 5.5: L'origine del mondo in ARCore viene fissata all'inizio della sessione: due dispositivi in posizioni diverse produrranno due world space diversi.

Poiché la loro posizione era differente al momento dell'inizio della sessione, vengono generati due sistemi di riferimento diversi. Infatti, mostrando le coordinate del cubo si nota che per lo smartphone s il cubo si trova nella posizione (0.3, 0.4, 0.4) mentre per lo smartphone t è localizzato in (0, 0.2, 0.4).

Visualizzando, invece, la posizione di un punto del cubo (ad esempio, un vertice) rispetto alla sua origine, si nota come questa sia la stessa per entrambi i device.

```
void Update()
{
  [...]

  Debug.Log("Cube position in world space: " +
      cube.transform.position);

  MeshFilter mf = cube.GetComponent<MeshFilter>();
  Debug.Log("Vertex local position: " + mf.mesh.vertices[0] * scale);

  // OUTPUT s
  // Cube position in world space: (0.3, 0.4, 0.4)
  // Vertex local position: (0.1, -0.1, 0.1)
```

```
  // OUTPUT t
  // Cube position in world space: (0.0, 0.2, 0.4)
  // Vertex local position: (0.1, -0.1, 0.1)
}
```

Listato 5.10: Frammento di codice C# che mostra i diversi output ottenuti da due dispositivi differenti su uno stesso dominio di studio

Questa analisi ci permette di fare una riflessione sulle ancore. Come abbiamo visto, le ancore possono essere utilizzate per fissare uno o più oggetti nell'ambiente. Esse possono essere adoperate, per esempio, come punto di riferimento per definire la posizione relativa di oggetti rispetto a esse.

Possiamo, quindi, vedere la posizione di un'ancora come l'origine di un sistema di riferimento, che si interpone tra un sistema prettamente locale dell'oggetto al sistema di riferimento del mondo. Immaginiamo che ARCore riconosca il piano di un tavolo e posizioni diversi cubi virtuali rispetto al centro di codesto piano, nel quale è stata posizionata un'ancora. Nonostante il sistema di riferimento del mondo cambi a seconda del punto in cui la sessione sia cominciata e, di conseguenza, la posizione assoluta degli oggetti differisca, transitando al sistema di riferimento agganciato all'ancora tali posizioni (relative) risultano uguali.

Analogamente alla trasformazione vista per il model space, è necessario utilizzare una `view matrix` per transitare dal sistema di riferimento del mondo al sistema di riferimento della fotocamera. Tale matrice, moltiplicata alla model matrix, permette l'allineamento del mondo dal punto di vista della fotocamera ed è costituita da:

- **Vettore posizione:** definisce la posizione della fotocamera rispetto al mondo
- **Vettore target:** definisce la direzione in cui sta puntando l'obiettivo
- **Vettore up:** definisce l'orientamento della fotocamera

5.4.3 View space

Definisce il sistema di riferimento dal punto di vista della fotocamera ed è il risultato della trasformazione delle coordinate del mondo in coordinate che si trovano di fronte alla vista dell'utente.

Nell'esempio proposto, il cubo aveva una posizione (0.38, 0.0, 0.3) rispetto l'origine del mondo. Supponiamo che, dopo un tempo t da cui ARCore ha cominciato la sessione, ci spostiamo con il nostro dispositivo in una posizione

(0.0, 0.0, -0.3) rispetto al mondo. Conseguentemente, impostando il sistema di riferimento dal punto di vista della fotocamera avremo che il mondo sarà in posizione (0.0, 0.0, 0.3). A sua volta, il cubo rispetto alla fotocamera si troverà nella posizione (0.38, 0.0, 0.6).

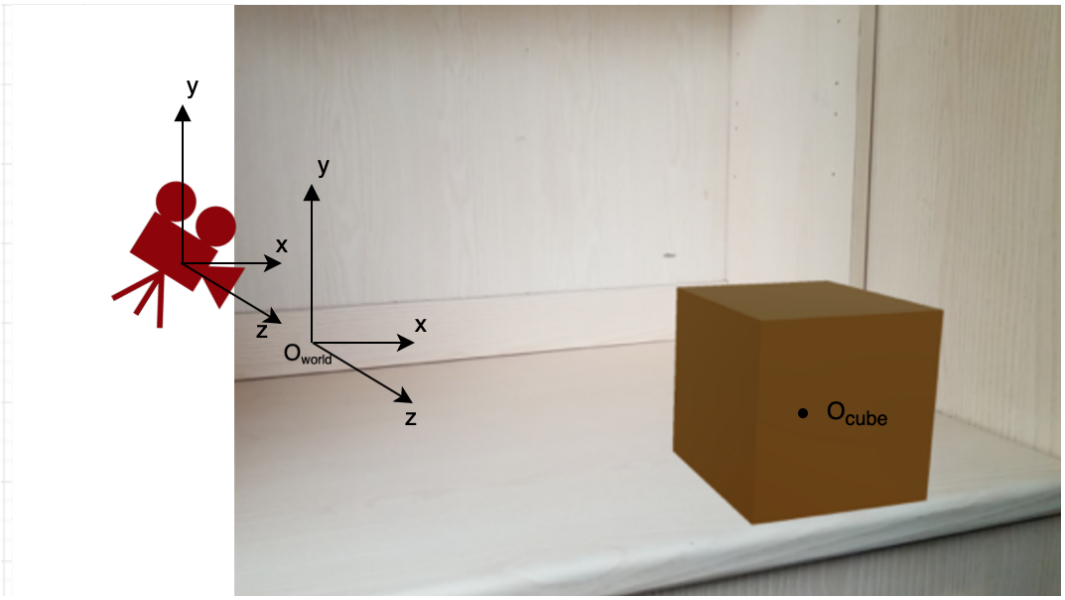

Figura 5.6: View space: il mondo è visto secondo il sistema di riferimento della fotocamera

Ora che abbiamo il punto di vista della fotocamera, l'ultimo passo è rappresentarlo secondo le coordinate dello schermo del device. Analogamente a quanto visto in precedenza, è necessaria una matrice, detta `projection matrix`, che ci permette di transitare dall'insieme delle coordinate della vista della fotocamera all'insieme delle coordinate dello schermo. Le coordinate visibili vengono mappate in un range [-1, 1] che, successivamente, tramite opportuni raffinamenti di viewport transform, convertono il range a seconda delle dimensioni dello schermo del dispositivo.

Esistono due tipi di projection matrix: *orthographic projection* e *perspective projection*.

Una orthographic projection matrix definisce un contenitore a forma di cubo di cui si specifica la larghezza, l'altezza e la lunghezza del contenitore visibile, nonché il piano più vicino e quello più lontano. Tutte le coordinate che si trovano, dopo la trasformazione, all'interno del cubo saranno visibili. Viceversa, se le coordinate hanno una posizione che supera la larghezza o l'altezza oppure si trovano di fronte al piano più vicino o dietro a quello più lontano, esse non saranno considerate.

Una perspective projection matrix, invece, considera anche la prospettiva, ovvero l'effetto secondo il quale oggetti più lontani appaiono più piccoli. Così come nella orthographic projection, viene definito un contenitore che specifica il range di coordinate considerate. Inoltre, si manipola un valore w per ogni coordinata dei vertici desiderati in modo tale che, più lontano un vertice appare, maggiore è il suo valore di w. A differenza dell'orthographic projection, il contenitore definito dalla perspective projection matrix può essere visto con una forma non uniforme, dovuta al parametro *fov* (field of view), che definisce quanto largo deve essere il campo visivo. Tale parametro è misurato in gradi, tipicamente 45.

In ARCore, di default, la projection utilizzata è di tipo perspective. Essa è visibile in Unity come componente *First Person Camera*, figlio di *ARCore Device*.

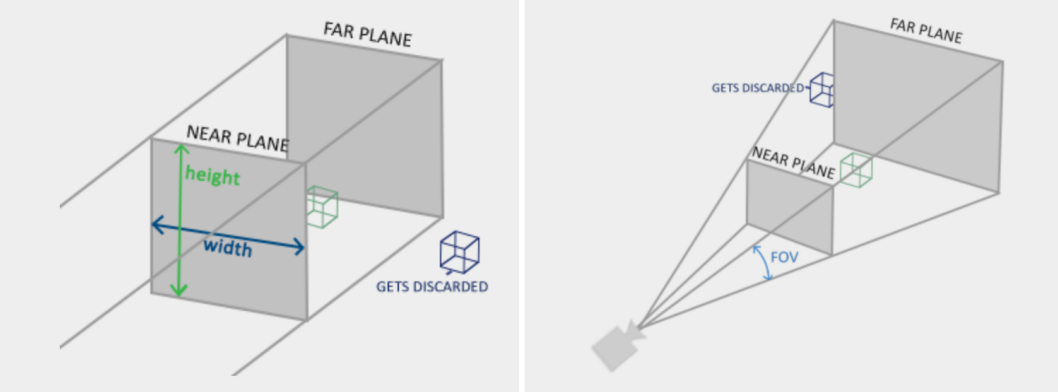

Figura 5.7: A sinistra: visualizzazione di una orthographic projection. A destra: visualizzazione di una perspective projection, tratto da `https://learnopengl.com`

5.4.4 Screen space

Rappresenta il sistema di coordinate (x, y) dello schermo del device nel quale si mostrano elementi grafici. A questo livello, le coordinate vengono espresse in pixel. Nel framework fornito per Unity la coordinata (0, 0) è posta in basso a sinistra dello schermo del device mentre (x_{max}, y_{max}) è posizionata in alto a destra.

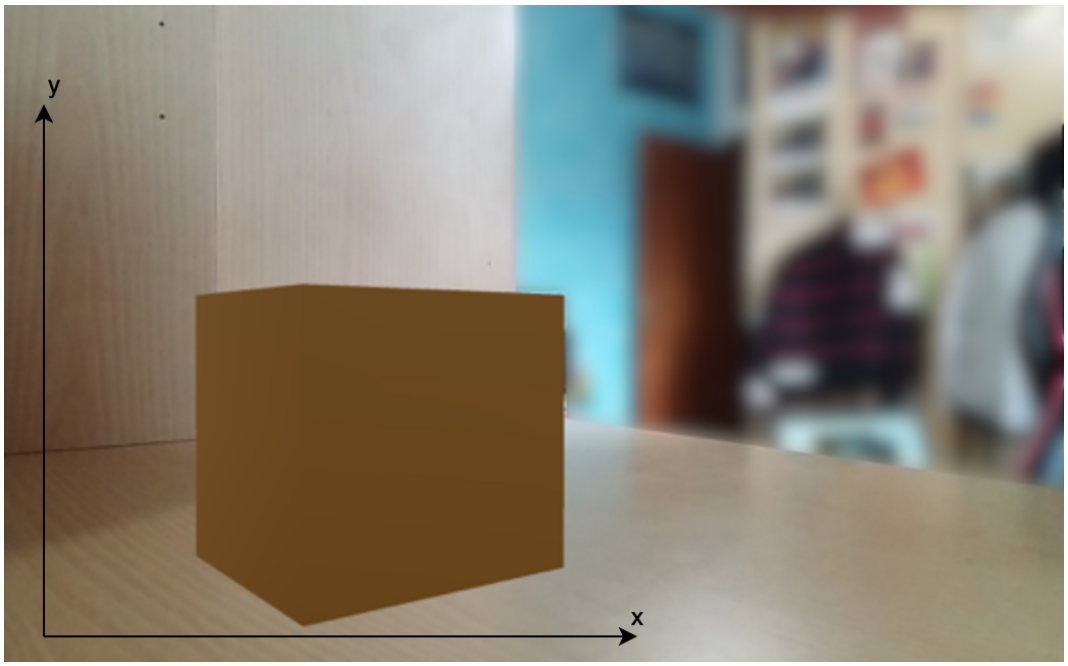

Figura 5.8: Screen space: visualizzazione della vista secondo il sistema di riferimento dello schermo.

Riferendoci all'esempio proposto, è possibile conoscere le coordinate del cubo, rispetto al sistema di riferimento dello schermo, grazie al metodo *WorldToViewportPoint*, come mostrato nel frammento di codice.

```
void Update()
{
  [...]

  Debug.Log("Cube screen coordinates: " +
      Camera.main.WorldToViewportPoint(cube.transform.position));

  // OUTPUT
  // Cube screen coordinates: " (579, 402)
}
```

Listato 5.11: Frammento di codice C# che mostra le coordinate del cubo rispetto al sistema di riferimento dello schermo

Capitolo 6

Integrazione della piattaforma ARCore in MiRAgE: analisi e progettazione

Dopo aver fornito una panoramica sulla realtà aumentata, proponendo MiRAgE come framework per lo sviluppo di mondi aumentati e approfondendo la piattaforma ARCore, come tool per lo sviluppo di applicazioni di augmented e mixed reality, si vuole entrare nel vivo dell'obiettivo della tesi.

Partendo dalla situazione attuale di MiRAgE, si definiranno nel dettaglio le estensioni che si intendono effettuare, seguendo un processo di analisi e progettazione.

6.1 Stato di MiRAgE prima dell'integrazione

Come abbiamo visto, l'architettura di MiRAgE si suddivide in tre componenti principali: `AW Runtime`, che gestisce l'esecuzione di istanze di mondi aumentati; `Hologram Engine`, che permette sia di mantenere aggiornate le rappresentazioni degli ologrammi in base alle proprietà delle entità aumentate (lato server), sia di fornire tutte le features necessarie per visualizzare detti ologrammi nei device degli utenti (lato client); infine, `WoAT Layer`, che rende l'augmented world e le entità aumentate risorse raggiungibili attraverso la rete, tramite opportune REST API.

Soffermiamoci sul Hologram Engine in esecuzione sulle user apps. Esso ha il compito di visualizzare gli ologrammi in base alle informazioni che riceve dall'omonimo componente lato server. Nello specifico, sfrutta la classe `HebEndpoint` per stabilire una connessione TCP e inviare e ricevere nuovi messaggi in formato JSON. Qualora i messaggi implichino la creazione di un nuovo ologramma

o l'aggiornamento delle sue proprietà, Hologram Engine può sfruttare la classe `Hologram Factory` che, una volta estrapolate le informazioni, procede alla gestione degli ologrammi e delle loro caratteristiche.

La visualizzazione degli ologrammi è affidata al plugin Vuforia, la cui versione attualmente utilizzata necessita di un marker per effettuare il rendering della scena. Tale marker, inoltre, deve rimanere parzialmente visibile, al fine di non perdere il tracking.

6.2 Requisiti per l'integrazione

Dallo studio della piattaforma ARCore e delle sue funzionalità, si rileva come questa sia più adatta alla gestione di ambienti smart indoor/outdoor rispetto alla controparte Vuforia.

Dette queste premesse, si vuole integrare la piattaforma ARCore all'interno del framework MiRAgE, affinché quest'ultimo sia abilitato all'utilizzo delle seguenti tecniche:

- **Motion tracking:** per identificare la posizione del device rispetto al mondo che lo circonda, tramite la tecnica markerless COM (Concurrent Odometry Mapping)
- **Environmental understanding:** per comprendere l'ambiente circostante, identificando punti di interesse e piani, permettendo di agganciarvi ancore
- **User Interaction:** per gestire l'interazione dell'utente con il mondo
- **Augmented images:** per riconoscere pattern specifici all'interno dell'ambiente

L'idea è incapsulare ARCore all'interno di MiRAgE, in modo tale che le funzionalità sopra descritte, assieme alla rappresentazione e ancoraggio degli ologrammi, siano gestiti a livello di framework, evitando al programmatore di occuparsene.

Poiché la gestione degli ologrammi è definita all'interno del Hologram Engine, sarà necessario modificare o estendere questo componente per lo scopo fissato.

L'obiettivo finale è fornire agli sviluppatori un framework robusto per lo sviluppo di sistemi agent-based pervasivi di mixed reality, sia per ambienti indoor sia per ambienti outdoor, nascondendo i dettagli implementativi e permettendo al programmatore di definire mondi aumentati specificando semplicemente:

- **Le entità aumentate:** le entità che popoleranno il mondo aumentato, che estendono la classe `AE` delle librerie di MiRAgE

- **Gli agenti:** dei quali si definisce il controllo che avranno sulle entità aumentate

- **Geometrie dell'ologramma:** definite tramite `prefabs` e inserite all'interno del progetto in Unity

6.3 Analisi dei modelli

Nello studio del modello di augmented world, abbiamo visto come questo possa essere concepito come un insieme di entità aumentate, che hanno una locazione all'interno del mondo digitale e possono essere associate a un ologramma nella controparte fisica.

Gli ologrammi sono immersi nell'ambiente e sono caratterizzati da una rappresentazione grafica, che permette loro di essere percepibili dagli utenti.

Come abbiamo visto nel capitolo precedente, ARCore fornisce un insieme di funzionalità che permettono di comprendere l'ambiente circostante, riconoscere la posizione dell'utente all'interno di esso e gestire la rappresentazione di elementi virtuali.

Tra le caratteristiche essenziali nella gestione del rendering di entità virtuali, ARCore propone i `trackable`, ovvero elementi di cui è possibile tenere traccia, che si dividono in:

- **Feature Point:** definisce un punto del mondo reale

- **Plane:** definisce un insieme di punti che costituiscono un superficie piana del mondo reale

- **Augmented Image:** definisce un'immagine del mondo reale che può essere identificata

In essi, è possibile inserire delle `anchors`, ovvero punti con una locazione specifica all'interno del mondo. A queste, è possibile associare elementi virtuali, che avranno una posizione e una rotazione rispetto all'ancora definita.

