ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

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

STUDIO, PROGETTAZIONE E SVILUPPO DI APPLICAZIONI DI REALTÀ MISTA COOPERATIVE

Elaborato in
SISTEMI EMBEDDED E INTERNET-OF-THINGS

Relatore
Prof. ALESSANDRO RICCI

Presentata da
FRANCESCO MAGNANI

Corelatore
SAMUELE BURATTINI

Anno Accademico 2021 – 2022

"The advance of technology is based on making it fit in so that you don't really even notice it, so it's part of everyday life"
Bill Gates

Indice

Introduzione

> Viviamo in un mondo complesso, pieno di una miriade di oggetti, strumenti, giocattoli e persone. Le nostre vite vengono spese in diverse interazioni con l'ambiente. Per la maggior parte, eppure, il calcolo informatico ha luogo mentre siamo seduti davanti a un singolo schermo luminoso fissandolo con collegati un insieme di pulsanti e un mouse. [...] Come possiamo fuggire dallo schermo del computer e mettere insieme questi due mondi?
>
> (*"Back to the real world"* — Pierre Wellner, Wendy Mackay e Rich Gold, 1993)

Con questa introduzione si apre l'articolo del 1993 di Pierre Wellner, Wendy Mackay e Rich Gold intitolato *"Back to the real world"* [1]. Se ci pensiamo bene, infatti, il livello di interazione umana possibile con un computer è veramente basso. I sistemi di input tradizionali altro non sono che alcuni pulsanti e una piccola periferica che permette il movimento di un puntatore. Non solo: anche le informazioni visibili e ottenibili da un computer sono solo immagini bidimensionali contenute in finestre sovrapponibili. L'uomo possiede molti altri strumenti per comprendere e manipolare informazioni molto più complesse di così.

L'informatica ha ormai portato la computazione a livelli di astrazione sempre più alti, sempre più vicini alla comprensione diretta dell'uomo. Le nuove tecnologie, realtà aumentata (AR) e realtà virtuale (VR), sono diventate attualità ormai da diversi anni e sono già state integrate in diversi ambienti. Ma quando tornano utili? Nell'articolo *"Augmented reality technologies, systems and applications"*, [3] di Julie Carmigniani vengono citate tante interessanti applicazioni di queste tecnologie. Solo per fare degli esempi, i possibili usi vanno dal campo della pubblicità all'educazione e medicina e tanto altro. E' possibile creare applicazioni che aiutino un medico o chirurgo nell'operazione, utilizzando la realtà mista (MR) come tramite per fornire dati importanti sul paziente o mostrare in forma di ologramma aiuti medici a chi porta il visore. Sempre in medicina è possibile potenziare la diagnosi, fornendo la terza dimensione a test e immagini prima bi-dimensionali. E' possibile creare in

forma di ologramma un'anteprima di una casa, di una stanza, di un mobile o di un qualsiasi oggetto per vederlo inserito nell'ambiente prima ancora che quell'oggetto esista in forma fisica. Per riassumere, tutto ciò che veniva prima fatto su un computer (o che ancora non è stato possibile fare) che richiedesse una qualche forma di interazione tridimensionale o inserimento nell'ambiente, è ora possibile in forma del tutto nuova e più vicina alla nostra percezione del mondo. Si capisce che più si pensa in questi termini, e più si comprendono le reali potenzialità di queste tecnologie.

Il mondo è oramai connesso informaticamente parlando. Da sempre nel mondo reale ci troviamo spesso a lavorare e interagire con altre persone. Se questa realtà si dovesse spostare in uno spettro non più solo "fisico" ma anche virtuale, questa interazione non dovrebbe venire meno. Ecco perché nasce l'esigenza delle esperienze cooperative (condivise), applicazioni in cui sia possibile "interoperare" con altri utenti allo stesso tempo, lavorando quindi in due realtà, fisica e virtuale. Questa forma di cooperazione già esiste da tempo, ma applicarla nel campo della Mixed Reality ne trasforma completamente il senso e le modalità. Come sviluppare, però, per queste tecnologie? Sono già mature? Esistono già strumenti affidabili che rendano agile l'accesso ad esse da parte di tutti? L'obiettivo di questa tesi consiste proprio in questo: capire qual è lo stato dell'arte nel panorama delle tecnologie e librerie che permettono di sviluppare in Mixed Reality applicazioni per esperienze cooperative. Vedremo cosa aspetta uno sviluppatore che voglia avvicinarsi a questo mondo, quali sono le principali difficoltà e qualche caso pratico come riferimento. In particolare, vedremo che cosa esiste per Unity, un motore grafico ormai molto utilizzato, che si è trovato a rappresentare un possibile, per ora incerto standard *de facto* per lo sviluppo di queste applicazioni. Nella prima parte saranno, quindi, affrontate le basi della Mixed Reality e le peculiarità delle esperienze condivise. Nella seconda parte, invece, applicheremo l'analisi e l'argomentazione delle tematiche precedentemente affrontate in un semplice caso implementativo, con un'applicazione Mixed Reality per il visore Hololens 2 di Microsoft.

Capitolo 1

Concetti introduttivi

Di fronte a progetti che sfiorano termini come Extended Reality, Mixed reality, Virtual reality ecc. è sempre opportuno introdurne i relativi significati, considerando l'erroneo utilizzo che spesso se ne fa (e, in certi casi, la mancanza di una vera definizione formale). Il rischio principale è quello di creare fraintendimenti su certe espressioni, oppure quello di ampliarne erroneamente la semantica nonostante siano, a tutti gli effetti, distinti campi di ricerca e studio. Questa prima sezione è dedicata all'introduzione proprio di quei concetti che serviranno ad una più profonda esplorazione delle esperienze condivise in Mixed Reality e delle tecnologie ad esse correlate.

1.1 Extended Reality

L'espressione forse meno conosciuta tra le altre già citate è proprio quella che meglio si presta all'introduzione di tutte quante, derivate. **"Extended Reality"**, ovvero realtà estesa, rappresenta tutti gli ambienti virtuali/reali in cui un'interazione uomo-macchina può venire generata/supportata da una tecnologia e da dispositivi esterni di cui l'utente può usufruire.

L'Extended Reality (d'ora in poi abbreviata in XR) racchiude tutti gli altri mondi di cui andremo a parlare, ovvero Virtual Reality (VR), Augmented Reality (AR) e Mixed Reality (MR) ma, essendo il termine ancora piuttosto nuovo, ancora oggi non tutti hanno una concezione univoca del termine. Per essere chiari e concisi, possiamo riassumere questo termine come tutto ciò che rappresenta un cambiamento del modo con cui le persone percepiscono la realtà, un nuovo modo di intendere le interazioni con un computer e dei relativi modelli che portano l'esperienza dell'utente ad un livello completamente nuovo.