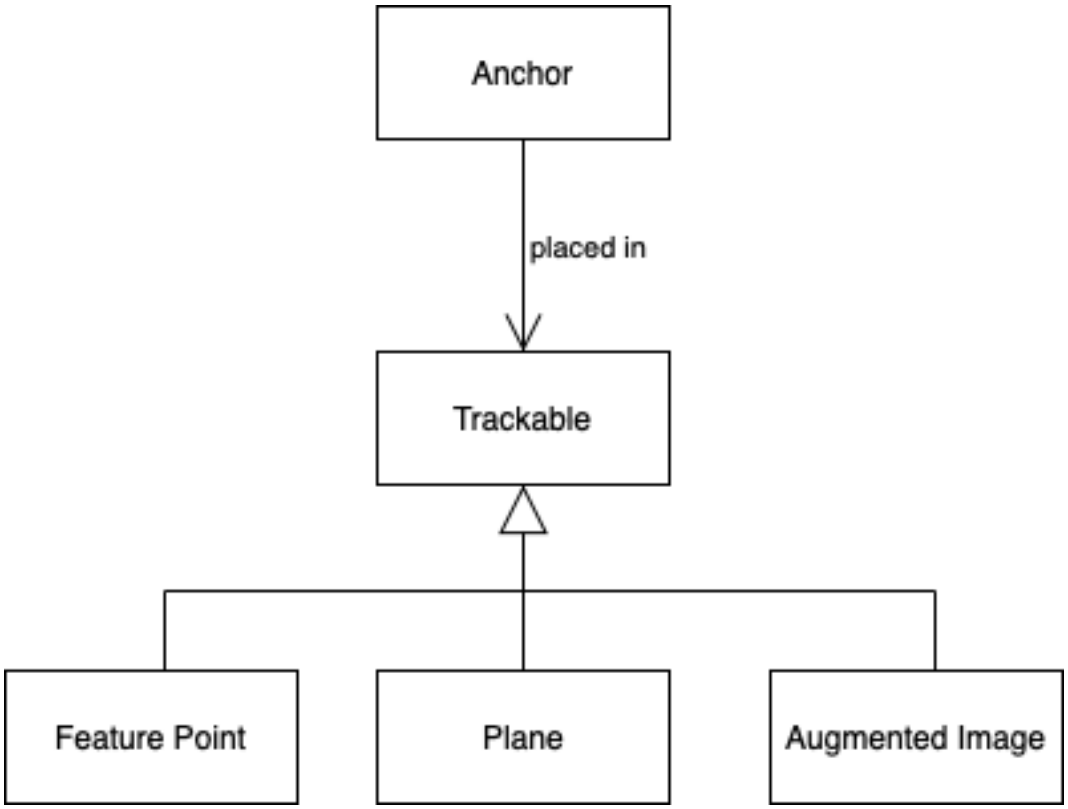

Figura 6.1: Modello degli elementi chiave di ARCore per la gestione di elementi virtuali in un ambiente

Dall'analisi effettuata, risulta evidente come le caratteristiche di ARCore possano essere integrate a livello di mondo fisico. Infatti, possiamo considerare il mondo come un insieme di trackable, ovvero feature point, piani e augmented images dislocati all'interno dell'ambiente.

Conseguentemente, la user app sarà in grado non solo di effettuare il render degli ologrammi ma, al contempo, di rilevare i trackable sopra citati, al fine di migliorare l'ancoraggio degli ologrammi e la loro rappresentazione all'interno dell'ambiente.

Il risultato è visibile nel modello seguente. Un mondo aumentato, nel livello digitale, rimane invariato e si compone di un insieme di entità aumentate. A livello fisico, l'ambiente può essere visto non solo come un insieme di ologrammi, associati a una specifica entità aumentata bensì, in aggiunta, come una composizione di trackable, nei quali può essere creata un'ancora. L'ancoraggio permette di definire un punto di riferimento per l'ologramma, al fine di migliorarne la stabilità nel mondo e una maggiore precisione.

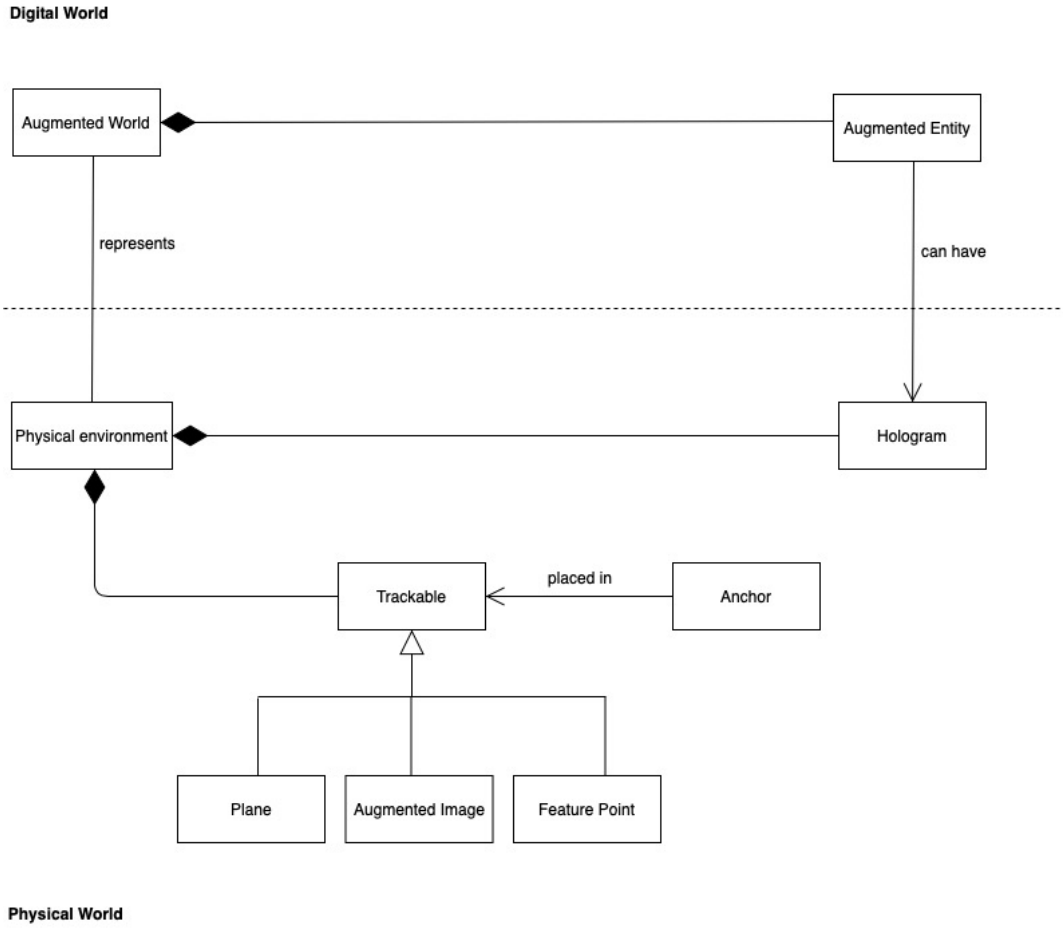

Figura 6.2: Integrazione di ARCore a livello di modello all'interno del mondo fisico

6.4 Gestione dell'integrazione di ARCore in MiRAgE

Una prima problematica da affrontare nella realizzazione dell'obiettivo della tesi è la modalità con la quale si intende incapsulare ARCore all'interno di MiRAgE. Attualmente, il core del framework, *Hologram Engine*, è sviluppato come script C#, assegnato a un GameObject della scena in Unity. Vuforia, integrato nell'ambiente di sviluppo, permette di visualizzare gli elementi che vengono generati dal Hologram Engine.

ARCore viene fornito, anch'esso, come un insieme di script C# che possono essere utilizzati o assegnati a GameObject della scena.

Occorre, quindi, definire un *link*, ovvero un collegamento con cui MiRAgE possa usufruire di ARCore. Si evidenziano, in particolare, due possibilità:

- Modifica interna dello script di Hologram Engine per utilizzare direttamente le librerie di ARCore
- Definizione di uno script aggiuntivo, che chiameremo `ARCore Extension`, in esecuzione assieme a Hologram Engine e con il quale possa interagire

La seconda opzione presenta diversi vantaggi. Innanzitutto, permette di differenziare in modo netto i compiti gestiti dal Hologram Engine, quali creazione, aggiornamento e modifica degli ologrammi e quelli gestiti dall'estensione di ARCore, come il riconoscimento dell'ambiente. Detta suddivisione facilita, inoltre, l'aggiornamento dell'estensione, in quanto un developer può effettuare le modifiche direttamente nello script aggiuntivo.

Infine, fornisce un ottimo grado di modularità al framework. Infatti, in uno scenario futuro è facile ipotizzare che MiRAgE possa essere integrato con ARKit, controparte di ARCore per dispositivi iOS oppure con una qualsiasi nuova piattaforma che distribuisca le proprie librerie per l'ambiente Unity. Sarà sufficiente, quindi, sviluppare un ulteriore script e affiancarlo alle estensioni già presenti.

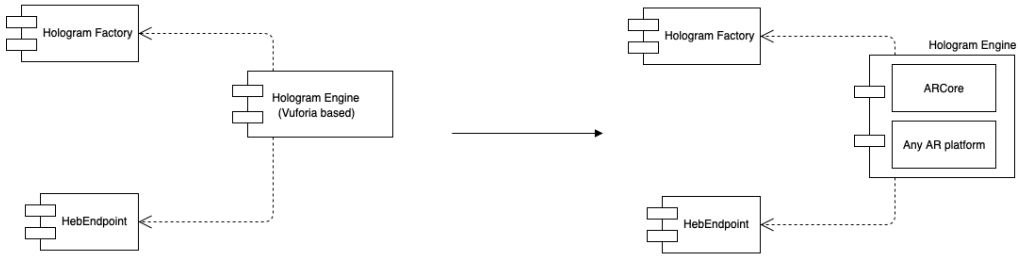

Figura 6.3: Modifica alla struttura di Hologram Engine: a sinistra la situazione originaria, a destra il risultato che si vuole ottenere

6.5 Gestione della rappresentazione del mondo aumentato

Uno dei compiti chiave di ARCore all'interno di MiRAgE sarà quello di fornire una rappresentazione del mondo aumentato e degli elementi che lo compongono.

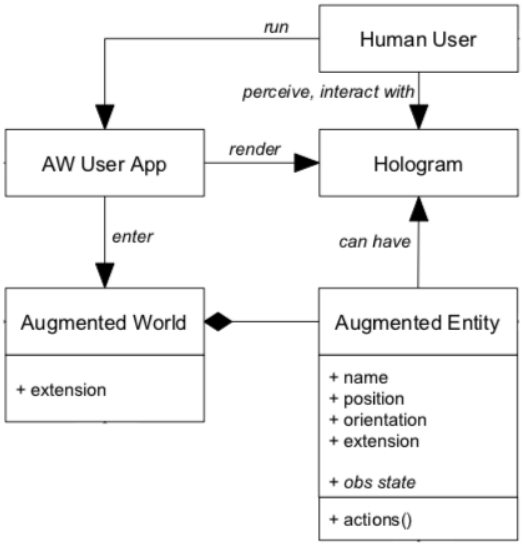

Figura 6.4: Dettaglio del modello di augmented world, che si focalizza sulla percezione del mondo aumentato da parte dell'utente come un insieme di ologrammi

Riprendiamo il modello concettuale di un augmented world: come possiamo notare, l'utente esegue una `AW User App`, che permette il rendering di ologrammi, che identificano una geometria alle entità aumentate del mondo. Ciascuna di esse definisce una posizione e una rotazione rispetto a un punto comune di riferimento, che possiamo indicare come l'origine del mondo.

Poiché siamo in un contesto di mixed reality, gli utenti dovranno contemporaneamente percepire gli stessi ologrammi nella medesima posizione, indifferentemente dalla loro locazione all'interno del mondo. È necessario, quindi, definire un punto comune nel mondo fisico, detto *punto di sincronizzazione*, che permetta agli utenti di avere una rappresentazione corretta degli ologrammi.

Come abbiamo appreso dallo studio su ARCore, esso fornisce sia tecniche markerless, come l'identificazione di piani, sia tecniche marker-based, grazie alle Augmented Images. Definire un punto di sincronizzazione comune tramite ricerca di piani risulta complesso e poco intuitivo per l'utente finale: infatti, tutti i partecipanti dovrebbero identificare lo stesso identico piano, di uguali misure, inserendo i diversi ologrammi a partire da un punto di esso. Risulta, invece, più conveniente inserire un marker dal quale definire l'origine del mondo. Gli utenti, inquadrando detto marker, avranno visibilità del mondo aumentato, con la possibilità di percepire le geometrie che definiscono gli ologrammi e interagire con quest'ultimi.

Un elemento fondamentale nel processo della rappresentazione degli ologrammi è `Hologram Engine`, il cuore di MiRAgE lato client. Esso, infatti,

ricevute le informazioni sulle entità aumentate dall'omonimo componente lato server, si avvale della classe `Hologram Factory` per creare o aggiornare gli ologrammi corrispondenti.

Al fine di migliorare l'esperienza dell'utente, incrementando l'efficienza, si ipotizza che Hologram Engine riceva già dall'avvio dell'app e a connessione avvenuta con il server le informazioni relative al mondo aumentato e alle entità che lo popolano. L'estensione basata su ARCore, che si intende sviluppare, riceverà i relativi ologrammi solamente dopo aver inquadrato il marker di riferimento, impostando quindi un'origine fisica per il mondo.

Dette queste premesse, si mostra nel seguente diagramma di sequenza come avviene la visualizzazione degli ologrammi. Notiamo che l'avvio della AW User App comporta l'esecuzione dapprima di Hologram Engine e, conseguentemente, la relativa estensione di ARCore. Quest'ultima, identificato il marker di riferimento, richiede gli ologrammi a Hologram Engine e, una volta ottenuti, li rende visibili.

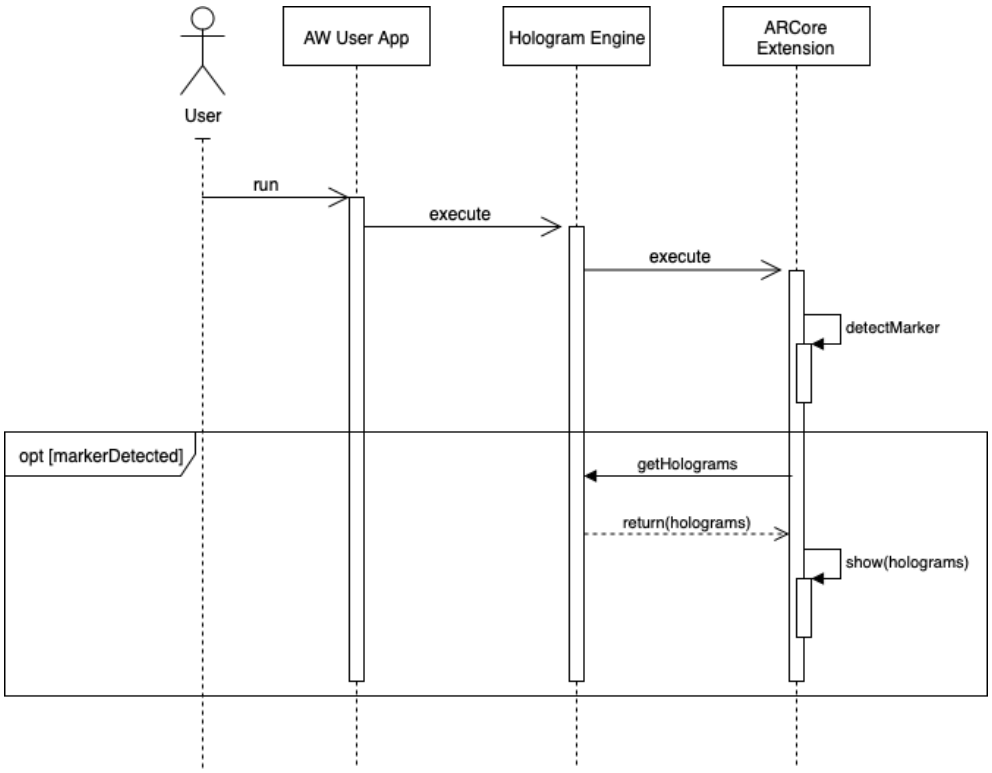

Figura 6.5: Diagramma di sequenza che mostra la visualizzazione degli ologrammi a seguito dell'identificazione del marker. Si suppone che Hologram Engine abbia già ricevuto le informazioni sulle entità aumentate dalla controparte server (non visibile in figura)

Quanto detto finora premette che esista un unico marker di riferimento, dal quale ogni utente possa sincronizzarsi. Sebbene questa soluzione sia valida, potrebbe essere necessario, soprattutto in ambienti o contesti ampi, inserire un numero maggiore di punti di sincronizzazione. Questa possibilità garantirebbe non solo agli utenti di poter accedere al mondo da diverse locazioni dell'ambiente ma di avere, in aggiunta, una maggiore precisione nella rappresentazione degli ologrammi vicini al marker selezionato, migliorando l'esperienza complessiva dei partecipanti.

Supponiamo, quindi, che un mondo aumentato possa essere acceduto da più punti di sincronizzazione, inseriti nell'ambiente. Di conseguenza:

- Occorre definire un marker di origine, che avrà posizione (0, 0, 0) e rotazione (0, 0, 0) nell'ambiente considerato
- Occorre definire la posizione e la rotazione dei marker restanti in funzione di quello di origine

Ogni qualvolta un utente inquadri un marker differente da quello dell'origine, Hologram Engine dovrà computare le nuove posizioni degli ologrammi, in modo tale che la relativa estensione di ARCore possa rappresentarli correttamente. A tale fine, è necessario un cambio di sistema di riferimento.

Dalla matematica sappiamo che, dati due sistemi di riferimento:

$$\{v_1, v_2, v_3, P_0\}$$

$$\{u_1, u_2, u_3, Q_0\}$$

Possiamo esprimere uno in termini dell'altro:

$$u_1 = \gamma_{11}v_1 + \gamma_{12}v_2 + \gamma_{13}v_3$$

$$u_2 = \gamma_{21}v_1 + \gamma_{22}v_2 + \gamma_{23}v_3$$

$$u_3 = \gamma_{31}v_1 + \gamma_{32}v_2 + \gamma_{33}v_3$$

$$Q_0 = \gamma_{41}v_1 + \gamma_{42}v_2 + \gamma_{43}v_3 + P_0$$

In modo tale da definire la matrice di cambiamento M:

$$\begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & 1 \end{bmatrix}$$

Tale per cui:

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ Q_0 \end{bmatrix} = M \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}$$

Quanto descritto è alla base dei cambiamenti di sistemi di riferimento visti in ARCore, dove si transita da un sistema di riferimento locale fino a un sistema di riferimento dello schermo del device.

Analogamente, è necessario che Hologram Engine, fornite la posizione e la rotazione del marker secondario rispetto a quello di origine, sia in grado di costruire la matrice di cambiamento in modo tale da convertire una qualsiasi posizione, rispetto all'origine, in una posizione rispetto al marker considerato.

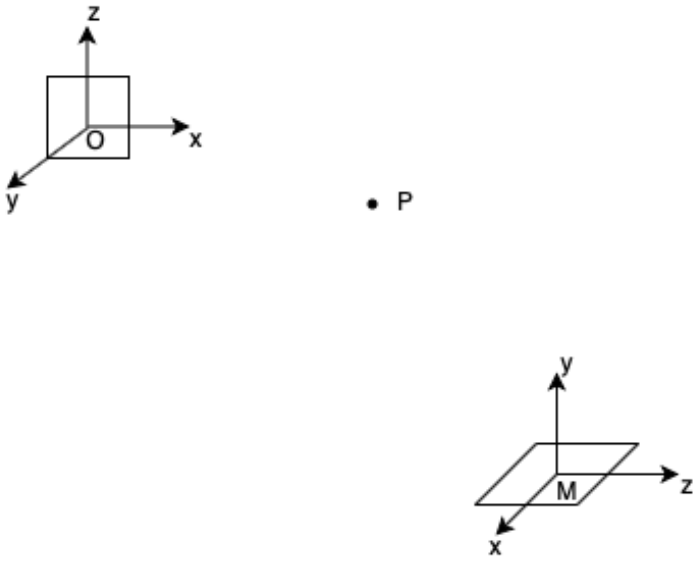

Figura 6.6: Un punto P visto dal marker di origine e da un marker secondario, avente un sistema di riferimento differente

Supponiamo, come mostrato in figura, di avere un marker principale, il cui sistema di riferimento ha origine in O e un marker secondario, il cui sistema ha, invece, origine in M. È evidente che un punto P avrà coordinate differenti a seconda del sistema di riferimento considerato. Conoscendo la traslazione e la rotazione del sistema di riferimento con origine in M rispetto a O e la posizione del punto P rispetto O, possiamo affermare che:

$$P_O = \begin{bmatrix} Custom & Matrix \end{bmatrix} P_M$$

L'inversa della matrice generata ci permette di ottenere le coordinate di P rispetto al marker con origine in M.

Il diagramma di sequenza che segue mostra come avviene l'interazione tra i componenti nella gestione di un approccio multi-marker. Ricevute le informazioni sulle entità aumentate dal server, Hologram Engine procede a creare i corrispondenti ologrammi. Successivamente, quando l'estensione basata su ARCore rileva un marker valido, invia a Hologram Engine la sua posizione e rotazione rispetto all'origine. Hologram Engine, ricevute dette informazioni, procede ad effettuare il ricalcolo delle pose degli ologrammi, generando la matrice di cambiamento sopra descritta. Gli ologrammi aggiornati vengono trasmessi all'estensione, che prontamente li visualizzerà.

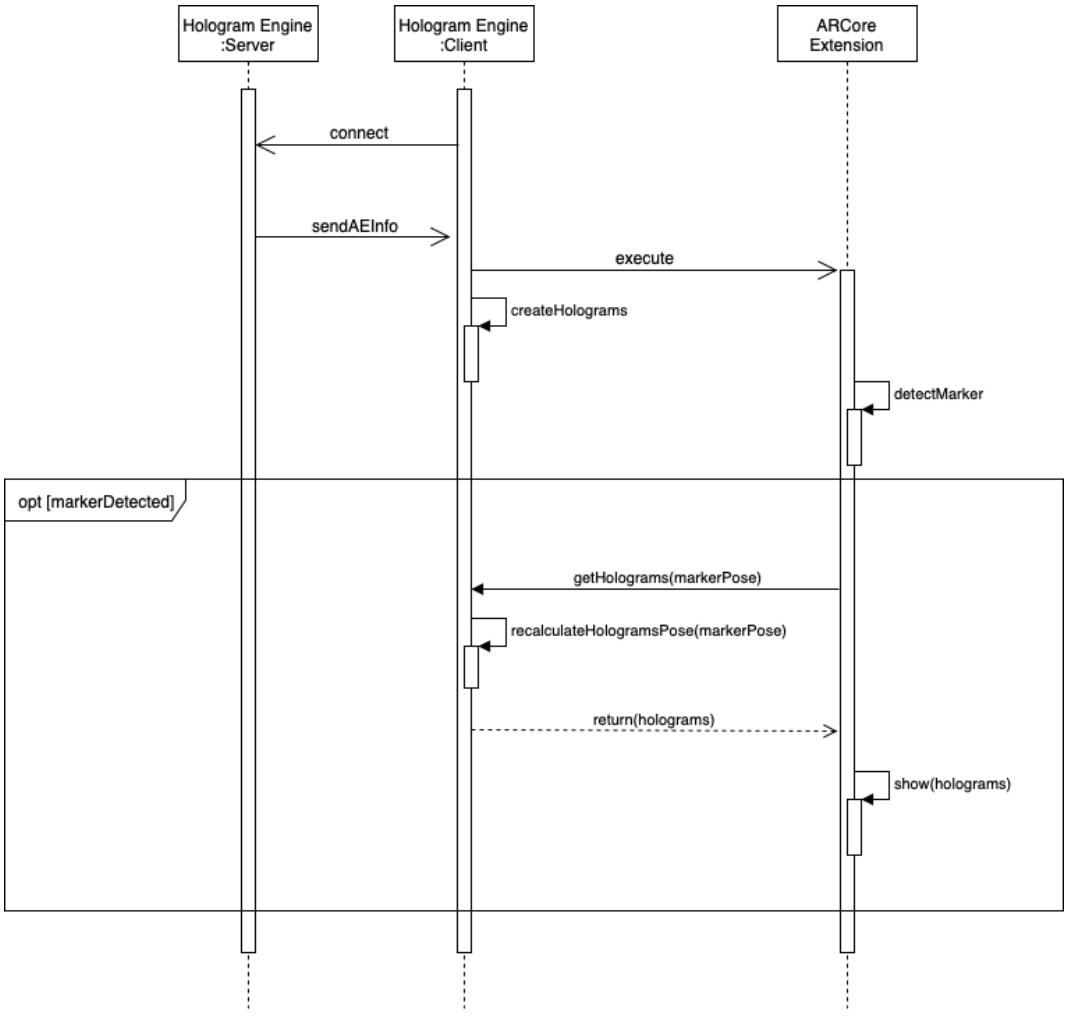

Figura 6.7: Diagramma di sequenza che mostra l'interazione dei componenti del sistema nella gestione di un approccio multi-marker

6.6 Gestione dell'interazione dell'utente

Nella visione di *augmented world* l'utente è in grado di percepire le entità aumentate come rappresentazione di ologrammi e, allo stesso tempo, egli può essere rilevato all'interno del mondo digitale.

In quest'ottica l'utente, muovendosi nel mondo fisico, è associato nel livello digitale a un'entità aumentata, che tiene traccia del suo stato durante tutta la sua permanenza all'interno del mondo.

Attualmente MiRAgE non supporta questa funzionalità, considerando l'utente solamente come partecipante del mondo fisico. Risulta opportuno, di conseguenza, considerare anche questo aspetto nel processo di integrazione della piattaforma ARCore.

Riprendiamo il modello di augmented world, focalizzandoci su `Human User`. Come si evince dal diagramma, l'utente entra all'interno del mondo grazie alla user app e ha una rappresentazione di se stesso grazie a `User Avatar`.

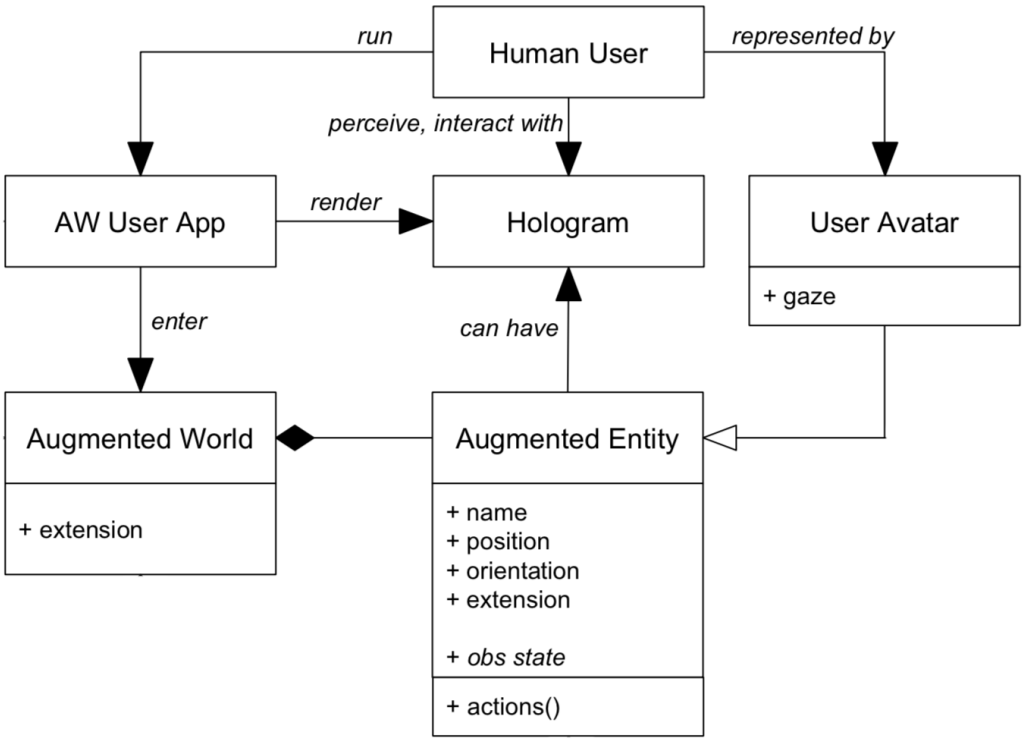

Figura 6.8: Dettaglio del modello di augmented world, che si focalizza sulla rappresentazione dell'utente nel mondo come una specializzazione di un'entità aumentata

User avatar definisce una specializzazione dell'entità aumentata: insieme alle caratteristiche che eredita da quest'ultima, propone la proprietà *gaze*, ovvero un versore rappresentante la direzione dello sguardo dell'utente.

Ogni qualvolta venga effettuata la *enter* nel mondo aumentato attraverso la user app, è necessario definire il relativo mirror dell'utente nel livello digitale, meglio noto come *avatar*.

Come abbiamo visto, un mondo aumentato può essere considerato nel livello fisico come un ambiente accessibile da diversi punti di sincronizzazione o marker. È facile ipotizzare che, al primo accesso al mondo, venga generato l'avatar dell'utente dalla user app che, successivamente, informerà il server di tale creazione.

Attualmente, la comunicazione in MiRAgE è prettamente unilaterale: Hologram Engine, in esecuzione sul server, informa tutti i client di eventi o aggiornamenti relativi alle entità aumentate, che si riflettono nel mondo fisico come modifiche agli ologrammi corrispondenti. A fronte di quanto descritto in precedenza, è necessario, invece, che la comunicazione sia bilaterale: i client, acceduti al mondo, informano il server, che procederà a generare il relativo user avatar; analogamente il server, a fronte di aggiornamenti o modifiche nelle entità del mondo, propagherà le informazioni a tutti i client connessi.

L'accesso al mondo, tramite *enter*, dovrà scatenare i seguenti eventi:

- Creazione, da parte di Hologram Engine lato client, della rappresentazione dell'utente all'interno del mondo, in termini di posizione e orientamento
- Invio delle informazioni sull'utente a Hologram Engine lato server
- Generazione dello user avatar e inserimento all'interno dell'augmented world
- Generazione delle credenziali dell'utente e invio delle stesse al client

Così come accade per gli ologrammi, anche la posizione dell'utente dipende dal sistema di riferimento fissato, ovvero dal marker inquadrato. Le entità aumentate, nel livello digitale del mondo, definiscono una posizione e una rotazione che, invece, dipendono univocamente da un sistema di riferimento considerato come origine. Conseguentemente, è necessario che, a fronte di qualsiasi posizione dell'utente, questa venga ricalcolata tramite un'opportuna matrice di trasformazione affinché si riferisca al marker di origine e possa essere memorizzata all'interno della corrispondente entità aumentata.

Il diagramma di sequenza che segue riassume tutte le interazioni che avvengono durante il processo di *enter*. Si nota come, al primo riconoscimento

di un marker, si ottengano le informazioni relative all'utente rispetto al sistema di riferimento selezionato e si inviino a Hologram Engine. Quest'ultimo, in base alle informazioni del marker, ricalcola la pose dell'utente affinché rispecchi l'origine del mondo e invia le informazioni aggiornate alla controparte server. Hologram Engine, in esecuzione sul server, ricevuto il messaggio dal client, definisce l'avatar dell'utente come un'entità aumentata, inserendolo all'interno del mondo digitale. Infine, restituisce al client lo *userId*, che definisce le credenziali tramite le quali la user app può richiedere l'aggiornamento delle proprietà relative all'utente.

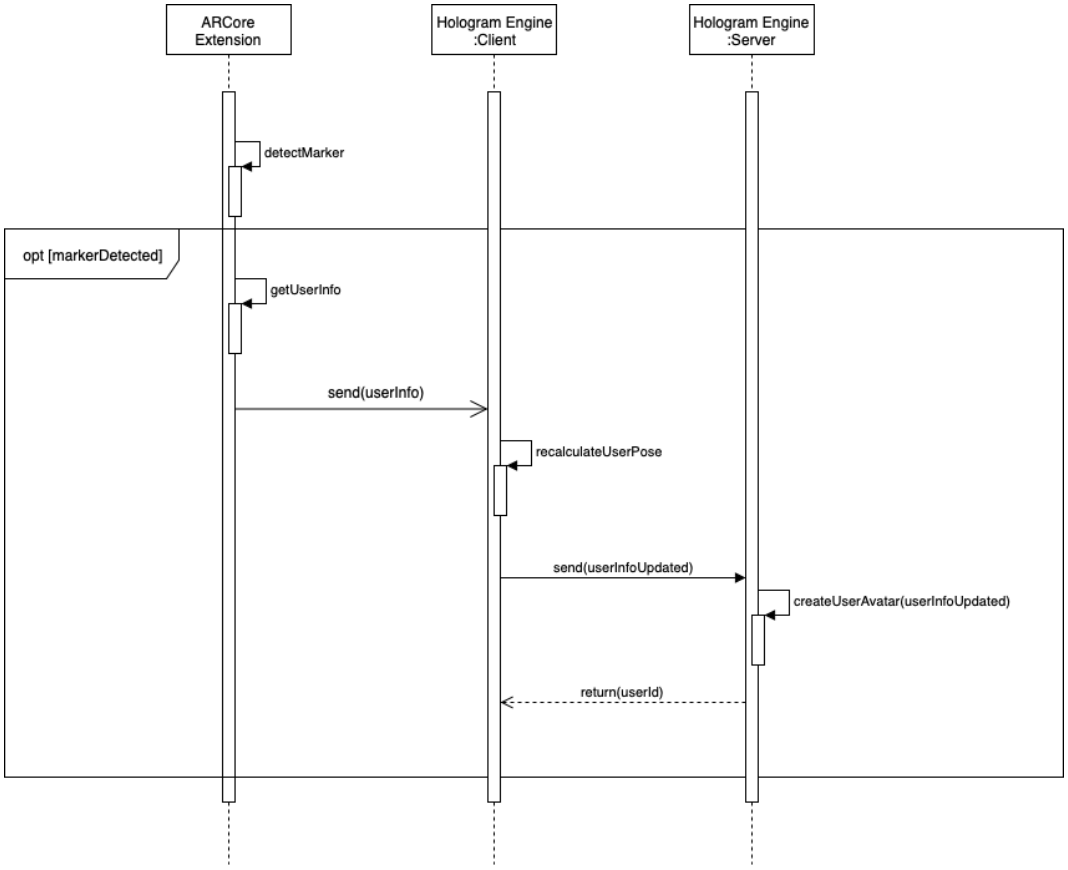

Figura 6.9: Diagramma di sequenza che riassume l'operazione di *enter* nel mondo aumentato

Ricevuto lo *userId*, l'utente sarà in grado di visualizzare gli ologrammi nel mondo fisico e potrà muoversi all'interno di esso. È inevitabile che, affinché l'identità dell'utente rispecchi quella del corrispondente avatar, i cambiamenti relativi alla sua posizione e al suo orientamento devono essere costantemente aggiornati. È opportuno, di conseguenza, che la user app, a fronte di cambia-

menti rilevanti nella pose dell'utente, invii le nuove informazioni al server, che provvederà all'update.