1.2 Virtual Reality e Augmented Reality

Sostanzialmente diversi tra loro, VR e AR sono due concetti che stanno alla base della XR. Nonostante la loro fondamentale differenza, è valida la scelta di affrontare insieme i due concetti per meglio comprenderne il significato. "Virtual Reality" significa un tipo di simulazione della realtà che utilizza come tramite un mondo interamente virtuale. In essa, un utente è completamente immerso in questo nuovo ambiente ed è incapacitato ad avere interazioni con la realtà al di fuori: per questo si dice che la realtà fisica viene *sostituita*. Un esempio di Virtual Reality è un visore che copre totalmente la vista dell'utente, proiettando al suo interno immagini "spaziali", ovvero a 360 gradi.

Augmented Reality (realtà aumentata) rappresenta uno stacco meno netto rispetto al mondo circostante, in quanto l'utente è ancora in grado di interagire con tutto ciò che è reale e può muoversi in esso. Oltre a ciò, tuttavia, subisce un "aumento" di questa realtà tramite elementi virtuali che si sovrappongono alla visione e possono rappresentare uno strumento, un aiuto o in generale un arricchimento per la percezione sensoriale umana. E' curioso notare come la realtà aumentata sia spesso presentata [4] [5] come un sottotipo della realtà virtuale, o meglio una sua *variazione*. Questo testimonia la differenza di visioni che riguardano questi concetti: forse la definizione stessa di Augmented Reality presuppone l'esistenza di una fusione con più mondi, e quindi non si può etichettare come un estremo opposto alla VR. Un esempio di Augmented Reality è un occhiale che mostra un ologramma con informazioni sul meteo sovrapposte a ciò che vediamo.

1.3 Mixed Reality

Che cosa significa Mixed Reality? Il termine "realtà mista" è stato introdotto nel 1994, in un articolo di Paul Milgram e Fumio Kishino, *"A Taxonomy of Mixed reality Visual Display"* [2], articolo fondamentale per la definizione di questi concetti. "Realtà mista" significa una fusione tra mondo fisico e digitale, un nuovo ambiente in cui sono possibili interazioni uomo-macchina intuitive e naturali immerse in uno spazio 3D che ha quindi "due facce", ciascuna con diverse proprietà.

Spesso diverse applicazioni della realtà mista non si pongono allo stesso modo per quanto riguarda la collocazione tra i due mondi, fisico e virtuale. A seconda di come vengono modificati questi due estremi, la realtà può assumere diversi aspetti, più virtuali o più reali. Per questo, più che definire due concetti contrapposti, si parla di **spettro** (continuum) della realtà mista, intervallo che rappresenta dove è collocato il nostro mondo "misto", più virtuale o più fisico. Mentre gli estremi rappresentano le due realtà, lo spazio che c'è tra essi

Figura 1.1: Spettro della realtà mista, adattato da quello di Milgram e Kishino [2]

consiste sia di **augmented reality**, ovvero un potenziamento virtuale dello spazio reale, sia di **augmented virtuality**, cioè un'aggiunta di elementi reali ad uno spazio virtuale [2].

1.3.1 "Oggetti" della realtà mista

Gli ologrammi sono alla base dell'applicazione in realtà mista. Sono ciò che rende possibile l'interazione uomo-macchina in questo tipo di esperienza. Un **ologramma** è un oggetto fatto di luce che appare nel mondo intorno all'utente, apparendo come fosse un oggetto reale. Come è stato già detto, gli ologrammi sono alla base dell'interazione con il mondo dell'applicazione MR, di conseguenza devono essere manipolabili, interagibili ma, soprattutto, devono reagire alle superfici del mondo reale. E' sicuramente fondamentale in esperienze di realtà mista, sapere gestire con cura gli ologrammi: è facile trovarsi ad avere a che fare con certi comportamenti "innaturali" di questi oggetti virtuali, che richiedono molta attenzione nel loro posizionamento e movimento. Non si deve, cioè, dare l'idea che siano "adesivi" appiccicati sulla vista dell'utente, e donare loro, quindi, un impianto tridimensionale.

1.3.2 Differenze tra MR e AR

Sebbene la definizione di AR precedentemente data non sia tra le più complete, appare chiara l'esistenza di una sovrapposizione tra AR e MR (come si vede anche dalla figura 1.1): entrambe, infatti, sembrano voler ottenere una commistione tra i due mondi, reale e virtuale. Per essere più chiari, però, **l'AR non presuppone nessuna interazione della parte virtuale con quella reale, ma solo un'aggiunta ad essa** (per questo "aumentata"). La Mixed Reality, invece, rappresenta anche un insieme di esperienze in cui avviene una fusione completa tra le due parti, motivo per cui si chiama "mixed", proprio a voler sottolineare questo aspetto di mescolanza tra i due mondi. In MR,

per esempio, sarebbe possibile piazzare un ologramma virtuale di una palla su un tavolo reale, come quello di una cucina, e vederla rimbalzare su di esso o sulle pareti della stanza qualora un utente decidesse di dare un colpetto con la mano a questo ologramma. In AR questo ologramma potrebbe apparire di fronte a noi, ma sarebbe impossibile vederlo interagire con il mondo reale.

In seguito a questa spiegazione la differenza tra MR e AR potrebbe apparire a prima vista come una differenza solo di concetto, di convenzione: la semplice aggiunta di un elemento di interazione tra questi due mondi, tuttavia, presuppone l'entrata in gioco di tecnologie hardware e software totalmente differenti.

1.3.3 Interfacciamento tra due realtà

Un'applicazione in realtà mista ha bisogno di strumenti come l'intelligenza artificiale, tracciamento dello sguardo, sensori e tanto altro per poter ottenere risultati soddisfacenti, motivo per cui questo aspetto verrà trattato più dettagliatamente nelle sezioni successive. Per quale motivo sono necessari questi strumenti? Un'applicazione in MR richiede più proprietà di un'altra in AR: se l'utente chiedesse, per esempio, all'applicazione AR di piazzare un ologramma dietro la sedia che gli sta davanti, come farebbe l'applicazione a capire quali parti dell'ologramma mostrare e quali no? La sua visibilità dipenderebbe dalla posizione e dall'angolo di vista dell'utente che indossa il dispositivo. Questa proprietà in particolare viene chiamata **occlusione**.