Il diagramma di sequenza che segue mostra le interazioni che avvengono durante il processo di *update* di uno user avatar.

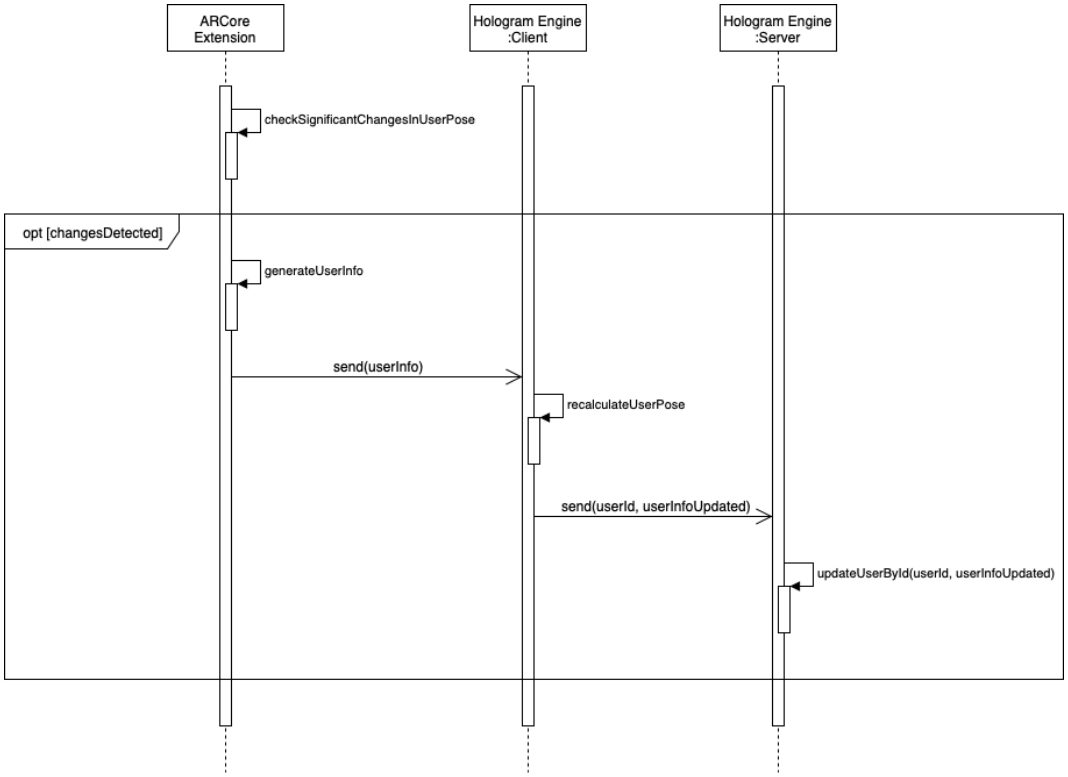

Figura 6.10: Diagramma di sequenza che mostra l'operazione di *update* delle proprietà di uno user avatar

Quando viene rilevato uno spostamento significativo da parte dell'utente (per esempio, cambiamenti di posizione od orientamento superiori a una certa soglia), le informazioni vengono inviate a Hologram Engine lato client. Quest'ultimo, dopo aver calcolato la nuova pose dell'utente rispetto al punto di origine, richiede l'update alla controparte server, inviando lo *userId* assegnato in precedenza all'utente. Hologram Engine, in esecuzione sul server, procederà ad identificare lo user avatar, tramite l'id fornito, e attuerà l'aggiornamento delle sue informazioni.

Poiché si ipotizza che l'utente si sposti continuamente durante la sua permanenza nel mondo aumentato, qualora venga rilevato un cambiamento in una sola proprietà dell'utente, anche tutte le altre verranno aggiornate. In questo modo si garantisce un maggiore livello di precisione sullo stato attuale dell'utente.

Durante l'esecuzione della user app, l'utente può trovarsi in tre stati principali:

- **Non registrato al mondo:** l'utente non ha inquadrato nessun marker, di conseguenza non ha ancora effettuato il primo accesso al mondo
- **Associato:** l'utente ha inquadrato un marker ed è correttamente associato con il rispettivo user avatar nel mondo
- **Non associato:** la user app ha perso il tracking, di conseguenza l'utente è momentaneamente dissociato dal corrispondente user avatar fino a quando non inquadra nuovamente un marker

Affinché si tenga traccia dello stato dell'accoppiamento dell'utente con la rispettiva entità aumentata, è opportuno estendere le proprietà dello user avatar, inserendo un attributo *coupled* che definisce se l'utente sia attualmente associato nel mondo e stia inviando aggiornamenti relativi a se stesso.

Inoltre, risulta opportuno che il server mantenga, per ogni avatar, un *timestamp* che identifica il momento in cui è avvenuto l'ultimo update. In questo modo un ipotetico agente, che sta monitorando lo stato degli user avatar, potrà eseguire azioni di *garbage collection*, eliminando dal mondo le rappresentazioni degli utenti che non hanno ricevuto aggiornamenti durante un periodo di tempo definito.

Capitolo 7

Implementazione di ARCore in MiRAgE

Terminate le fasi di analisi e progettazione, si procede ora a descrivere e motivare le scelte di implementazione adottate. Seguendo l'ordine proposto nel capitolo precedente, si procederà dapprima a mostrare come l'estensione basata su ARCore sia stata sviluppata e integrata in MiRAgE; successivamente, si entrerà nel dettaglio di alcune problematiche, nello specifico la gestione di un approccio multi-marker e dell'interazione dell'utente, proponendo la relativa soluzione adottata.

7.1 Implementazione dell'estensione

Come definito nella fase di analisi, l'integrazione di ARCore dovrà avvenire come estensione rispetto agli script di MiRAgE esistenti. Seguendo questo approccio, si garantisce un grado di modularità più ampio, permettendo in versioni future del framework di aggiungere ulteriori piattaforme di AR, mantenendo il core di MiRAgE pressoché inalterato e aggiungendo, invece, una nuova estensione per l'ulteriore tecnologia software che si intende incapsulare.

Poiché lavoriamo su ambiente di sviluppo Unity, al fine di raggiungere questo obiettivo il core di MiRAgE, meglio noto come *engine*, dovrà essere un oggetto padre di altrettanti *GameObject*, ognuno dei quali definirà l'estensione per la corrispondente piattaforma di AR, come visibile nella figura seguente.

Figura 7.1: L'engine di MiRAgE contiene diversi oggetti figli, ognuno corrispondente a una estensione per una piattaforma di AR

A ogni oggetto figlio sarà assegnato uno script, che implementerà l'estensione. Di default, tutti gli oggetti inizialmente non sono abilitati, a eccezione dell'engine. Sarà quest'ultimo, infatti, a tempo di esecuzione decidere quale estensione abilitare, a seconda del valore impostato dall'utente nel corrispondente menù a tendina dell'*inspector* dello script.

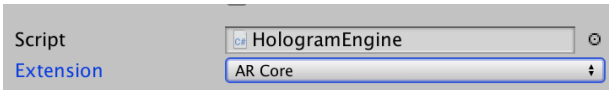

Figura 7.2: L'utente può selezionare l'estensione di interesse tramite il menù a tendina sviluppato

Come visibile nel codice proposto, un *enum* definisce tutte le estensioni disponibili. All'avvio dell'engine, ovvero il metodo *Start* dello script, viene selezionato l'oggetto figlio, corrispondente all'estensione selezionata e lo si abilita tramite il metodo *SetActive*.

Questo comporterà l'esecuzione dello script dell'estensione, che si inizializzerà anch'essa tramite il metodo *Start*.

```
void Start()
{
  [...]

  EngineManager.StartEngine();

  this.transform.GetChild((int)
      extension).gameObject.SetActive(true);
}

public enum Extension
{
    ARCore,
    Other
}
```

Listato 7.1: Esecuzione da parte dell'engine dell'estensione selezionata

Definito come l'estensione verrà eseguita dal'engine, occorre ora implementare i compiti che essa dovrà svolgere. Mentre lo script *Hologram Engine* dovrà occuparsi della ricezione dei messaggi dalla controparte server, di gestire gli ologrammi e le loro proprietà, l'estensione di ARCore, che si intende sviluppare, dovrà invece limitarsi a gestire gli elementi chiave della piattaforma, che non sono disponibili in MiRAgE.

Nello specifico, *ARCoreExtension* dovrà anzitutto comprendere l'ambiente circostante, identificando le *AugmentedImage*, ovvero marker che permettono di accedere al mondo e tracciando piani, che conferiscono maggiore stabilità nella rappresentazione. In secondo luogo, dovrà gestire le ancore in modo tale da poterle associare correttamente agli ologrammi della scena.

Nel frammento di codice è possibile vedere la gestione delle *AugmentedImage* nell'estensione sviluppata. Si nota come il tracciamento avviene solamente alla ricezione delle informazioni sul mondo aumentato dalla controparte server. Successivamente, è possibile rilevare nuovi marker e filtrare quelli che attualmente sono tracciati grazie alla proprietà *TrackingState*. Nell'ultima versione di ARCore (1.9), rilasciata a Maggio 2019, è stato aggiunta la proprietà *TrackingMethod* che ci permette di conoscere se il tracking dell'*AugmentedImage* sta avvenendo grazie all'immagine prodotta dalla fotocamera oppure dall'ultima pose nota. È opportuno, in questo caso, filtrare i marker il cui metodo di tracking sia *FullTracking*.

Benché il caso tipico, in particolare in spazi ampi, sia di un solo marker visibile nel campo visivo della fotocamera, si è ritenuto opportuno gestire la possibilità che possano essere tracciati più marker contemporaneamente. Qualora l'evento occorra, l'algoritmo seleziona il sistema di riferimento del marker più vicino alla posizione della fotocamera.

```
void Update()
{
  if (HologramEngine.Instance.AwServiceInfo == null ||
      Session.Status != SessionStatus.Tracking)
  {
      HideHolograms();
      return;
  }

  Session.GetTrackables<AugmentedImage>(markersTracked,
      TrackableQueryFilter.Updated);

  //ARCore 1.8
  /*List<AugmentedImage> tracking = markersTracked.Where(m =>
      m.TrackingState == TrackingState.Tracking).ToList();*/

  //ARCore 1.9
  List<AugmentedImage> tracking = markersTracked.Where(m =>
      m.TrackingMethod ==
      AugmentedImageTrackingMethod.FullTracking).ToList();

  if (tracking.Count > 0)
```

```
  {
    if (tracking.Count <= 1)
    {
        AugmentedImage im = tracking.First();

        if (trackedImage == null || im.Name != trackedImage.Name)
        {
            trackedImage = im;
            reload = true;
        }
    }
    else
    {
       trackedImage = GetNearestImage(tracking);
       reload = true;
    }
  }

  [...]

}

private AugmentedImage GetNearestImage(List<AugmentedImage> tracking)
{
    Dictionary<AugmentedImage, float> aiDistances = new
        Dictionary<AugmentedImage, float>();

    return tracking.Aggregate((first, second) =>
        Vector3.Distance(first.CenterPose.position,
        Camera.main.transform.position) <
        Vector3.Distance(second.CenterPose.position,
        Camera.main.transform.position) ? first : second);
}
```

Listato 7.2: Frammento di codice relativo a *ARCoreExtension* che mostra come avviene la rilevazione e la selezione del marker di riferimento

Al fine di migliorare la stabilità della rappresentazione, risulta opportuno creare l'ancora non direttamente nell'augmented image, bensì nel piano sottostante. Benché questa implementazione aggiunga un ritardo nella visualizzazione degli ologrammi, dipendente da un tempo variabile T impiegato per il riconoscimento del piano, la stabilità è maggiore.

Nel codice sottostante si evidenzia l'aggiornamento proposto in cui, a seguito del riconoscimento del'augmented image, si procede a effettuare la tecnica

raycast per verificare la presenza di un piano. Il punto di raycast corrisponde alla pose centrale del marker vista dal sistema di riferimento dello schermo del device. Solamente a seguito dell'identificazione dell piano, si procede al caricamento della scena, impostando l'ancora e visualizzando gli ologrammi.

```
void Update()
{
  [...]

  planeDetected = DetectPlaneInImage(im);

  if ((trackedImage == null || im.Name != trackedImage.Name) &&
      planeDetected != null)
  {
      trackedImage = im;
      reload = true;
  }

  [...]
}

private DetectedPlane DetectPlaneInImage(AugmentedImage im)
{
   TrackableHit hit;
   DetectedPlane plane = null;

   Vector3 imageCenterOnScreen =
       Camera.main.WorldToScreenPoint(im.CenterPose.position);

   if (Frame.Raycast(imageCenterOnScreen.x, imageCenterOnScreen.y,
       TrackableHitFlags.PlaneWithinPolygon, out hit))
   {
       bool isHitFromTheBackOfThePlane =
          Vector3.Dot(Camera.main.transform.position -
          hit.Pose.position, hit.Pose.rotation * Vector3.up) < 0;

        if ((hit.Trackable is DetectedPlane) &&
            isHitFromTheBackOfThePlane)
        {
            Debug.Log("Hit at back of the current DetectedPlane");
        }
        else
        {
            plane = (DetectedPlane)hit.Trackable;
```

```
        }
    }
    return plane;
}
```

Listato 7.3: Aggiornamento in *ARCoreExtension* per la creazione dell'ancora nel piano sottostante il marker

7.2 Implementazione di un approccio multi-marker

Durante la permanenza all'interno del mondo, l'utente può sincronizzarsi con diversi marker, posizionati nell'ambiente. Ognuno di questi definisce una pose rispetto al marker di origine e, conseguentemente, anche gli ologrammi devono riflettere l'insieme di traslazioni e rotazioni.

A tale fine, è necessario innanzitutto definire un file di configurazione che permetta di specificare rapidamente l'insieme dei marker disponibili e la loro locazione nello spazio. Come visibile nel codice proposto, esso è costituito da un file JSON dove si definisce un array di marker, ognuno caratterizzato da un nome, una posizione e una rotazione.

```
{
    "markers": [
      {
        "name": "origin",
        "position": [0.0, 0.0, 0.0],
        "rotation": [0.0, 0.0, 0.0]
      },

     {
        "name": "second",
        "position": [0.0, -0.49, 0.0],
        "rotation": [0.0, 0.0, 180.0]
      },

      {
        "name": "third",
        "position": [0.0, 0.82, -0.48],
        "rotation": [-90, 0.0, 180.0]
      }
   ]
}
```

Listato 7.4: Esempio di file di configurazione per definire la dislocazione e la rotazione dei diversi marker

Definiti i marker, l'estensione basata su ARCore dovrà riconoscerli e inserire le opportune ancore da associare come *parent* degli ologrammi. Questo implica che la nuova origine per gli elementi sarà definita da quell'ancora, posizionata nel piano in una posizione corrispondente all'incirca al centro del marker.

Il calcolo delle posizioni e rotazioni aggiornate, invece, viene delegato a Hologram Engine, in particolare al componente Hologram Factory. Come possiamo vedere nel frammento di codice, ottenuti i valori di posizione e rotazione del marker, questi vengono impostati in Hologram Factory, in modo tale che possa effettuare l'update degli ologrammi.

```
private void ReloadScene()
{
    Vector3 markerPosition = markers[trackedImage.Name].Position();
    Quaternion markerRotation =
        markers[trackedImage.Name].Rotation();

    HologramFactory.SetExtraPosition(markerPosition);
    HologramFactory.SetExtraRotation(markerRotation);

    foreach (Hologram h in HologramsManager.GetAllActiveHolograms())
    {
        HologramFactory.UpdateHologramPropertiesById(h.Id);
    }

    [...]
}
```

Listato 7.5: Delegazione a Hologram Factory del ricalcolo della pose degli ologrammi

Come abbiamo visto nella fase di analisi e progettazione, è necessaria una matrice di trasformazione per effettuare l'aggiornamento della posizione degli ologrammi in relazione al nuovo sistema di riferimento. Avendo a disposizione la pose del marker rispetto all'origine e la posizione originaria dell'ologramma, possiamo creare in Unity una matrice TRS (Translation - Rotation - Scale) e utilizzare la sua inversa per calcolare la nuova locazione. Per quanto riguarda la rotazione, invece, sarà sufficiente calcolare l'inversa di quella associata al marker e moltiplicarla per la rotazione originaria dell'ologramma.

```
HologramController hc = HologramsManager.GetHolograByEntityId(id)
```

```
   .Object.GetComponent<HologramController>();
Vector3 position = hc.gameObject.transform.localPosition;
Quaternion rotation = hc.gameObject.transform.localRotation;

Matrix4x4 cm = Matrix4x4.TRS(extraPosition, extraRotation,
    Vector3.one);

hc.gameObject.transform.localRotation =
    Quaternion.Inverse(extraRotation) * rotation;

hc.gameObject.transform.localPosition =
    cm.inverse.MultiplyPoint3x4(position);
```

Listato 7.6: Ricalcolo della posizione e della rotazione di un ologramma

7.3 Implementazione della gestione dell'interazione dell'utente

Nella versione attuale di MiRAgE l'interazione dell'utente non è gestita, il quale diventa solo uno spettatore passivo all'interno del mondo aumentato. Al fine di renderlo percepibile dalle entità aumentate, è opportuno implementare tale funzionalità a livello client e aggiornare il server, in modo tale che sia in grado di gestirne la creazione e l'aggiornamento di un avatar corrispondente all'utente.

7.3.1 Lato client

La prima operazione da eseguire per raggiungere l'obiettivo proposto è modellare l'entità `UserAvatar`, ovvero una classe che definirà il nostro utente e le sue proprietà, tra le quali la pose e il gaze.

Successivamente, occorre esplicitare come l'estensione basata su ARCore possa, a partire dal marker di riferimento, ricavare la pose e il gaze dell'utente da inviare a Hologram Factory, che si occuperà di ricalcolare tali informazioni rispetto al punto di origine. A tale fine, è opportuno definire due *GameObject* non visibili, che terranno traccia della pose dell'utente e del gaze grazie alle proprietà definite in *Camera.main.transform*. Queste informazioni sono relative all'origine del mondo che, come sappiamo dallo studio di ARCore, corrisponde alla posizione in cui il device si trovava al momento dell'avvio del tracking. È possibile convertire dette informazioni in relazione al marker di riferimento, impostando l'ancora corrispondente come *parent* di tali oggetti.

Infine, si inviano le informazioni al server sullo user avatar, la cui creazione è delegata a Hologram Factory, in formato JSON.

```
//Set user avatar local pose
this.userAvatar.transform.position = Camera.main.transform.position;
this.userAvatar.transform.rotation = Camera.main.transform.rotation;
this.userAvatar.transform.SetParent(anchor.transform);

//Set gaze local position
this.gaze.transform.position = Camera.main.transform.forward;
this.gaze.transform.SetParent(anchor.transform);

//Create corresponding avatar
UserAvatar avatar = HologramFactory.CreateUserAvatar(
    this.userAvatar.transform.localPosition,
    this.userAvatar.transform.localRotation,
    this.gaze.transform.localPosition, coupled);

if (toCreate)
{
    HologramEngine.Instance.sendEventToAWService(
        C.Messages.CREATE_USER_AVATAR, avatar.ToJSON());
}
else
{
    if (HologramEngine.Instance.UserId != C.AppString.DEFAULT_USER)
    {
        HologramEngine.Instance.sendEventToAWService(
            C.Messages.UPDATE_USER_AVATAR, avatar.ToJSON());
    }
}
```

Listato 7.7: Calcolo della pose e gaze locali dello user avatar, creazione dello stesso tramite Hologram Factory e invio dei messaggi a Hologram Engine lato server

Esistono due tipologie di messaggio che il client può inviare a Hologram Engine in esecuzione sul server:

- **Create user avatar:** si richiede al server di creare lo user avatar nel livello digitale del mondo, con le informazioni fornite nel messaggio
- **Update user avatar:** si richiede al server di aggiornare lo user avatar esistente, corrispondente allo *userId* fornito nel messaggio

Il primo messaggio richiede al server di generare, assieme allo user avatar, lo *userId* da fornire al client ogni qualvolta ci sarà un aggiornamento sulla pose dell'utente. Risulta, quindi, necessario che Hologram Engine in esecuzione sul client sia in grado di ricevere un'ulteriore tipologia di messaggio, ovvero **user avatar created**, che confermi l'avvenuta creazione dell'avatar e fornisca lo *userId* generato.

Il secondo messaggio, invece, viene generato ogni qualvolta la pose dell'utente all'interno del mondo si modifichi in modo considerevole. Nello specifico, quando l'orientamento o la posizione si discostano rispetto a una soglia definita, viene creato uno user avatar con i valori aggiornati e il messaggio corrispondente di aggiornamento viene inviato al server, che lo processerà.

Così come per gli ologrammi, la pose di un utente è in relazione al marker considerato. Viceversa, lato server, tutte le entità definiscono una posizione e una rotazione in riferimento a un unico sistema di origine. Risulta necessario, di conseguenza, definire anche per lo user avatar una matrice di trasformazione TRS che permette di calcolare le informazioni sulla pose dell'utente e sul gaze corrispondente in relazione all'origine, dato un qualsiasi marker di riferimento, come visibile nel frammento di codice proposto.

```
public static UserAvatar CreateUserAvatar(Vector3 position,
    Quaternion rotation, Vector3 gaze, bool coupled)
{
    UserAvatar userAvatar = new
        UserAvatar(HologramEngine.Instance.UserId, USER_TAG);
    GameObject uaObject = new GameObject();

    Matrix4x4 cm = Matrix4x4.TRS(extraPosition, extraRotation,
        Vector3.one);

    uaObject.transform.localPosition = cm.MultiplyPoint3x4(position);
    uaObject.transform.localRotation = extraRotation * rotation;

    userAvatar.Object = uaObject;
    userAvatar.Gaze = cm.MultiplyPoint3x4(gaze);
    userAvatar.Coupled = coupled;

    return userAvatar;
}
```

Listato 7.8: Ricalcolo della pose dello user avatar e del gaze corrispondente in relazione all'origine

7.3.2 Lato server

Attualmente Hologram Engine, in esecuzione sul server, ignora i messaggi in ingresso, preoccupandosi solamente di inviare gli aggiornamenti a tutti i client connessi. Come abbiamo visto, affinché l'utente sia un'entità attiva all'interno del mondo aumentato, è opportuno che il server sia in grado di ricevere i messaggi provenienti dai client.

Nello specifico, è stata aggiunta una coda di messaggi in ingresso, definita dal tipo *BlockingQueue*, nella classe *HologramEngineBridge*. Ogni messaggio, che viene ricevuto dalla socket, viene aggiunto alla coda. Un thread, denominato *InboxHandler*, rimane in ascolto sulla coda e, qualora fosse disponibile un nuovo messaggio, provvede ad analizzarlo. In base alla tipologia, viene eseguita l'azione corrispondente. Come abbiamo visto dalla controparte client, i messaggi accettati definiscono la creazione di un nuovo user avatar o l'aggiornamento del medesimo.

```
class InboxHandler extends Thread {

    private volatile boolean stop;

    public InboxHandler() {
      this.stop = false;
    }

    @Override
    public void run() {
        while(!stop) {
          try {
            JsonObject message = inbox.take();
            JsonObject event = message.getJsonObject("event");
            String awID = event.getString("awID");
            String hostIp = message.getString("host");

            switch (event.getString("message")) {

            case C.Heb.CREATE_USER_AVATAR:
                createUserAvatar(event, awID, hostIp);
                break;

            case C.Heb.UPDATE_USER_AVATAR:
                updateUserAvatar(event, awID);
                break;
```

```
            default:
                break;
            }
            Thread.sleep(200);
           } catch (Exception e) {
            System.err.println(e.getMessage());
           }
    }
}
```

Listato 7.9: Implementazione di InboxHandler per gestire i messaggi in ingresso

Capitolo 8

Validazione e collaudo

In questo ultimo capitolo si procede a descrivere i diversi processi atti a validare il framework sviluppato. Dopo diversi unit test, utilizzati per verificare il corretto funzionamento di alcuni componenti, si costruiscono degli esempi mirati a validare le funzionalità sviluppate, elencando i risultati e le misurazioni effettuate. Infine, si presenta Rocca delle Caminate come contesto per testare il framework, nell'ottica di valorizzare il patrimonio culturale inserendo elementi di mixed reality.

8.1 Collaudo

Durante questa fase, si vogliono collaudare alcuni componenti in relazione alle funzionalità aggiunte. In particolare, si vuole testare l'efficacia di Hologram Factory nel calcolare la pose di un ologramma rispetto a un marker di riferimento e nel definire la posizione e il gaze dell'utente rispetto all'origine, partendo da un sistema di riferimento differente.

Unity permette di definire *unit test* grazie al framework `NUnit` ed è possibile eseguirli grazie al tool `Test Runner`.

8.1.1 Verifica della computazione della pose degli ologrammi in un approccio multi-marker

Come abbiamo visto, ogni qualvolta venga riconosciuto un marker, Hologram Factory deve ricalcolare la posizione e la rotazione degli ologrammi in funzione del nuovo sistema di riferimento.

Date la traslazione e la rotazione di un marker secondario rispetto all'origine e la pose di un ologramma, possiamo verificare che Hologram Factory calcoli correttamente i nuovi valori, confrontando i risultati prodotti con quelli che ci si attende.

Nel frammento di codice proposto, si mostrano due esempi di test: il primo verifica che venga calcolata correttamente la nuova posizione dell'ologramma mentre il secondo constata che la nuova rotazione sia corretta.

Per ogni test vengono forniti i dati di input e il risultato atteso, confrontando quest'ultimo con il valore ottenuto dalla funzione che si vuole collaudare (in questo caso, *ComputeHologramPosition* e *ComputeHologramOrientation*).

```
namespace Tests
{
    public class MultiMarkerTests
    {
      [Test]
      public void TestMultiMarkerPosition()
      {
          Vector3 markerPos = new Vector3(40, 20, -30);
          Quaternion markerRot = Quaternion.Euler(0, 0, 90);
          Vector3 hologramPos = new Vector3(10, 15, -20);
          Vector3 expectedResult = new Vector3(-5, 30, 10);
          Assert.AreEqual(expectedResult,
              ComputeHologramPosition(markerPos, markerRot,
              hologramPos));
          [...]
      }

      [Test]
      public void TestMultiMarkerOrientation()
      {
          Quaternion markerRot = Quaternion.Euler(0, 0, 90);
          Quaternion hologramRot = Quaternion.Euler(0, 0, -90);
          Quaternion expectedResult = Quaternion.Euler(0, 0, 180);
          Assert.AreEqual(expectedResult,
              ComputeHologramOrientation(markerRot, hologramRot));
          [...]
      }
    }
}
```

Listato 8.1: Esempi di unit test per verificare il corretto calcolo della pose degli ologrammi

8.1.2 Verifica della computazione delle informazioni relative allo user avatar

A Hologram Factory viene delegato il compito di calcolare anche la posizione e il gaze dell'utente rispetto il marker di origine, dato un sistema di riferimento differente. Questa trasformazione risulta necessaria in quanto, lato server, le entità aumentate sono in relazione a un unico sistema di riferimento mentre nel mondo fisico esistono più punti di accesso.

Fornite la posizione dell'utente rispetto al marker di riferimento e la pose di detto marker rispetto all'origine, si verifica che il risultato dell'operazione *ComputeUserPosition* combaci con quello atteso.

In modo analogo si controlla che il ricalcolo del gaze, ottenuto da *ComputeUserGaze*, corrisponda al valore previsto.

```
namespace Tests
{
    public class MultiMarkerTests
    {
      [Test]
      public void TestUserPosition()
      {
          Vector3 markerPos = new Vector3(10, 5, 0);
          Quaternion markerRot = Quaternion.Euler(0, 0, 90);
          Vector3 userGaze = new Vector3(1, 3, 2);

          Vector3 expectedResult = new Vector3(7, 6, 2);
          Assert.AreEqual(expectedResult,
              ComputeUserPosition(markerPos, markerRot, userPos));

          [...]
      }

      [Test]
      public void TestUserGaze()
      {
          Vector3 markerPos = new Vector3(10, 5, 0);
          Quaternion markerRot = Quaternion.Euler(0, 0, 90);
          Vector3 userGaze = new Vector3(1, 11, 2);

          Vector3 expectedResult = new Vector3(-1, 6, 2);
          Assert.AreEqual(expectedResult, ComputeUserGaze(markerPos,
              markerRot, userGaze));
```

```
        [...]
    }
  }
}
```

Listato 8.2: Esempi di unit test per verificare il corretto calcolo della posizione e del gaze dell'utente rispetto all'origine del mondo aumentato

8.2 Validazione funzionale

Nel campo della realtà aumentata, è opportuno costruire degli esempi ad-hoc che permettano di validare le funzionalità sviluppate. Infatti, dovendo gestire elementi virtuali all'interno dell'ambiente, risulta necessario verificare che la loro rappresentazione sia coerente con quanto implementato.

Si procede, di conseguenza, a descrivere tre esempi di applicazione che, utilizzando il framework sviluppato, permettano di dimostrare la validità di un approccio multi-marker, la corretta interazione dell'utente con gli ologrammi nell'ambiente e il funzionamento di un approccio multi-user cooperativo.

8.2.1 Applicazione d'esempio per la validazione dell'approccio multi-marker

Il seguente esempio vuole verificare la corretta pose degli ologrammi, a seguito dell'inquadramento con il proprio dispositivo di marker differenti, dislocati in diversi punti dell'ambiente fisico. Si costruirà una user app tramite il framework sviluppato e si procederà ad inquadrare i diversi marker, verificando che l'aggiornamento delle informazioni sugli ologrammi sia corretto e rispecchi quanto previsto.

Contesto

Supponiamo di avere una scrivania, di dimensioni 90 cm di lunghezza, 50 cm di profondità e 85 cm di altezza, sulla quale sono posizionati tre marker, che differiscono sia per posizione che per rotazione. L'origine del mondo è data dal primo marker a sinistra mentre gli altri due sono in funzione del primo.

Sopra la scrivania, si vogliono posizionare una lampada e un robot virtuali, rispettivamente in una posizione (espressa in metri) di (0.2, 0.0, 0.3) e (0.1, 0.0, 0.2) rispetto all'origine.

Figura 8.1: Contesto in cui verrà eseguita la validazione funzionale dell'approccio multi-marker

Implementazione

Anzitutto, è necessario definire il file di configurazione che specifica la posizione e la rotazione dei diversi marker. A seguito della rilevazione delle misure nell'ambiente fisico, si produce il file JSON mostrato di seguito.

```
{
  "markers": [
  {
    "name": "flowers",
    "position": [0.0, 0.0, 0.0],
    "rotation": [0.0, 0.0, 0.0]
  },

  {
    "name": "dog",
    "position": [0.60, -0.04, -0.1],
    "rotation": [-90.0, 0.0, 0.0]
```

```
  },

  {
    "name": "earth",
    "position": [0.77, 0.33, 0.05],
    "rotation": [-90.0, 90.0, 0.0]
  }
 ]
}
```

Listato 8.3: File JSON di configurazione dei marker per il contesto definito

Successivamente, è opportuno definire lato server le entità aumentate che rappresenteranno la lampada e il robot, nonché gli agenti adibiti alla loro creazione ed eventuale tracking.

Le prime vengono definite come classi Java che estendono da AE, la cui annotazione @HOLOGRAM permette di specificare alcune proprietà, tra cui il nome della geometria che dovrà essere utilizzata.

```
@HOLOGRAM(geometry = "Lamp")
public class Lamp extends AE {
  [...]
}

@HOLOGRAM(geometry = "Robot")
public class Robot extends AE {
  [...]
}
```

Listato 8.4: Definizione delle entità aumentate per la lampada e il robot

Gli agenti, invece, vengono definiti come file .asl e si occuperanno di gestire le entità aumentate. Per l'esempio proposto, è stato definito un *lampAgent* che si occuperà della creazione della lampada e un *robotAgent* che, analogamente, procederà a generare un robot.

```
//Lamp agent
+!start
    <- .print("=== LAMP AGENT ===");
   makeArtifact("woatClient-lampAgent", "mirage4agents.WoatClient",
       ["127.0.0.1"], WoatClient);
   focus(WoatClient);
   lookupArtifact("woatTools", _);
   joinAW("testMultiMarker").
```

```
+join_done("testMultiMarker")
    <- !createLamp("lamp001").

+!createLamp(Name)
    <-   cartesianLocation(0.20, 0.0, 0.30, L);
   angularOrientation(-90, 0, 0, 0);
   createAE("testMultiMarker", Name, "Lamp", L, 0).

//Robot agent
+!start
    <-   .print("=== ROBOT AGENT ===");
   makeArtifact("woatClient-robotAgent", "mirage4agents.WoatClient",
       ["127.0.0.1"], WoatClient);
   focus(WoatClient);
   lookupArtifact("woatTools", _);
   joinAW("testMultiMarker").

+join_done("testMultiMarker")
    <- !createRobot("robot001").

+!createRobot(Name)
    <- cartesianLocation(0.10, 0.0, 0.20, L);
   angularOrientation(0, 0, 0, 0);
   createAE("testMultiMarker", Name, "Robot", L, 0).
```

Listato 8.5: Definizione di due agenti, rispettivamente adibiti alla gestione della lampada e del robot

Lato client, è necessario definire le geometrie degli ologrammi come prefab, che verranno successivamente caricate dalla user app al momento della rappresentazione.

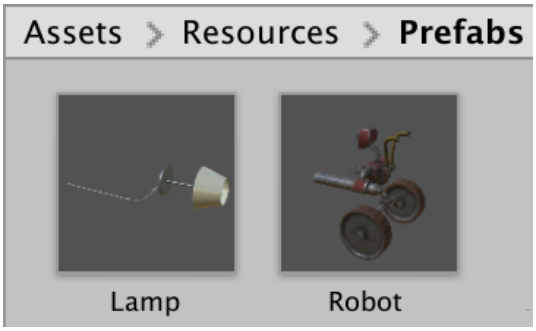

Figura 8.2: Prefabs che definiscono le geometrie per gli ologrammi

Risultato

Durante l'esecuzione della user app, è stata inserita nella schermata del device una label che specificasse il marker corrente selezionato, al fine di fornire un risultato attendibile.