Avere una conoscenza del mondo in entrambe le realtà significa, nel caso della realtà mista, ottenere dati a sufficienza per "capire" com'è fatto lo spazio 3D che l'utente ha davanti e come quest'ultimo si muove in esso. Il dispositivo deve "ricordare" ciò che ha appreso sul mondo e aggiornarlo continuamente per poter ottenere una consapevolezza sempre più dettagliata e precisa dello spazio che ci sta attorno: questa capacità viene chiamata **spatial mapping**. Si potrebbe dire che il mondo reale esiste, quindi, in due versioni: una è quella che vediamo e che viene ripresa dal dispositivo, l'altra è creata dallo stesso e muta continuamente, cercando di clonare quella realmente esistente ed utilizzata per le interazioni tra il mondo virtuale e reale. Per questo motivo se fosse necessario piazzare un ologramma dietro un oggetto reale, il dispositivo MR deve sfruttare la copia del mondo reale che lui stesso ha creato tramite delle stime (in un certo senso, come il dispositivo "crede che sia" il mondo fisico) per capire che quell'ologramma va renderizzato oppure no e come. Certo, in questo caso si stanno sfiorando dettagli implementativi, ma allo stesso tempo si tratta di un *requisito* della realtà mista, necessario per "mischiare" i due mondi.

Questi sono solo alcuni elementi richiesti da un'applicazione MR. Più il livello di interazione richiesto aumenta, più si rendono necessari ulteriori strumenti. Se un utente deve interagire con un ologramma, ad esempio, è necessario che alcuni suoi movimenti vengano tracciati, per esempio quelli della mano, delle dita o anche del polso.

Capitolo 2

Tecnologie di sviluppo MR

Prima di passare ad un caso di studio applicativo, è bene fare una panoramica di ciò che esiste nel mondo delle tecnologie in grado di ospitare la realtà mista. Questo capitolo servirà a meglio esplorare cosa sono e cosa rappresentano.

2.1 Ambienti di sviluppo

Approccio comune allo sviluppo di applicazioni in Mixed Reality è l'utilizzo di ambienti di sviluppo che già hanno dimostrato la propria robustezza per altri progetti, soprattutto quelli in ambito gaming. Lo sviluppo di videogiochi e quello di applicazioni in realtà mista hanno, come si noterà, diversi punti in comune, in maniera particolare nel contesto di giochi multigiocatore ed esperienze condivise in MR.

2.1.1 Unity

Unity è un motore grafico multi-piattaforma, sviluppato da Unity Technologies per la creazione di videogiochi o altri contenuti come simulatori, visualiz-

Figura 2.1: Logo di Unity

zatori di architettura, ingegneria ecc. Al giorno d'oggi Unity viene largamente utilizzato da tantissimi creatori di contenuti, circa 1,5 milioni utenti lo scelgono mensilmente per creare videogiochi e altro.

Unity ha saputo proporsi spesso come motore principe per molti progetti, tra cui progetti AR e MR: i suoi vantaggi sono principalmente il **costo**, che è nullo visto che è gratuito e la vistosa **portabilità** (le creazioni di Unity possono essere eseguite su più di 20 piattaforme diverse). L'obiettivo della compagnia fu, infatti, quello di rendere lo sviluppo di videogiochi e non solo disponibile a tutti.

Unity si sa distinguere fin da subito per la sua semplicità d'uso: gli effetti di ciò che si crea sono immediatamente visibili e trasparenti, la modellazione dell'applicazione risulta naturale e veloce. Le entità principali sono i **GameObject**, tassello base di ogni applicazione Unity, che possono avere più **Components**, ovvero script che ne determinano il comportamento. Non solo: su Unity si possono salvare certi GameObject così come sono per poter essere riutilizzati, i cosiddetti **Prefab**.

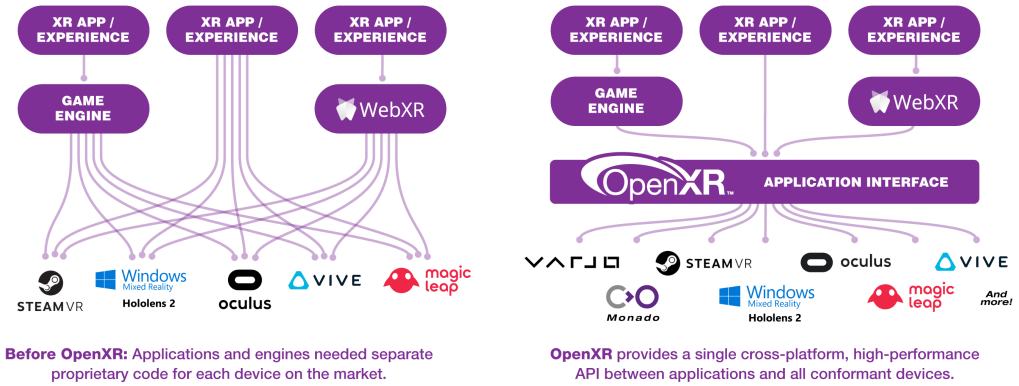
Unity dispone, inoltre, di un'interfaccia di debugging che permette facilmente di vedere i risultati di cambiamenti apportati ai GameObject in tempo reale. Ovviamente queste sono solo alcune delle tantissime funzioni di cui dispone.

2.1.2 Unreal Engine

Unreal Engine è un motore grafico sviluppato da Epic Games, principalmente orientato allo sviluppo di videogiochi, come Unity. Grazie alla sua compatibilità con molti strumenti presenti anche su Unity e al fatto di essere di gratuito (proprio come il motore di Unity Technologies), si può a tutti gli effetti identificare come il principale competitor di quest'ultimo. A differenza di Unity, tuttavia, che si è presentato come motore che consente uno sviluppo semplice e più adatto ai meno esperti, Unreal sembra più puntare sull'aspetto **fotorealistico** e **immersivo** delle sue applicazioni (grazie anche al suo utilizzo di C++, linguaggio noto per la sua efficienza).

2.1.3 Sviluppo nativo

> *"OpenXR cerca di semplificare lo sviluppo AR/VR, permettendo alle applicazioni di raggiungere un più ampio insieme di piattaforme hardware senza dover "portare" e riscrivere il codice e successivamente permettendo ai fornitori di piattaforme che supportano OpenXR di accedere a più applicazioni. Con l'uscita delle specifiche*



(a) Prima di OpenXR (b) Dopo OpenXR

Figura 2.2: Cambiamento dopo la nascita di OpenXR

> *di OpenXR 1.0, gli sviluppatori possono creare* ***vere esperienze XR (extended reality) cross-platform****".*
>
> (Brent E. Insko — Lead XR architect presso Intel)

Grazie a standard come **OpenXR** è possibile anche realizzare la propria implementazione del motore grafico, magari ottimizzato per l'applicazione finale. Che cos'è OpenXR?