Si evidenzia che la rappresentazione degli ologrammi, in termini di posizione e rotazione, risulta corretta e avviene dove previsto. Tale pose è tanto più precisa quanto più vicino è il marker che è stato selezionato. Tuttavia, anche in marker più distanti, il discostamento degli ologrammi è trascurabile.

Figura 8.3: Rappresentazione degli ologrammi attraverso diversi marker: a sinistra, il marker di origine; al centro, il marker in posizione (0.77, 0.33, 0.05); a destra, il marker in posizione (0.60, -0.04, -0.1)

Effettuando delle misurazioni relative alla latenza nella visualizzazione degli ologrammi, si nota quanto segue:

- La prima visualizzazione degli ologrammi comporta una latenza di circa 850 - 900 ms. Questo è dovuto al tempo di login dell'utente all'interno del mondo, che deve essere registrato e deve essergli assegnato un id.

- Le successive rappresentazioni, invece, sono dell'ordine di pochi ms, in quanto l'utente è già registrato all'interno del mondo ed è necessario, eventualmente, solo un cambiamento di ancora dipendente dal nuovo marker inquadrato

8.2.2 Applicazione d'esempio per la validazione della user interaction

In questo esempio si vuole verificare l'efficacia della gestione dello user avatar, mostrando l'interazione dell'utente con gli ologrammi nell'ambiente.

Contesto

Supponiamo che il contesto ambientale sia lo stesso dell'esempio precedente, ovvero una scrivania di dimensioni 90 cm di lunghezza, 50 cm di profondità e 85 cm di altezza, sulla quale sono posizionati tre marker che differiscono per posizione e rotazione. Su tale scrivania, inquadrando un qualsiasi marker, si vogliono mostrare un cubo e una sfera, inizialmente di colore giallo e rispettivamente nelle posizioni (0.10, 0.0, 0.20) e (0.40, 0.0, 0.20) rispetto al marker di origine. L'utente, avvicinandosi a ognuno di questi oggetti, vedrà il loro colore modificarsi in verde; allontanandosi nuovamente, il colore tornerà giallo.

Implementazione

Si mantiene lo stesso file di configurazione dei marker dell'esempio precedente e, successivamente, si creano le entità aumentate che definiranno il cubo e la sfera. Ognuna di esse avrà la proprietà *color* che definirà il colore corrente.

```
@HOLOGRAM(geometry = "Cube")
public class Cube extends AE {

  @PROPERTY private String color = "yellow";

}

@HOLOGRAM(geometry = "Sphere")
public class Sphere extends AE {

  @PROPERTY private String color = "yellow";

}
```

Listato 8.6: Implementazione delle entità aumentate cubo e sfera

In seguito, si modellano gli agenti che agiranno all'interno del mondo. Per questo esempio, sono stati definiti due agenti, *cubeAgent* e *sphereAgent*, che si occuperanno di creare la rispettiva entità aumentata e definire una regione su di essa, in modo tale da verificare eventi che accadono nella porzione di spazio definita (ad esempio, entrata e uscita di un utente). Se l'utente si trova

nella regione, l'agente modificherà il colore dell'entità che sta monitorando; analogamente, quando l'utente uscirà dalla regione, il colore originario verrà ripristinato.

```
[...]

+join_done("userInteractionExample")
    <- !createCube("cube001");
   !createRegion("region001").

+!createCube(Name)
    <- cartesianLocation(0.10, 0.0, 0.20, L);
   angularOrientation(0, 0, 0, 0);
   createAE("userInteractionExample", Name, "Cube", L, 0).

+!createRegion(Name)
    <-  cartesianLocation(0.10, 0.0, 0.20, Position);
   circularCartesianArea(Position, 0.2, Area);
   defineRegion("userInteractionExample", Name, Area).

+region_defined("region001")
    <-  trackRegion("userInteractionExample", "region001");

+region_entities_update("region001", L)
   <- !checkEntityInRegion(L, "user").


+!checkEntityInRegion([H|T], E)
   <- if (.length([H|T]) <= 1) {
       updatePropertyOnAE("userInteractionExample", "cube001",
          "color", "yellow");
     } else {
        if (.substring(E,H)) {
          updatePropertyOnAE("userInteractionExample", "cube001",
             "color", "green");
        } else {
          !checkEntityInRegion(T, E);
        }
     }.
```

Listato 8.7: Definizione dell'agente adibito alla gestione del cubo

Lato client, si definiscono le geometrie degli ologrammi e il relativo controller per la proprietà custom *color*. Nello specifico, è stato sviluppato un

ColorController, associato a ogni ologramma, che ricevuto l'aggiornamento da parte del server, provvede a effettuare le modifiche.

```
public override void onCustomPropertyUpdateReceived(string property,
    object value)
{
    switch (property)
    {
        case COLOR_PROPERTY:
            manageColorPropertyUpdates(value);
            break;

        default:
           break;
    }
}

private void manageColorPropertyUpdates(object value)
{
    string color = value.ToString();

    switch(color)
    {
        case "yellow":
            gameObject.transform.GetComponent<Renderer>()
                .material.color = Color.yellow;
            break;

        case "green":
           gameObject.transform.GetComponent<Renderer>()
               .material.color = Color.green;
           break;

        default:
           break;
    }
}
```

Listato 8.8: Implementazione di una estensione della classe *HologramController* per la gestione della proprietà custom *color*

Risultati

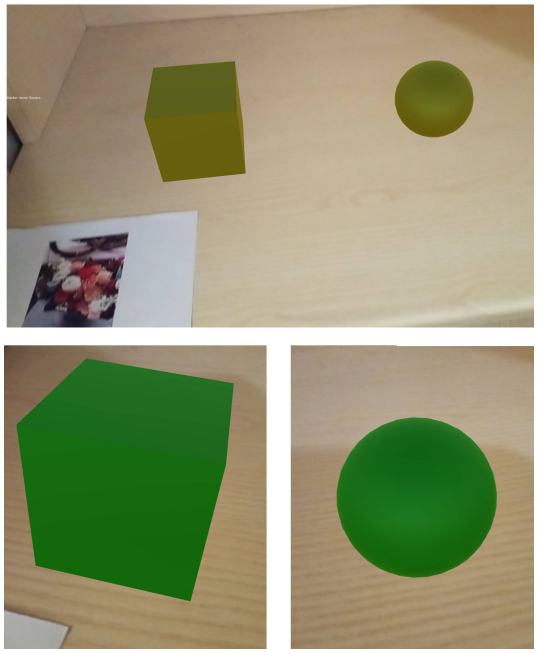

Figura 8.4: Nella parte superiore, stato iniziale dell'esempio; nella parte inferiore, il risultato che si ottiene quando si entra nella regione di ogni specifico ologramma

Eseguendo l'applicazione, il comportamento degli ologrammi ha rispecchiato quanto previsto, cambiando il loro colore in verde entrando nella loro regione e acquisendo nuovamente il colore giallo allontanandoci da essi. Non si sono evidenziati lag durante il cambiamento di stato ed, effettuando diverse misurazioni sul tempo che intercorre tra l'evento che rileva l'utente all'interno della regione lato server e il completamento dell'aggiornamento dell'ologramma da parte del client, si rileva una media di 90 ms, tempo trascurabile da parte dell'utente.

8.2.3 Applicazione d'esempio per la validazione di un approccio multi-user cooperativo

Lo sviluppo di questo esempio mira a verificare che i cambiamenti che avvengono negli ologrammi debbano essere percepiti da tutti gli utenti connessi al mondo. Infatti, l'interazione che un utente ha su un oggetto può comportare delle modifiche che devono essere visibili da un possibile spettatore, che sta eseguendo sul proprio la dispositivo la stessa user app.

Contesto

Ipotizziamo un contesto molto simile al precedente, in cui abbiamo un tavolo con un marker su di esso. Inquadrando detto marker, vengono visualizzati un cubo e una sfera di colore giallo. Si vuole verificare che, se un utente si avvicina a un ologramma, modificando il colore di quest'ultimo in verde, questo aggiornamento sia visibile anche al secondo partecipante.

Implementazione

Manteniamo l'implementazione identica all'esempio precedente. Le entità aumentate sono definite da due classi che estendono da AE e rappresentano la sfera e il cubo. Per ognuno di essi, un agente definisce una regione e, qualora sia presente un utente al suo interno, effettua l'aggiornamento della proprietà dell'entità che rappresenta il colore. Questo cambiamento viene successivamente propagato al controller lato client, che effettua l'update sull'ologramma.

Risultato

Figura 8.5: A sinistra, situazione iniziale con due dispositivi che vedono gli ologrammi di colore giallo; a destra, un dispositivo è entrato nella regione del cubo e il cambiamento di colore è visibile al secondo partecipante

Come visibile dalla figura, notiamo che il caso iniziale è di due partecipanti che sono esterni alle regioni degli ologrammi e, conseguentemente, il colore di quest'ultimi è giallo. Quando un utente si avvicina con il proprio dispositivo all'ologramma, entrando nella sua regione, questo diventa di colore verde e il cambiamento è visibile anche al secondo partecipante.

Al fine di rilevare alcune misurazioni, sono stati disposti due timer, eseguiti in contemporanea durante la situazione iniziale dell'applicazione e terminati nel momento in cui un dispositivo entrasse nella regione di un ologramma e percepisse il cambiamento di colore. Benché il test risenta della reattività dei partecipanti nell'eseguirlo all'unisono e terminarlo nell'istante in cui avvenisse l'aggiornamento dell'ologramma, si denota che il cambiamento di colore dell'ologramma avviene per un dispositivo all'incirca 50 - 70 ms dopo rispetto al secondo, ritardo trascurabile da parte dell'utente.

8.3 Caso di studio: Rocca delle Caminate

Come accennato, si vuole sfruttare un contesto concreto nel quale utilizzare e validare il framework prodotto. In questo paragrafo si procede a descrivere il progetto sviluppato per Rocca delle Caminate, atto a integrare gli spazi di quest'ultima con elementi di augmented e mixed reality

8.3.1 Contesto

Rocca delle Caminate[1] è un fortilizio situato su una collina del comune di Meldola, a 4 km di distanza da Predappio. È circondata da un ampio cortile esterno, nel quale una rampa inclinata permette l'accesso alla fortezza. Entrandoci, un visitatore si trova all'interno di un grande spazio all'aperto, chiuso dalle possenti mura da un lato e da un'ala del castello dall'altra, nel cui sfondo si erge il grande torrione. Questa area prende il nome di corte interna.

Si vuole sfruttare questa area per inserire elementi virtuali di carattere medievale. Nello specifico, si vuole realizzare un'applicazione per dispositivi mobili che permetta, una volta eseguita, di avere accesso a un *augmented world*, contenente un insieme di entità sia statiche che dinamiche, con alcune delle quali sia possibile l'interazione con l'utente.

[1] `http://www.roccadellecaminate.com`

Figura 8.6: Rocca delle Caminate: corte interna

8.3.2 Implementazione

In precedenza, è stata sviluppata un'applicazione che permetteva la virtualizzazione all'interno della rocca di un insieme di elementi medievali prettamente statici (una tenda, un carro, un cavaliere e un fabbro), nel quale l'utente aveva solamente la funzione di spettatore esterno. Non era prevista, di conseguenza, né interazione con le entità né cooperazione tra gli utenti.

Partendo da questa base, si vuole integrare l'applicazione inserendo il framework sviluppato, abilitandola ai mondi aumentati. Questo permette di realizzare un contesto multi-user, in cui gli utenti vedono gli stessi ologrammi e i relativi cambiamenti, nonché la possibilità di interagire con essi.

Agli elementi già presenti, vengono aggiunti i seguenti:

- **Una lanterna:** inserita all'interno della tenda, che si accende avvicinandosi a essa e, viceversa, si spegne allontanandosi
- **Un quadro dinamico:** posizionato su una parete della rocca, che mostra a intervalli di tempo definiti diverse immagini
- **Un forziere:** che si apre se un utente si avvicina a esso. Se un forziere è stato già aperto in precedenza, un successivo utente lo vedrà già aperto.
- **Un cavaliere:** che saluta l'utente che lo sta osservando. Si vuole, quindi, verificare l'efficacia del gaze dell'utente quando è diretto verso elementi della scena, che possono interagire.

Implementazione della lanterna

Definiamo, anzitutto, la lanterna come classe che estende da AE, definendo la proprietà *status* che può assumere i valori *on* e *off*.

```
@HOLOGRAM(geometry = "Lantern")
public class Lantern extends AE {

  @PROPERTY private String status = "off";

}
```

Listato 8.9: Definizione dell'entità aumentata rappresentante la lanterna

Definiamo, inoltre, un agente adibito al controllo dell'entità aumentata rappresentante la lanterna. Esso, dapprima, si occuperà di creare la lanterna e definire una regione in una specifica posizione (*+!createLantern* e *+!createRegion* nel codice). Successivamente, terrà traccia della regione per verificare la presenza o meno di un utente. In caso affermativo, procede a cambiare lo stato della lanterna (*updatePropertyOnAE* nel codice).

```
[...]

+!createLantern(Name)
   <- cartesianLocation(10.80, 0.80, -4.98, L);
      angularOrientation(0, 0, 0, 0);
      createAE("augmentedFortress", Name, "Lantern", L, 0).

+!createRegion(Name)
   <- cartesianLocation(10.80, 0.80, -4.98, Position);
      circularCartesianArea(Position, 0.7, Area);
      defineRegion("augmentedFortress", Name, Area).

+ae_created("lantern001")
   <- trackAE("augmentedFortress", "lantern001").

+region_defined("region001")
   <- trackRegion("augmentedFortress", "region001").

+region_entities_update("region001", L)
   <- !checkEntityInRegion(L, "user").

 +!checkEntityInRegion([H|T], E)
   <- if (.length([H|T]) <= 1) {
```

```
      updatePropertyOnAE("augmentedFortress", "lantern001",
         "status", "off");
     } else {
        if (.substring(E, H)) {
           updatePropertyOnAE("augmentedFortress", "lantern001",
              "status", "on");
        } else {
           !checkEntityInRegion(T, E);
        }
     }.
```

Listato 8.10: Frammento di codice relativo all'agente che si occupa di gestire la lanterna e la regione attorno a essa

A livello di client, è necessario definire la geometria della lanterna come prefab e, inoltre, definire il relativo *HologramController*. In esso si gestisce il cambiamento di stato a livello di ologramma. In questo caso si accede allo *SpriteRenderer* che definisce la fiamma all'interno della lanterna e si abilita o meno a seconda del valore ricevuto dal server.

```
public class LanternController : HologramController
{
    [...]
    public override void onCustomPropertyUpdateReceived(string
        property, object value)
    {
        switch(property)
        {
            case "status":
                UpdateLanternStatus(value.ToString());
                break;

            default:
                break;
        }
    }

    private void UpdateLanternStatus(string value)
    {
        switch(value)
        {
            case ON:
                this.gameObject.transform.GetChild(FIRE)
                  .GetComponent<SpriteRenderer>().enabled = true;
```

```
                break;

            case OFF:
                this.gameObject.transform.GetChild(FIRE)
                  .GetComponent<SpriteRenderer>().enabled = false;
                break;

            default:
                break;
        }
    }
}
```

Listato 8.11: Implementazione di LanternController

Il seguente diagramma di sequenza riassume le principali operazioni eseguite e i componenti che vi partecipano. Notiamo come l'agente, dopo aver creato e tenuto traccia della lanterna e della regione, rimane in ascolto di eventi che informano se sia presente o meno un utente nell'area definita. A seconda di questi, viene eseguito *updatePropertyOnAE* che effettua l'aggiornamento della proprietà, che viene notificata al relativo *Hologram Controller* lato client.

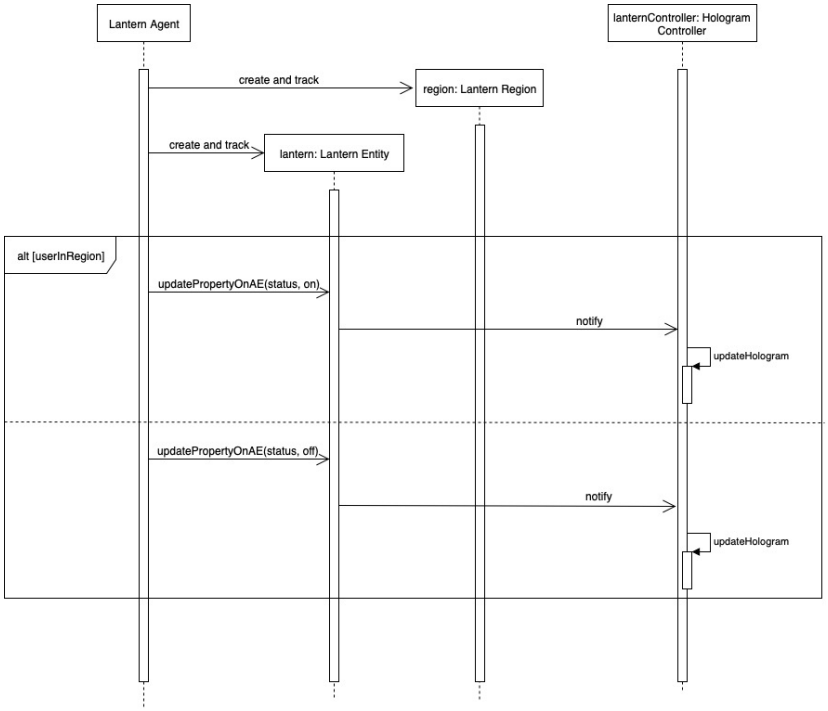

Figura 8.7: Diagramma di sequenza che mostra i principali componenti e azioni nella gestione della lanterna

Implementazione del quadro dinamico

Il quadro, posizionato su una parete, dovrà mostrare diverse immagini. Ogni utente percepirà la medesima immagine e ogni aggiornamento della stessa.