OpenXR è uno standard aperto che offre alte prestazioni per l'accesso ai dispositivi per la realtà aumentata e virtuale (collettivamente dispositivi XR). Le implementazioni di OpenXR sono innumerevoli: HTC, Microsoft, Oculus, SteamVR, Varjo.

OpenXR è stato creato dal Khronos Group, un consorzio aperto e senza fini di lucro di oltre 150 compagnie leader del settore che dal 2000 crea standard aperti e royalty-free per vari ambiti dell'informatica, tra cui grafica 3D, VR e AR, programmazione parallela e machine learning. Da sempre ha permesso un'accelerazione nel portare tecnologie all'avanguardia ad uno standard accettato e disponibile al pubblico.

Tutti gli strumenti, compreso MRTK (sezione 2.3.1), che puntano allo sviluppo cross-platform su extended reality si basano su questo standard. Esso, infatti, permette di avere uno strato sottostante che, grazie ad un'API, realizzano l'accesso a funzionalità utilizzate comunemente sui vari dispositivi.

Come si vede nella Figura 2.2, prima dell'entrata in scena di OpenXR la frammentazione presente nel mondo VR e AR rendeva impossibile la creazione di una base comune, motivo per cui esistevano numerosi problemi per chi voleva puntare allo sviluppo su queste piattaforme. Tra questi, sicuramente **costi maggiori**, **spreco di tempo e risorse** e la **non portabilità** delle applicazioni.

2.1.4 Web

Lo sviluppo web ha preso sempre più piede e ancora oggi è in crescita. Anche per applicazioni per AR e MR è possibile sviluppare applicazioni implementando lo standard **WebXR**, un'API che supporta l'accesso per dispositivi di realtà aumentata e virtuale ad applicazioni Web, utilizzando quindi Javascript come linguaggio.

Croquet OS

In questa sezione si è ritenuto opportuno citare l'esempio di **Croquet OS**, un sistema operativo basato sul web per la creazione di applicazioni tridimensionali, particolarmente focalizzato su quelle multi-utente. Il sistema di sincronizzazione di Croquet permette agli utenti di lavorare insieme con bassissima latenza adatta ad applicazioni **real time**, senza dover intervenire con codice server-side.

Particolarmente interessante è il sistema degli **snapshots** che mette a disposizione, che consiste in copie della sessione corrente che permettono agli utenti che entrano nella stessa sessione di sincronizzarsi con gli altri interagendo con questi oggetti presenti nel sistema distribuito. Croquet offre molti spunti di riflessione per quanto riguarda l'architettura (capitolo 4) del motore utilizzato per lo sviluppo di applicazioni MR, ma non saranno il fulcro di questa esperienza.

2.2 Dispositivi MR

Nonostante diverse aziende si stiano avvicinando allo sviluppo di piattaforme per la realtà mista, attualmente non esistono numerose opzioni valide per dispositivi Mixed Reality/Augmented Reality, e quelle che esistono non offrono tutte lo stesso livello di immersione.

2.2.1 Hololens

Hololens, o meglio **Hololens 2**, che rappresenta la seconda generazione, è un dispositivo per realtà mista di Microsoft, pensato per diversi usi tra cui quello industriale, progettuale e sanitario. Esso ricade, in particolare, in quella categoria di *display device* chiamata **Head-mounted display** (HMD), ovvero un display ottico da indossare come una specie di elmo. Esistono diversi tipi di HMD, al giorno d'oggi la maggior parte sono visori per la realtà virtuale (oculus, htc vive ecc.): hololens, invece, è il principale visore per la realtà mista di questo tipo.

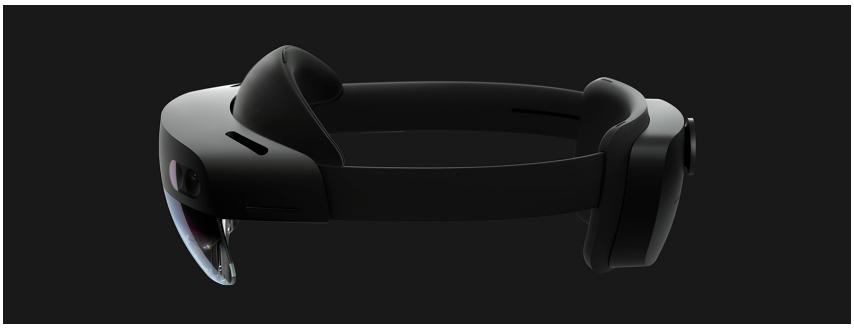


Figura 2.3: Hololens 2 visto di profilo

Hololens dispone di tantissime funzionalità: tracciamento della mano, tracciamento oculare, tracciamento della testa, percezione dello spazio e tanto altro. Oltre a questo, presenta un hardware ottimizzato il rendering di ologrammi e per gli algoritmi di visione artificiale. Holographic Processing Unit (HPU) è il nome del coprocessore che conduce il calcolo dei dati per integrare il mondo reale nell'esperienza mista. Esso libera la CPU da questi calcoli e dall'interazione con l'IMU, ovvero l'unità di misurazione inerziale, che serve a combinare i movimenti della testa nei calcoli. Non solo: l'HPU è incaricata anche di processare le gestures e i dati della voce.

2.2.2 Magic Leap e altri visori

Magic Leap 2 è un visore realizzato dalla startup americana Magic Leap, Inc. Come Hololens è un HMD che proietta luce in delle lenti per creare immagini 3D olografiche. A differenza di Hololens, offre un interfaccia aperta Android AOSP(Android Open Source Project)-based, è più leggero e possiede specifiche hardware leggermente superiori.

Nonostante i vari punti di forza di quest'ultimo non possiede certe feature di Hololens, interessanti per questo progetto, per esempio il potente salvataggio di informazioni sull'ambiente fisico o la ricca collezione di gestures disponibili. La stessa app presa come esempio per questo progetto, Microsoft Mesh, sviluppata e pensata per Hololens, ha fatto spostare il caso di studio su quest'ultima piattaforma.

Altri visori esistenti attualmente sul mercato come per esempio il Lenovo ThinkReality A3 sono considerati visori AR, dunque non offrono lo stesso livello di immersione e non verranno trattati in questa esperienza.