Come in precedenza, definiamo la relativa entità aumentata. Essa avrà una proprietà che definirà l'attuale id dell'immagine che si sta mostrando. Inoltre, il metodo *changePicture* permetterà ogni tre secondi di modificarla con una nuova.

```
@HOLOGRAM(geometry = "Frame")
public class Frame extends AE {
   @PROPERTY private String pictureId = "rocca1";

   private volatile boolean stop = false;
   private String[] pictures = {"rocca1", "rocca2", "rocca3"};

   @ACTION public void changePicture() {
      new Thread( () -> {
         while(!stop) {
            try {
               for(String id : pictures) {
                  customProperty("pictureId", id);
                  Thread.sleep(3000);
               }
            } catch (Exception e) { }
         }
      }).start();
   }
}
```

Listato 8.12: Definizione dell'entità aumentata rappresentante il quadro

Anche in questo caso, avremo un agente che si occuperà di gestire il quadro, creandolo ed eseguendo l'azione che permette di visualizzare dinamicamente le immagini.

```
[...]

+!createFrame(Name)
   <- cartesianLocation(0.98, 2.10, 2.10, L);
      angularOrientation(0, 95, 0, 0);
      createAE("augmentedFortress", Name, "Frame", L, 0).

+ae_created("frame001")
```

```
   <- println("entity frame001 created!");
      trackAE("augmentedFortress", "frame001").

+ae_tracking_begin("frame001")
   <- println("Tracking started for entity frame001");
      !!changePicture("frame001").

+!changePicture(Frame)
  <- executeActionOnAE("augmentedFortress", "frame001",
      "changePicture").
```

Listato 8.13: Codice relativo all'agente che si occupa di gestire il quadro

Come in precedenza, anche per l'ologramma rappresentante il quadro avremo il rispettivo *HologramController*, che si occuperà di mostrare l'immagine corrispondente all'id ricevuto dal server. Nello specifico, si modifica il campo *mainTexture* del componente *material*, caricando la nuova immagine associata a quell'id.

```
public class PictureFrameController : HologramController
{
    public override void onCustomPropertyUpdateReceived(string
        property, object value)
    {
        ChangePicture(value.ToString());
    }

    private void ChangePicture(string id)
    {
        foreach (Material m in
            this.gameObject.GetComponent<Renderer>().materials)
        {
            if (m.name.Contains("picture"))
            {
                m.mainTexture =
                    Resources.Load<Texture2D>(TEXTURES_PATH + id);
            }
        }
    }
}
```

Listato 8.14: Implementazione di PictureFrameController

Possiamo visualizzare un riassunto delle interazioni nel seguente diagramma di sequenza. L'agente crea l'entità *Frame* ed esegue l'azione di cambiamento

di immagine. A ogni cambiamento corrisponde l'aggiornamento dell'id della figura che si intende visualizzare e, successivamente, tale valore viene notificato al controller lato client, che procede a effettuare l'update sull'ologramma.

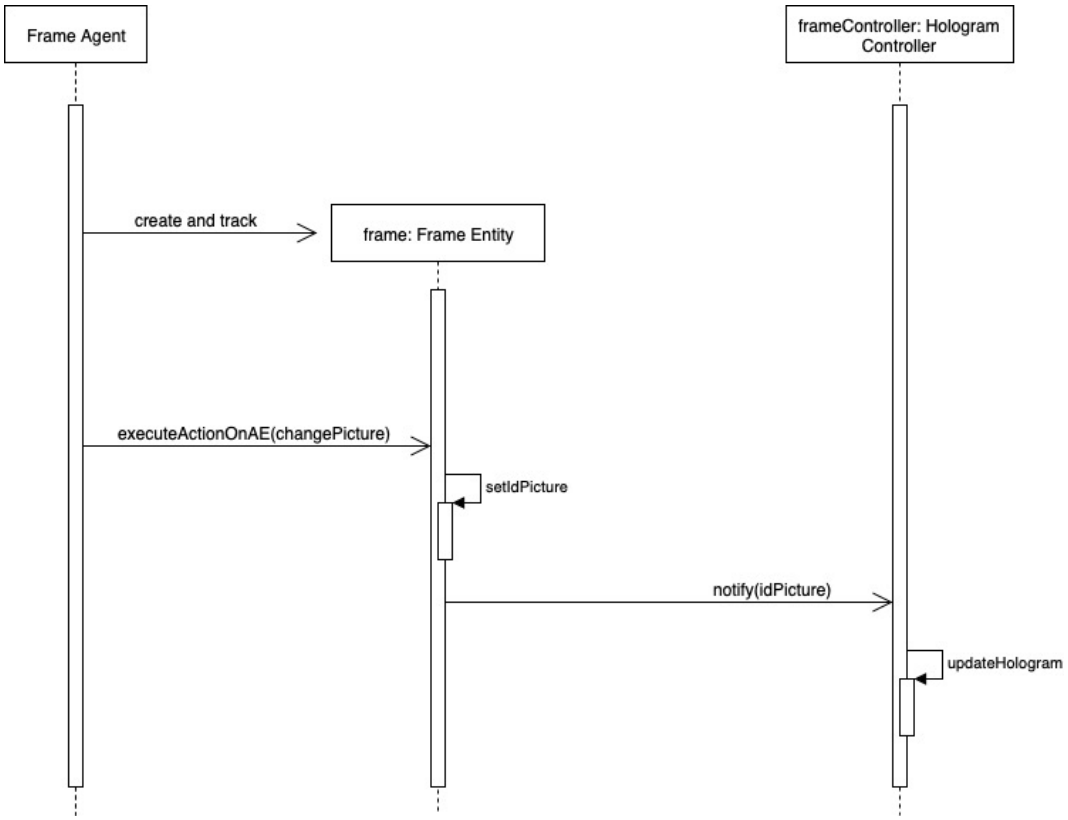

Figura 8.8: Diagramma di sequenza che mostra i principali componenti e azioni nella gestione del quadro

Implementazione del forziere

Analogamente alla lanterna, definiamo una classe che estende da AE e definisce uno stato, che può assumere i valori di *opened* e *closed*, rispettivamente se il forziere è aperto o chiuso. Definiamo, inoltre, un metodo *openTreasureBox* che permette di aprire un forziere chiuso.

```
@HOLOGRAM(geometry = "TreasureBox")
public class TreasureBox extends AE {
   [...]
   @PROPERTY private String status= "closed";

   @ACTION public void openTreasureBox() {
```

```
      if (status == CLOSED) {
         try {
            customProperty("status", OPENED);
         } catch (PropertyNotFoundException e) { }
      }
   }
}
```

Listato 8.15: Definizione dell'entità aumentata rappresentante il forziere

In seguito, implementiamo l'agente che si occuperà dapprima di creare il forziere e, successivamente, definire la relativa regione per verificare la presenza o meno dell'utente. Qualora venga rilevato un utente nelle vicinanze del forziere, verrà eseguita l'azione che permette di aprirlo.

```
[...]

+!createTreasureBox(Name)
   <- cartesianLocation(2.04, -0.2, 2.9, L);
      angularOrientation(0, 140, 0, O);
      createAE("augmentedFortress", Name, "TreasureBox", L, O).

+!createRegion(Name)
   <- cartesianLocation(2.04, -0.2, 2.9, Position);
      circularCartesianArea(Position, 1.4, Area);
      defineRegion("augmentedFortress", Name, Area).

+ae_created("treasurebox")
   <- trackAE("augmentedFortress", "treasurebox").


+region_defined("region002")
   <- trackRegion("augmentedFortress", "region002").

+region_entities_update("region002", L)
   <- !checkEntityInRegion(L, "user").

 +!checkEntityInRegion([H|T], E)
   <- if (.length([H|T]) > 1) {
        if (.substring(E,H)) {
           executeActionOnAE("augmentedFortress", "treasurebox",
              "openTreasureBox");
        } else {
           !checkEntityInRegion(T, E);
```

```
        }
    }.
```

Listato 8.16: Implementazione dell'agente che si occupa di gestire il forziere

Il client verificherà il nuovo valore di stato del forziere tramite l'opportuno controller e, in caso di *opened*, provvederà a recuperare il componente *Animator* e lo abiliterà.

```
public class TreasureBoxController : HologramController
{
    [...]
    public override void onCustomPropertyUpdateReceived(string
        property, object value)
    {
        switch(property)
        {
            case "status":
                UpdateTreasureBoxStatus(value.ToString());
                break;

            default:
                break;
        }
    }

    private void UpdateTreasureBoxStatus(string value)
    {
        if (value.Equals(OPENED))
        {
           this.gameObject.GetComponent<Animator>().enabled = true;
        }
    }
}
```

Listato 8.17: Implementazione di TreasureBoxController

L'insieme delle interazioni tra i componenti e le relative operazioni sono visibili nel diagramma di sequenza seguente nel quale si denota che un agente, creati e tenuto traccia del forziere e della regione, rimane in attesa di un evento che attesti la presenza dell'utente all'interno dell'area definita. In caso affermativo, procede a eseguire l'azione dell'entità tramite *executeActionOnAE* che provvederà ad aprire il forziere, qualora non lo fosse già. Infine, viene notificato l'aggiornamento al controller dell'ologramma, che effettuerà anch'esso l'update.

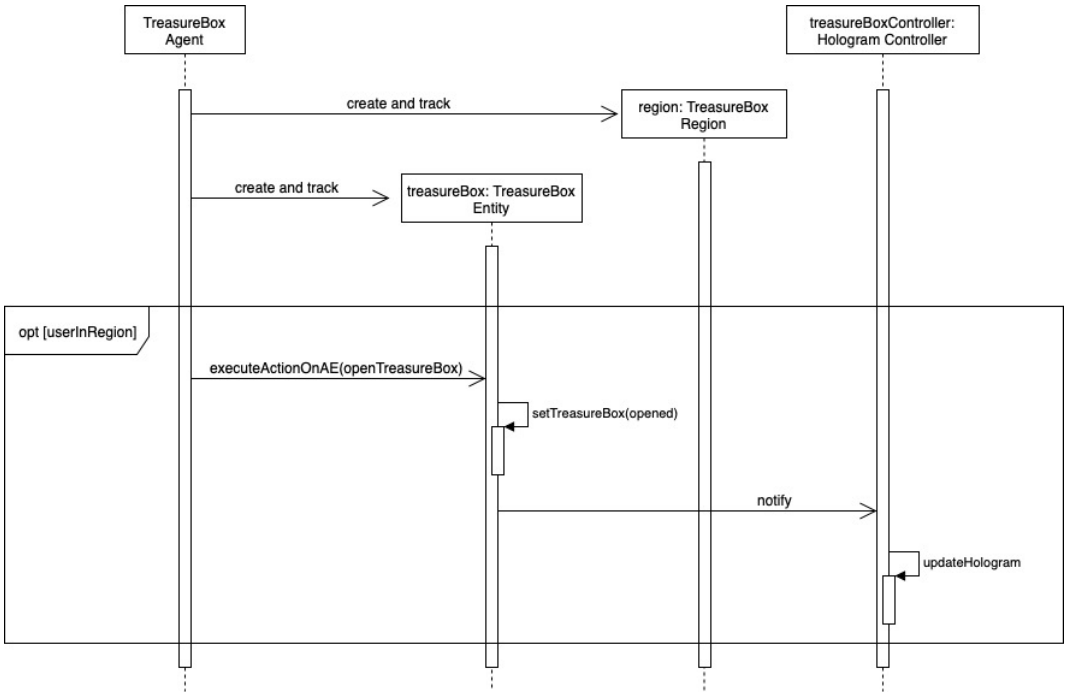

Figura 8.9: Diagramma di sequenza che mostra i principali componenti e azioni nella gestione del forziere

Implementazione del cavaliere

Si vuole definire un cavaliere che reagisca allo sguardo dell'utente. Nello specifico, se l'utente lo osserva, il cavaliere lo saluterà. Definiamo un'entità aumentata *Knight* con una proprietà che espliciterà il suo stato: *idle*, se non sta eseguendo alcuna azione oppure *sayingHello*, se sta interagendo con l'utente. Inoltre, definiamo l'azione *sayHello*, che permetterà di effettuare il cambiamento di stato dell'entità, qualora la posizione dell'utente sia di fronte a quella del cavaliere e il gaze sia rivolto verso quest'ultimo.

```
@HOLOGRAM(geometry = "Knight")
public class Knight extends AE {
   [...]
   @PROPERTY private volatile String status = "idle";

   @ACTION public void sayHello(Object pos, Object gaze) {
      [...]

      if (knightPos.equals(userGaze) && userPos.z() < knightPos.z())
      {
         try {
```

```
            if (status.equals(IDLE)) {
                new Thread(new StatusUpdater()).start();
            }
        } catch (PropertyNotFoundException e) {}
    }

    } catch (AugmentedEntityNotFoundException e) {}
  }

  class StatusUpdater implements Runnable {

    @Override
    public void run() {
      try {
        customProperty("status", SAYING_HELLO);
        Thread.sleep(8000);
        customProperty("status", IDLE);
      } catch (PropertyNotFoundException | InterruptedException e)
          {
        e.printStackTrace();
      }
    }

  }
}
```

Listato 8.18: Definizione dell'entità aumentata rappresentante il cavaliere

Implementiamo, inoltre, l'agente che, dapprima, si occuperà di creare l'entità aumentata e la relativa regione. Ogni qualvolta avvenga un aggiornamento sul numero di entità all'interno di essa, si verifica se è presente un utente. In caso affermativo, l'agente comincerà il *tracking*, al fine di ricevere aggiornamenti sulla posizione e gaze dell'utente. Ottenute queste informazioni, procede a eseguire l'azione *sayHello* dell'entità aumentata.

```
[...]

+ae_created("knight001")
  <- trackAE("augmentedFortress", "knight001").

+region_defined("region003")
  <- trackRegion("augmentedFortress", "region003").

+region_entities_update("region003", L)
```

```
   <- !checkEntityInRegion(L, "user").

+ae_property_update(E, "gaze", V) : user_in_region("yes")
 <- -+user_gaze(V).

+ae_property_update(E, "location", L) : user_gaze(G)
   <- executeActionOnAE("augmentedFortress", "knight001",
       "sayHello", L, G);
        --user_gaze(G).

+!createKnight(Name)
   <- cartesianLocation(1.32, -0.39, 2.89, L);
      angularOrientation(0, 165, 0, O);
      createAE("augmentedFortress", Name, "Knight", L, O).

+!createRegion(Name)
   <- cartesianLocation(1.32, -0.39, 2.89, Position);
      circularCartesianArea(Position, 3, Area);
      defineRegion("augmentedFortress", Name, Area).

 +!checkEntityInRegion([H|T], E)
   <- if (.substring(E,H)) {
         -+user_in_region("yes");
         trackAE("augmentedFortress", H);
      } else {
         !checkEntityInRegion(T, E);
      }.

+!checkEntityInRegion([], E)
   <- -+user_in_region("no").
```

Listato 8.19: Implementazione dell'agente che si occupa di gestire il cavaliere

A livello di client, in base allo stato ricevuto, il controller definito avvierà o meno l'animazione corrispondente al saluto.

```
public class KnightController : HologramController
{

    public override void onCustomPropertyUpdateReceived(string
        property, object value)
    {
        [...]
        switch (property)
```

```
        {
            case "status":
                UpdateKnightStatus(value.ToString());
                break;

            default:
                break;
        }
    }

    private void UpdateKnightStatus(string value)
    {
        if (value.Equals(SAYING_HELLO))
        {
            if (!gameObject.GetComponent<Animator>()
            .runtimeAnimatorController.name.Equals(SAYING_HELLO))
            {
                gameObject.GetComponent<Animator>()
                .runtimeAnimatorController =
                    Resources.Load(HELLO_ANIMATION_PATH) as
                    RuntimeAnimatorController;
            }
        }
        else
        {
            if (!gameObject.GetComponent<Animator>()
            .runtimeAnimatorController.name.Equals(IDLE))
            {
               gameObject.GetComponent<Animator>()
               .runtimeAnimatorController =
                   Resources.Load(IDLE_ANIMATION_PATH) as
                   RuntimeAnimatorController;
            }
        }
    }
}
```

Listato 8.20: Implementazione di KnightController

Nel seguente diagramma di sequenza possiamo vedere una panoramica dei principali componenti e operazioni durante il processo di aggiornamento della proprietà del cavaliere. Notiamo che l'agente dapprima crea e tiene traccia dell'entità aumentata, rappresentante il cavaliere e della regione. Quando un utente entra nella regione, effettua il *tracking* e attende aggiornamenti di po-

sizione e gaze. Ottenuti questi valori, viene richiamato il metodo *sayHello* dell'entità: se l'utente è effettivamente di fronte al cavaliere e lo sta guardando, viene eseguito il thread *Status Updater*, che provvede ad aggiornare lo stato dell'entità aumentata prima in *sayingHello* e, dopo un tempo T, in *idle*. I cambiamenti vengono propagati al controller lato client, che esegue le corrispondenti animazioni.