Figura 2.4: Logo del MRTK

2.3 Strumenti e librerie

Lo sviluppo di librerie cross-platform è vitale per la nascita di una comunità di sviluppatori su dispositivi di nuova generazione. In particolare il **Mixed Reality Toolkit** ha avuto un ruolo cruciale per l'inizio dello sviluppo su visori di realtà mista, ed è forse tuttora il maggior SDK (Software Development Kit) per la creazione di applicazioni MR.

2.3.1 MRTK

Mixed Reality ToolKit (MRTK) è un progetto condotto da Microsoft che punta ad accellerare lo sviluppo di applicazioni Mixed Reality cross-platform, contenente una serie di componenti e features supportati da diversi dispositivi (tra cui ovviamente Hololens)[16]. Microsoft, attraverso MRTK, si propone di offrire diverse funzionalità, tra cui:

- Un sistema di input cross-platform e componenti che permettono la costruzione di interazioni spaziali e UI
- Prototipazione veloce tramite una simulazione in-editor
- Estensibilità del framework per permettere allo sviluppatore di scambiare alcuni componenti chiave.

Il materiale fornito da questo progetto è fondamentale per lo sviluppo di applicazioni MR: tantissimi aspetti, tra cui il sistema di input, vengono enormemente semplificati grazie ad esso. Il mondo del Mixed Reality, ma anche dei contraltari AR e VR, ha ancora bisogno di molta "standardizzazione", per questo motivo progetti come questo (ma anche soprattutto OpenXR) sono degli ottimi punti di partenza per lo sviluppo di questo settore.

Integrazione tra MRTK e Unity

Grazie a comodità e punti di forza spiegati precedentemente, Unity ben si presta allo sviluppo per applicazioni che richiedono **movimenti in mondi 3D in cui gli oggetti sono interagibili**. Per questo motivo, Microsoft ha scelto Unity come motore principale per lo sviluppo di applicazioni MR, integrando il set di componenti da loro fornito, ovvero MRTK, proprio con il motore grafico in questione.

Tale set di componenti, infatti, si traduce in un pacchetto contenente prefab di oggetti come una tastiera virtuale, ologrammi di bottoni, caselle di testo ecc. All'interno si trovano anche numerosi script per facilitare il comportamento dei GameObject dentro l'ambiente MR. Uno degli script più usati, ad esempio, è **Object Manipulator**, che permette la trasformazione usando il sistema di hand-tracking di Hololens 2 dei GameObject che possiedono questo componente.

2.3.2 Unity XR e altri strumenti

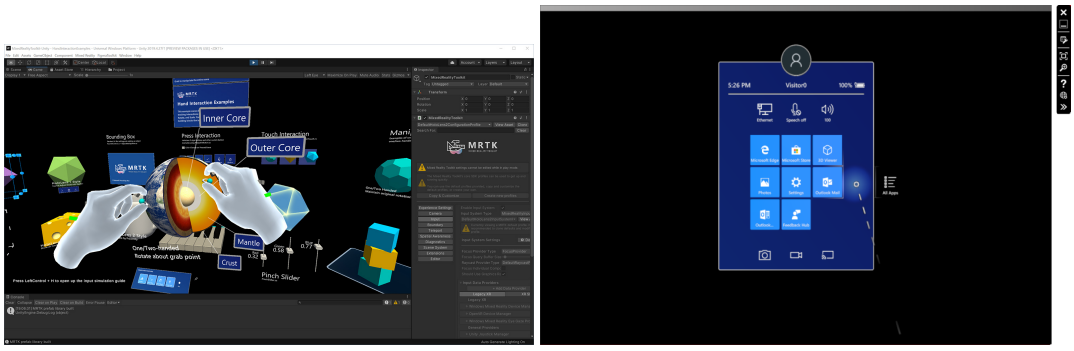
L'MRTK non è, tuttavia, l'unico strumento/libreria attualmente disponibile per lo sviluppo MR. Unity stessa ha messo a disposizione un toolkit chiamato Unity XR (Extended Reality) per questo scopo, che può essere utilizzato sia per la virtual reality che augmented reality, e per quest'ultima mette a disposizione **ARFoundation**, il framework specifico per l'AR. Non solo: supporta anche una grande varietà di piattaforme tra cui anche Hololens e Magic Leap.

Esistono anche altre librerie e SDK, alcune ancora scarsamente utilizzate ma comunque alternative alle due già citate. La maggior parte di queste, tuttavia, è ancora orientata all'AR e non propriamente MR. Una è ad esempio ARCore, sviluppata da Google e pensata per smartphone e tablet, contrapposta ad ARKit di Apple. Altre sono il Vuforia SDK, o anche minori come il Bouvet Development Kit (BDK), pensato per Hololens 2.

E' da notare che molti di questi strumenti sono compatibili l'uno con l'altro, perciò questi mondi possono facilmente intersecarsi.

2.4 Modalità di Debugging

Una delle prime preoccupazioni di uno sviluppatore, preparandosi a sviluppare per visori o dispositivi dalle funzioni particolari (come Hololens e gli altri visori di realtà mista) può essere sicuramente il debugging. Testare la propria applicazione su dispositivi di questo tipo può diventare molto complicato, sia perché possiedono un hardware completamente diverso sia perché



(a) Simulazione all'interno di Unity (b) Simulazione usando l'emulatore

Figura 2.5: Strumenti per debugging

certe capacità di questi visori non sono presenti in un normale PC (come il riconoscimento di gestures da parte dell'HPU per esempio). Per fortuna esistono varie alternative, nello specifico due, usate entrambe in quest'esperienza.

2.4.1 Play Mode di Unity

All'interno di Unity è possibile testare l'applicazione semplicemente premendo un tasto, il che ti permette di accedere ad una schermata in cui un emulatore simula un ambiente MR utilizzando uno spazio interamente virtuale. Nonostante sia possibile emulare alcune gestures come ad esempio la rotazione del polso, altre funzionalità sono impossibili da implementare correttamente: la lettura di QR code non è utilizzabile così come la fotocamera.

2.4.2 Hololens Emulator

Per quanto riguarda Hololens, Microsoft mette a disposizione anche un emulatore installabile separatamente per testare le applicazioni olografiche sul PC. Viene utilizzata una macchina virtuale dove è possibile testare input umani e ambientali letti dai sensori di Hololens, simulandoli tramite tastiera, mouse o controller.

Nonostante sia più avanzato del corrispettivo su Unity, anche su questo emulatore è impossibile testare certe funzionalità, come le ancore spaziali (3.5) e, inoltre, per testare le applicazioni è necessario effettuare il deploy dell'applicazione, senza la possibilità di lanciare al bisogno il programma durante lo sviluppo. Per questo motivo, questo strumento verrà scarsamente considerato.

2.5 Framework di rete

Il principale caso di studio e cardine dell'intera analisi dello sviluppo di applicazioni MR cooperative consiste proprio nello strumento che permette una comunicazione di rete adattabile ad un contesto così particolare com'è quello della realtà mista cooperativa.

Essendo il principale fulcro di questa esperienza, è chiaro che questo sarà l'aspetto poi sviscerato nei capitoli intermedi di questa tesi. Per fare, quindi, un introduzione, cosa andrà ricercato in queste tecnologie e perché?

Muovendoci verso i prossimi capitoli, verranno specificate le tecnologie utilizzate per il nostro caso di studio, tra cui il citato Unity (2.1.1); per questo motivo appare naturale chiedersi quali librerie o strumenti esistono per questo engine. La domanda tocca tante possibili risposte: Unity, infatti, essendo un ambiente per lo più orientato ai videogiochi, offre diverse risorse per la creazione di **giochi multiplayer**. Questa tipologia di giochi, come già detto, è molto simile alle applicazioni MR, ma in un aspetto particolare differisce in maniera radicale: la concezione dello spazio.

Tecnologie che consentono il trasporto via rete di informazioni quali la posizione, rotazione, velocità o stato (generico) di oggetti **real-time** necessitano di un controllo stretto di quello che è lo spazio (sistema di coordinate) in cui si trovano, motivo per cui questi framework di rete utilizzabili per Unity sono costruiti basandosi sul fatto di trovarsi in uno spazio **interamente virtuale**, in cui dati come la posizione non sono affatto *stimati*, bensì sono **esatti e condivisibili**. Non solo: la presenza di un server che centralizza la complessità o meno nell'architettura di rete usata cambia totalmente il modo di gestire l'applicazione.

Com'è possibile notare da queste incertezze, la scelta di uno strumento rispetto ad un altro e il modo di integrarlo nella realtà mista rappresentano aspetti non banali e da verificare. Questa voragine nello sviluppo sarà quindi da affrontare in un capitolo a parte.

Capitolo 3

Esperienze Condivise in Realtà Mista: Inquadramento e Analisi dei problemi

In questo capitolo verranno approfondite le **esperienze condivise** in realtà mista, punto focale di questa esperienza, prendendo in esame le difficoltà e le sfide incontrate durante lo sviluppo del progetto applicativo, preso come caso di studio. Il tipo di esperienza a cui si mirerà sarà un'applicazione in realtà mista che permetta a più utenti di modificare gli stessi ologrammi, di crearne di nuovi, manipolarne alcune caratteristiche (come ad esempio posizione, dimensione, rotazione e stato, inteso come una generica caratteristica interna a ciascun ologramma e rappresentabile esternamente in qualche modo, per esempio il colore dell'ologramma). Per questa applicazione verrà utilizzato Unity e come dispositivo target Hololens 2.

In questa sezione i termini "utente" e "client" verranno spesso usati per indicare la stessa entità, dove il primo termine fa più riferimento alla persona utilizzatrice e il client al dispositivo connesso alla rete.

3.1 Esperienze condivise

Un'esperienza condivisa è un applicazione che permette di renderizzare gli stessi oggetti, in quanto rappresentazione (ologramma) e in quanto concetto, tra più utenti con più dispositivi. In particolare Mark Billinghurst e Hirokazu Kato in "Collaborative Augmented Reality" [13] citano quattro attributi di ambienti collaborativi in realtà aumentata (in questo caso validi anche per la realtà mista in generale):

- **Virtuality**: oggetti che non esistono nel mondo reale possono essere visti ed esaminati
- **Augmentation**: oggetti reali possono essere "aumentati" usando annotazioni virtuali
- **Cooperation**: multipli utenti possono vedersi a vicenda e cooperare in maniera naturale
- **Independence**: utenti individuali controllano il proprio punto di vista indipendentemente.

Nonostante si stia trattando le esperienze condivise ad un alto livello di astrazione, è chiaro che una definizione del genere presupponga l'utilizzo della rete o un altro mezzo di comunicazione, che è necessario ai fini della realizzazione di questo genere di applicazioni.

Si tratta di una categoria enorme di applicazioni, basti pensare alle esperienze multi-utente non in realtà mista: app per video-chiamare, editor di testo con accesso condiviso o videogiochi multi-giocatore ecc.

Proprio su quest'ultimo esempio vale la pena soffermarsi, in quanto forse è proprio il caso che più si avvicina alle esperienze condivise in realtà mista. In alcuni videogiochi moderni, infatti, è possibile muoversi in uno spazio 3D dove certe caratteristiche delle entità che vi partecipano, come la posizione dei giocatori, dev'essere condivisa da tutti gli utenti. Se ci si sofferma brevemente, questa descrizione può corrispondere ad una sintetica, approssimativa spiegazione di cosa deve fare un'applicazione per ottenere un'esperienza di MR condivisa. Non è un caso che, attualmente, lo standard de facto per sviluppare applicazioni MR sia Unity, un software utilizzato principalmente per la creazione di videogiochi, introdotto precedentemente ed utilizzato in questo progetto.

3.1.1 Due aspetti chiave delle Esperienze Condivise

Per creare una vera realtà mista cooperativa devono essere verificate due importanti proprietà, forse tra le più articolate e complesse da perseguire: **interoperabilità** e **consistenza**.

L'**interoperabilità** rappresenta la misura con cui due sistemi riescono ad operare insieme (*interoperare* appunto), e la qualità di questa interazione. Essa consiste nel rendere possibile un'interazione trasparente e naturale tra tutte le entità coinvolte nella stessa realtà. Per dirla in maniera più concreta, significa offrire uno scambio tra più dispositivi, anche eterogenei, in modo da farli interagire e da condividere dei flussi di informazioni. È facile intuire quante

sfaccettature si porti dietro una proprietà del genere. In una realtà che sta a metà tra due mondi, virtuale e fisico, non solo è necessario fare in modo che essi siano "sincronizzati", ma anche che questo risultato sia condiviso da chiunque acceda allo stesso spazio misto (quest'ultimo aspetto avrà in seguito un capitolo a parte) [13]. Per concludere questa descrizione con un esempio, si potrebbe dire che l'interoperabilità è quella caratteristica che permette situazioni come un incontro in cui più partecipanti indossano ognuno un dispositivo per visualizzare degli ologrammi che ognuno può modificare a piacimento.