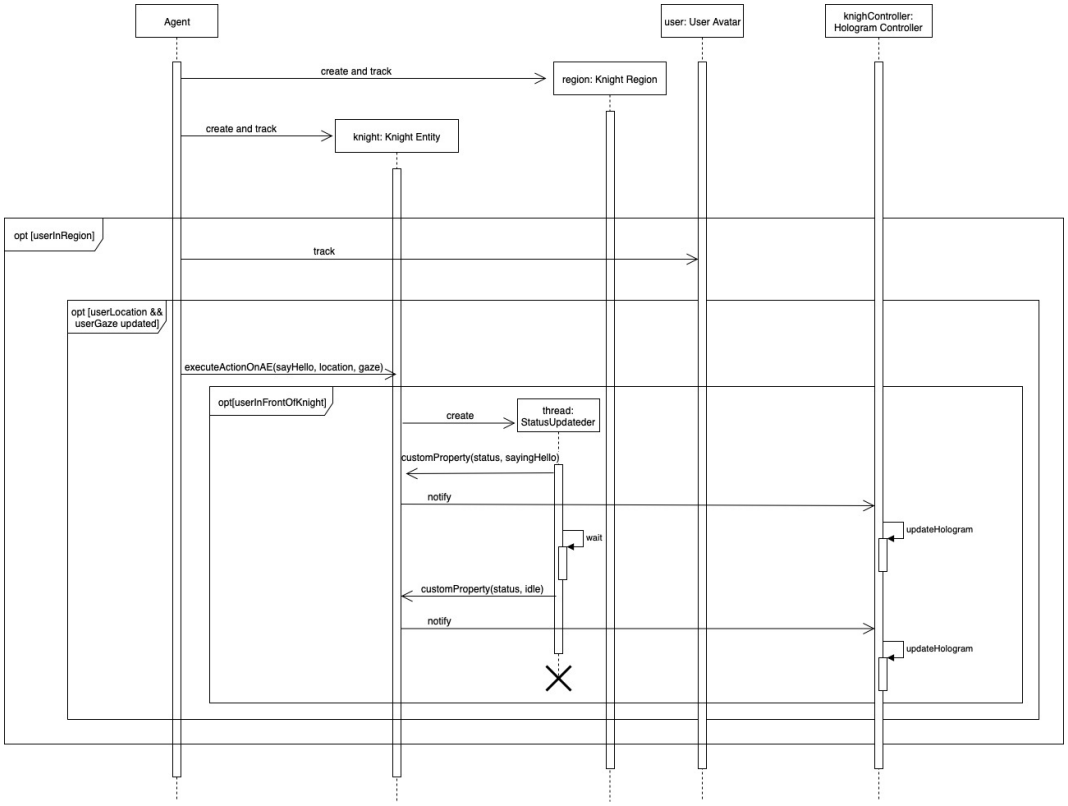

Figura 8.10: Diagramma di sequenza che riassume i principali componenti e azioni nella gestione del cavaliere e del gaze dell'utente

8.3.3 Risultato ottenuto

L'utilizzo del framework nel contesto di Rocca delle Caminate ha ulteriormente dimostrato la semplicità con cui è possibile definire nuove entità all'interno del mondo aumentato. Infatti, focalizzandosi sul modello degli elementi e sui relativi agenti, è stato possibile aggiungere agilmente gli ologrammi sopra descritti. L'architettura, benché necessiti di qualche aggiornamento per migliorarne le performance, ha risposto alle esigenze richieste, permettendo di immergersi in un contesto di mixed reality cooperativo, abilitando l'interazione con alcuni elementi virtuali.

Dopo aver inquadrato il marker di riferimento, si nota subito sopra di esso il quadro che, a intervalli di tempo, modifica l'immagine visibile a tutti i partecipanti. Il test ha permesso di verificare la gestione di elementi dinamici della scena, in particolare il controllo dell'agente su una o più entità e la propagazione dei cambiamenti al relativo *hologram controller*. Inquadrando con dispositivi diversi lo stesso quadro, si nota che il cambiamento avviene in simultanea, senza criticità o problemi di latenza. La posizione del quadro corrisponde a quella prevista, anche se si può notare una leggera variazione, in termini di qualche cm, all'aumentare della distanza dalla quale si osserva la scena. Per evitare o ridurre questa casistica, è importante permettere ad ARCore di analizzare e comprendere correttamente l'ambiente, individuando i feature point necessari a migliorare l'ancoraggio degli elementi virtuali.

Figura 8.11: Ologramma del quadro dinamico: a sinistra viene mostrata la prima immagine, a destra è avvenuto il cambiamento

Spostandoci verso sinistra avremo la rappresentazione di una tenda, all'interno della quale è presente la lanterna. Il comportamento di questa rispecchia quanto previsto: avvicinandosi, essa si accende; viceversa, si spegne. Con questo test si è voluto verificare anzitutto che l'utente venisse riconosciuto all'interno della regione definita dall'agente. Infatti, come abbiamo visto nella fase di implementazione, Hologram Engine lato client si occupa di calcolare la posizione dell'utente rispetto a un punto di origine e, successivamente, la invia alla controparte server affinché il relativo user avatar possa essere aggiornato. Se la posizione è inclusa nel range definito dalla regione, si verifica che l'a-

gente, che tiene traccia della suddetta, viene notificato correttamente. Si può confermare che il calcolo della posizione avviene in modo corretto e, anche in questo caso, il riconoscimento dell'utente nella regione avviene senza evidenti criticità, con una latenza dell'ordine di qualche ms.

Figura 8.12: Ologramma della lanterna: avvicinandosi si accende la fiamma

Un comportamento analogo avviene con il forziere, posto subito a destra dal marker di riferimento. Avvicinandosi, esso si apre e tutti i partecipanti lo vedranno aperto. Anche in questo caso, si è voluto verificare anzitutto che il riconoscimento dell'utente nella regione fosse corretto, ovvero che la posizione dell'utente rispetto al marker fossa calcolata senza errori e, conseguentemente, l'agente adibito al controllo di quella regione fosse notificato. Inoltre, si è potuto verificare che un secondo utente, spettatore della scena all'esterno della regione, potesse percepire l'utente interagire con l'ologramma. Lo spettatore ha visualizzato correttamente l'utente avvicinarsi al forziere e l'animazione

corrispondente, senza avvertire lag durante il cambiamento di stato, che è stato percepito all'unisono dagli utenti. Poiché, in questo caso, il forziere rimane aperto dopo l'interazione, si è potuto verificare che gli utenti, entranti nel mondo successivamente al cambiamento di stato dell'ologramma, vedessero quest'ultimo aggiornamento. Anche in questo caso, l'esito è stato positivo e gli utenti, acceduti al mondo successivamente, hanno percepito il forziere già aperto.

Figura 8.13: Ologramma del forziere e cambiamento di stato avvicinandosi a esso

Oltre al riconoscimento dell'utente in una regione definita, la *user interaction* può avvenire anche tramite *gaze*, ovvero lo sguardo dell'utente. Al fine di verificarne l'efficacia, si aggiunge alla scena un cavaliere che ci saluta quando il nostro sguardo incrocia il suo. Come si nota dalla figura, il cavaliere esegue l'azione appena lo osserviamo e mostra una scritta, visibile solo all'osservatore.

Figura 8.14: Il cavaliere saluta l'utente che ha incrociato il suo sguardo

Il calcolo del gaze avviene correttamente lato client e l'aggiornamento dello stesso lato server permette all'agente di percepirlo, eseguendo le azioni di conseguenza. Anche in questo caso, si è voluto verificare che un secondo spettatore potesse percepire l'utente rivolgere il proprio sguardo al cavaliere e visualizzare l'animazione di quest'ultimo, confermando di essere in un ambiente cooperativo di mixed-reality. Il risultato è stato ottimo, permettendo all'utente di osservare l'animazione dalla sua prospettiva.

Un ulteriore test del gaze è stato effettuato inserendo nella scena un secondo quadro, con implementazione simile al cavaliere. Se osservato dall'utente, il quadro mostra un messaggio di benvenuto; viceversa, visualizza un'immagine di default. Nella figura si può notare il cambiamento che avviene quando viene rilevato lo sguardo dell'utente: dapprima il quadro mostra solo una foto, successivamente anche una scritta. Sebbene si siano notati, con una periodicità casuale, lievi ritardi nella visualizzazione del messaggio, il comportamento del quadro ha rispecchiato quanto previsto, modificando il suo stato al variare del gaze dell'utente.

Figura 8.15: Ologramma rappresentante un quadro che mostra un messaggio di benvenuto, se osservato

Complessivamente, possiamo affermare che il sistema riesce a gestire esperienze cooperative di mixed reality e l'estensione integrata, basata su ARCore, migliora notevolmente la rappresentazione e l'ancoraggio degli ologrammi. Benché la posizione degli elementi possa risentire delle condizioni ambientali e della corretta comprensione di punti di interesse da parte del framework, non si sono evidenziate differenze tra la posizione prevista e quella effettiva tali da compromettere l'esperienza dell'utente.

Un limite riscontrato è dato dai dispositivi utilizzati, i tablet, che non hanno un sensore di profondità. Questo implica che il render degli ologrammi avvenga in primo piano, nascondendo eventuali utenti nelle vicinanze. Ad esempio, un utente all'interno della tenda non sarà visibile da un secondo partecipante, che visualizzerà solamente la tenda. Questa problematica può essere risolta con l'introduzione di dispositivi che forniscono questo sensore, ad esempio i visori.

Conclusioni

L'infrastruttura di MiRAgE ci permette di costruire sistemi di mixed reality, focalizzandosi sul modello delle entità e la loro rappresentazione digitale, delegando la logica applicativa a livello di framework. L'estensione sviluppata prosegue questo livello di astrazione, incapsulando il framework ARCore per introdurre le caratteristiche avanzate di comprensione dell'ambiente e motion tracking, che hanno un ruolo chiave nella visualizzazione e ancoraggio degli ologrammi.

L'integrazione di questo framework ha permesso di verificare l'efficacia della rappresentazione e affrontare tematiche quali accesso e percezione del mondo da parte dell'utente e interazione con gli elementi ivi presenti. Il livello di stabilità raggiunto ha permesso di utilizzare quanto sviluppato in un contesto reale, nello specifico Rocca delle Caminate, con un ottimo risultato.

Seguendo un approccio modulare nell'integrazione di ARCore, MiRAgE può essere esteso in futuro con ulteriori piattaforme, permettendo un livello alto di interoperabilità. Mentre i singoli framework tendono a limitare la propria compatibilità, integrandoli in MiRAgE è possibile ampliare la platea di fruitori che possono accedere a tutte le funzionalità dei mondi aumentati, sia con dispositivi sia con sistemi operativi differenti.

Inoltre, mentre le diverse tecnologie software oggi disponibili tendono a offrire esperienze di mixed reality cooperative con molte limitazioni, l'infrastruttura di MiRAgE, assieme alla gestione dell'interazione dell'utente implementata, cerca di superare questa barriera, garantendo agli utenti di essere partecipanti attivi, che percepiscono all'unisono cambiamenti che avvengono all'interno dell'ambiente e alle entità digitali.

Il risultato ottenuto ci permette di definire velocemente esperienze di mixed reality condivise, in cui la gestione di un approccio multi-marker e l'identificazione di piani garantiscono una buona stabilità nell'ancoraggio degli ologrammi nell'ambiente, riducendo la discrepanza tra reale e virtuale, aumentando così l'illusione della coesistenza dei mondi fisico e non.

Sviluppi futuri

Negli sviluppi futuri del framework sarà opportuno considerare non solo gli ologrammi ma anche elementi fisici all'interno dell'ambiente. Essi, similmente all'utente, avranno una registrazione all'interno del mondo, che definirà una controparte digitale, ovvero un'entità aumentata con le caratteristiche specifiche dell'oggetto fisico. Cambiamenti che avverranno nell'entità si propagheranno agli elementi fisici, che aggiorneranno il loro stato (per esempio, un led che si accende o spegne).

Inoltre, sarà necessario integrare ulteriori tecnologie, sia hardware che software, al fine di avere un grado di interoperabilità molto elevato. Per esempio, una futura integrazione del framework ARKit di Apple permetterà a tutti i dispositivi dotati di sistema operativo iOS di accedere e sfruttare le funzionalità dei mondi aumentati. In aggiunta, è necessario che siano effettuati studi e analisi sui visori oggi disponibili (Meta2 e Hololens, per esempio), per verificare la possibilità di una loro integrazione, abilitando anche questa tipologia di dispositivi. Lo scenario futuro è offrire a device differenti di accedere allo stesso mondo aumentato e percepire le stesse esperienze di mixed reality.

Benché l'architettura di MiRAgE sia stata ideata per supportare la distribuzione dei mondi aumentati, attualmente l'infrastruttura viene eseguita su un singolo nodo. Prima della release ufficiale, occorrerà contemplare anche un'esecuzione distribuita, eventualmente sfruttando soluzioni basate su cloud.

Infine, un ulteriore aspetto da esplorare in futuri aggiornamenti del framework è l'inserimento di tecniche e paradigmi di machine learning, in particolare *reinforcement learning*. In questo modo gli agenti, oltre a essere in grado di controllare le entità del mondo aumentato, sono in grado di apprendere e adattarsi ai cambiamenti che avvengono nell'ambiente in cui sono immersi, intraprendendo azioni di conseguenza.

Ringraziamenti

Anzitutto, vorrei ringraziare il relatore, il professore Alessandro Ricci e il correlatore, il Dott. Ing. Angelo Croatti per avermi dato l'occasione di intraprendere questo percorso di tesi e per la fiducia e disponibilità mostrate nei miei confronti.

Un ringraziamento speciale va alla mia famiglia. Spesso si tende a trascurare i sacrifici dei genitori per permettere ai propri figli di raggiungere i propri obiettivi. Il conseguimento di questo traguardo lo devo anche al loro continuo incoraggiamento e stima in me.

Un ringraziamento particolare va a mio fratello Simone, che ha sempre creduto in me ed è stato sempre presente, anche in quei momenti in cui io ero completamente assente.

Infine, vorrei ringraziare la mia seconda famiglia: gli amici. Il vostro aiuto e supporto è stato fondamentale in questi anni. Non vedo l'ora di poter condividere e festeggiare con voi questo traguardo, con l'euforia che da sempre ci contraddistingue.

Bibliografia

[1] Abraham Campbell, John W. Stafford, Thomas Holz, and Gregory O'Hare. *Why, when and how to use augmented reality agents (AuRAs).* Virtual Reality, 2013.

[2] Julie Carmigniani, Borko Furht, Marco Anisetti, Paolo Ceravolo, Ernesto Damiani, and Misa Ivkovic. Augmented reality technologies, systems and applications. *Multimedia Tools and Applications*, 2011.

[3] Angelo Croatti and Alessandro Ricci. Mashing up the physical and augmented reality: The web of augmented things idea. In *WoT*, 2017.

[4] Angelo Croatti and Alessandro Ricci. Towards the web of augmented things. *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017.

[5] Angelo Croatti and Alessandro Ricci. Developing agent-based pervasive mixed reality systems: The mirage framework. In *Advances in Practical Applications of Agents, Multi-Agent Systems, and Complexity: The PAAMS Collection*, 2018.

[6] Angelo Croatti and Alessandro Ricci. A model and platform for building agent-based pervasive mixed reality systems. In *Advances in Practical Applications of Agents, Multi-Agent Systems, and Complexity: The PAAMS Collection*, pages 127–139, 2018.

[7] Henry Fuchs, Mark A. Livingston, Ramesh Raskar, D'nardo Colucci, Kurtis Keller, Andrei State, Jessica R. Crawford, Paul Rademacher, Samuel H. Drake, and Anthony A. Meyer. Augmented reality visualization for laparoscopic surgery. In *Medical Image Computing and Computer-Assisted Intervention — MICCAI'98*, pages 934–943, 1998.

[8] Thomas Holz, Abraham G. Campbell, Gregory M. P. O'Hare, John W. Stafford, Alan Martin, and Mauro Dragone. *MiRA-Mixed Reality Agents.* Academic Press, Inc., 2011.

[9] Nicholas R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 2000.

[10] Peddie Jon. *Augmented Reality: Where We Will All Live*. Springer Publishing Company, Incorporated, 2017.

[11] Timothy Jung, M. Claudia tom Dieck, Hyunae Lee, editor="Inversini Alessandro Chung, Namho, and Roland" Schegg. Effects of virtual reality and augmented reality on visitor experiences in museum. In *Information and Communication Technologies in Tourism 2016*, pages 621–635, 2016.

[12] Kangdon Lee. *Augmented Reality in Education and Training*. TechTrends, 2012.

[13] Lanham M. *Learn ARCore - Fundamentals of Google ARCore: Learn to build augmented reality apps for Android, Unity, and the web with Google ARCore 1.0*. Packt Publishing, 2018.

[14] Paul Milgram, Haruo Takemura, Akira Utsumi, and Fumio Kishino. Augmented reality: A class of displays on the reality-virtuality continuum. *Telemanipulator and Telepresence Technologies*, 1994.

[15] Andrew Nee, S K Ong, George Chryssolouris, and Dimitris Mourtzis. Augmented reality applications in design and manufacturing. *CIRP Annals - Manufacturing Technology*, pages 657–679, 2012.

[16] Alessandro Ricci, Michele Piunti, Luca Tummolini, and Cristiano Castelfranchi. The mirror world: Preparing for mixed-reality living. *Pervasive Computing, IEEE*, 2015.

[17] Ronald T. Azuma. A survey of augmented reality. *Presence: Teleoperators and Virtual Environments*, 1996.