La **consistenza** è un aspetto se vogliamo incluso nella proprietà precedentemente discussa. Ogni oggetto che si trova nella realtà mista ha uno stato che non deve essere in alcun modo compromesso: dev'essere a tutti gli effetti lo stato tangibile di quell'oggetto, sia che si tratti di un ologramma sia di materia reale. Per questo motivo dev'essere garantita la consistenza degli oggetti che si trovino nella realtà mista. Se comprassimo un mobile di un colore e lo collocassimo in una stanza, non potremmo cerco aspettarci che cambi colore al nostro prossimo accesso nella stessa stanza. Mentre per gli oggetti reali è chiaro che lo stato sia mantenuto consistente, per gli ologrammi non è così. Esempio: sempre durante un una riunione effettuata tramite dei visori di realtà mista, la modifica che un utente opera su un ologramma dev'essere effettiva su tutte le "rappresentazioni" di quell'ologramma. Non deve accadere, cioè, che possano esistere multipli stati possibili allo stesso tempo per un oggetto.

Durante l'introduzione di alcuni concetti sulla realtà mista, è stato sottinteso un aspetto del progetto e del caso di studio portato avanti in questa tesi: il fatto di puntare di ad esperienze **condivise**. A che cosa serve l'interoperabilità o la consistenza se il dispositivo MR è uno solo? Il fatto stesso che l'esperienza provenga da un solo dispositivo fa sì che la sua realtà goda di queste due proprietà. Anzi, si potrebbe affermare che esse hanno senso solo in ambiti condivisi. Ma è possibile davvero parlare di "realtà" mista in un contesto in cui un solo utente, anzi, un solo dispositivo ha accesso ad essa? È probabile che la piena espressione del concetto di realtà mista esista solo in un ambito condiviso.

3.1.2 Tipi di condivisione

Il modo di intendere un'esperienza condivisa per utilizzi concreti può variarne fortemente i requisiti. Ci sono molte caratteristiche che vanno considerate quando si vuole realizzare un'applicazione del genere: per esempio, se volessimo permettere ad ospiti esterni di entrare nella nostra applicazione, potremmo voler condividere solo alcune cose, tenendone nascoste altre. Questo esempio è sufficiente per comprendere che anche piccoli accorgimenti di questo tipo cambiano il modo di pensare l'applicazione.

Prendendo come riferimento Microsoft [7], lo sviluppo per realtà mista cooperativa prende in esame diversi aspetti dell'esperienza stessa:

- Dimensione del gruppo
- Posizione relativa dei partecipanti
- Sincronizzazione delle diverse sessioni
- Ambiente fisico
- Dispositivi utilizzati

Dimensione del gruppo

Il numero di utenti partecipanti ad una sessione condivisa influenza lo sviluppo e il design. È facile immaginare come aumenti la complessità dell'applicazione all'aumentare del numero di utenti. Più utenti sono connessi, maggiori sono i costi tecnici (latenze di rete) e sociali (stanza con tanti avatar): gruppi più grandi di 6 persone sono considerabili gruppi "grandi".

Posizione relativa dei partecipanti

Normalmente la realtà mista è pensata per avere utenti nello stesso posto fisico. In realtà, si tratta di un'esperienza chiamata "colocated", cioè nello stesso posto, ma non è l'unico modo di condividere lo spazio MR. In particolare possono esserci:

- **Esperienze "colocated"**: tutti gli utenti nello stesso spazio fisico
- **Esperienze in remoto**: tutti gli utenti in spazi fisici diversi
- **Entrambe**: alcuni utenti nello stesso spazio fisico, altri in remoto

Sincronizzazione delle diverse sessioni

In un caso **sincrono** gli utenti lavorerebbero contemporaneamente nello stesso ambiente olografico. In un caso **asincrono**, invece, interagirebbero in momenti diversi. È possibile anche misto tra i due. In particolare nel caso asincrono è importante l'aspetto della **persistenza**, ovvero la continua esistenza della realtà mista anche al di fuori di esperienze utente.

Ambiente fisico

Non necessariamente la stessa applicazione MR deve essere eseguita sempre nello stesso spazio fisico. L'applicazione deve sapersi adattare a nuovi ambienti, mantenendo le informazioni relative ad essi. Non solo: in caso di esperienze remote, gli ambienti fisici separati in cui esse stesse si svolgono possono essere molto diversi. È probabile che due ambienti reali non siano mai completamente identici, perciò è meglio dire che l'esperienza potrà svolgersi in ambienti *simili* oppure no.

Dispositivi utilizzati

In questo caso di studio applicativo verrà utilizzato **Hololens 2** come dispositivo, ma è importante specificare che **non è l'unico visore disponibile**. Nel capitolo 2 sono già state analizzate varie alternative a questo visore, ma è sempre da tenere presente che **alcune funzioni disponibili in certi visori, potrebbero non esserlo in altri**. Questo vale specialmente in esperienze multi-utente, in cui i dispositivi sono multipli ed è ancora più facile che non siano dello stesso tipo.

3.2 Le sfide principali

Nello sviluppare un'applicazione che include una grande varietà di interazioni con l'ambiente, per di più svolte anche da entità diverse e possibilmente anche in contemporanea, si incontra immediatamente un importante numero di difficoltà. Se poi si vuole, come in questo progetto, rendere concreta la "realtà mista", quindi far risultare l'interazione tra i due mondi naturale e agile, il problema diviene ancora più complesso.

Tra le entità partecipanti alla cooperazione deve sicuramente avvenire un qualche tipo di scambio di informazioni: come organizzo questo scambio? Devo stabilire sicuramente **quali informazioni "vale la pena" scambiarsi**, per non appesantire la comunicazione mentre allo stesso tempo devo decidere **che cosa è "locale" e cosa no**. Devo scegliere inoltre un modello di comunicazione: è possibile usare un modello noto e già robusto o è necessario un nuovo tipo di architettura?

La realtà è tale anche perché risponde in modo adeguato a determinate azioni, non reversibili. Se la realtà è parzialmente virtuale, **come è possibile mantenere traccia di informazioni riguardanti entrambi gli opposti della realtà mista, fisica e virtuale?**

Tutte queste verranno approfondite in questo capitolo, insieme a questioni più prettamente implementative.

3.3 La sfida dell'interazione

A seconda del livello di interoperabilità (3.1.1) richiesta dall'applicazione, la materia può diventare complessa e articolata: un utente che agisce su un oggetto in maniera concorrente con tutti gli altri che partecipano alla stessa sessione il tutto gestito su dispositivi "untethered" (senza collegamenti) utilizzando la rete come mezzo di comunicazione. Anche in questo caso aiuta l'esempio del videogioco: in molti giochi multi-giocatore è necessaria un grande mole di comunicazioni per far sì che tutti i giocatori siano sincronizzati tra di loro. In che modo questo è possibile?

Molti giochi hanno esigenze diverse: un gioco di scacchi online non ha bisogno di tutte le comunicazioni di un gioco in cui è possibile che più giocatori si muovano in un vasto mondo 3D. Partiamo, quindi, da un caso semplice: un'applicazione dove l'utente vede un ologramma, in questo caso a forma di cubo, ed è in grado di cambiarne il colore tramite un semplice metodo di input, per esempio uno slider. In questa semplice applicazione, se un utente modifica il colore tramite lo slider, tutti i client connessi devono ricevere l'aggiornamento del colore del cubo.

Subito nascono i primi problemi: come si comporta il sistema quando la modifica avviene in contemporanea? Quali informazioni devo trasportare? Quando le trasporto? Come le trasporto? Le risposte potrebbero sembrare banali, ma molto spesso è facile riconoscere, soprattutto nello scalare della complessità del progetto, che in questi sistemi non è così. Prendiamo per esempio la seconda domanda, che cosa trasportare nella comunicazione. In questo esempio la prima risposta sarebbe "il colore del cubo", ma altre alternative potrebbero essere per esempio il "valore dello slider aggiornato" oppure entrambi. Le scelte che si compiono nell'implementazione condizionano poi tutta l'applicazione. In questo caso, infatti, condividendo il valore dello slider entrambi i client potrebbero condividere anche lo stato "grafico" dello slider insieme al colore del cubo, senza che lo slider sia effettivamente un oggetto da condividere (esempio ovviamente molto semplice per far intendere come in sistemi più grandi certe scelte possono avere ripercussioni).

Quando dev'esserci interoperabilità il sistema di input gioca un ruolo fondamentale: nel caso di uno slider non è difficile gestire la trasformazione a partire dall'input, ma nel caso di manipolazioni di posizione, rotazione e scala tramite gestures con le mani come in un'applicazione di realtà mista integrare l'input con la comunicazione via rete potrebbe essere più complesso.

In questo caso di applicazione semplice, quindi, l'unica complessità sta nel meccanismo di condivisione del colore. In un'ottica client-server potremmo pensare in codice un sistema simile a questo:

```
// Esempio di comunicazione del colore

// Avviene nel client che ha modificato lo slider
void SliderUpdateClient(Color newColor) {
    ChangeColorServer(newColor);
}

// Avviene nel server
void ChangeColorServer(Color newColor) {
    this.color = newColor;
    NotifyClients(newColor); // impone ai rimanenti client di
        aggiornare graficamente il colore del cubo
}
```

In questo semplice modello, come nell'architettura classica, è chiaro che la complessità si sposta sul server. In caso di più complesse interazioni l'unica cosa importante è capire **cosa condividere e come**, ma dipende dall'applicazione. Rimane la domanda "quando condividere". Per sistemi real-time la temporalità è un fattore determinante. E' ormai chiaro che le applicazioni MR siano considerabili sistemi real-time: non si può permettere che un ologramma condiviso subisca computazioni in ritardo in un sistema rispetto ad un altro, altrimenti l'applicazione smetterebbe di essere "collaborativa". In questo esempio si potrebbe condividere il nuovo colore ad ogni modifica del valore dello slider, ma questo potrebbe saturare le comunicazioni di rete. In generale i sistemi di questo tipo scelgono di condividere informazioni quando queste producono una **differenza percettibile** a partire dall'input generato. Ad esempio, in questa applicazione del cubo si potrebbe mandare al server la modifica del colore solo quando questa porterebbe ad un cambiamento visibile nel rendering dell'ologramma (ad esempio la differenza superiore ad una certa soglia). In questo caso, però, si rischia di far diventare il sistema **inconsistente**, perciò questo aspetto verrà meglio affrontato in un secondo momento.

3.4 La sfida dell'architettura

Nonostante non siano ancora stati esplorati i framework di rete che si andranno ad utilizzare, si è già accennato ad un tipo di architettura di rete, ovvero quella **client-server**. E' giusto soffermarsi su una riflessione: esiste un'architettura migliore rispetto a tutte le altre per applicazioni MR condivise? In generale la risposta è negativa: multiple soluzioni hanno diversi pro e contro. Il campo della MR è ancora molto giovane ma già diversi tentativi hanno portato a nuove tecniche da considerare per il lato networking. Quan-

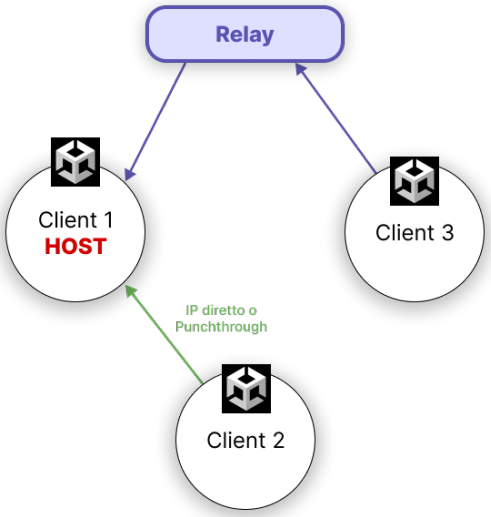


Figura 3.1: Diagramma di un'architettura client-hosted

do queste applicazioni richiedono un così alto livello di condivisione diventa fondamentale chiedersi: quanto è giusto centralizzare l'informazione nella rete?

In occasione del Unite del 2017 [18](una conferenza di Unity Technologies dove vengono riuniti vari "creators" che lavorano utilizzando l'engine omonimo), il fondatore di di Exit Games (autori di **Photon**, uno dei framework che verranno utilizzati), Christof Wegmann, illustra alcuni concetti relativi a giochi multiplayer e, tra questi, l'importanza dell'**architettura di rete** utilizzata per i RTMP (real time multiplayer), in particolare quelli sincroni.

Anche Unity stessa illustra nella documentazione [19] di **Netcode For GameObjects**, un altro degli strumenti che andremo ad utilizzare, varie **topologie e architetture di rete** utilizzabili per giochi multiplayer, insieme ai vantaggi e svantaggi di queste.

Esistono punti in comune ad entrambe e in questa sezione verranno riassunti ed confrontati. E' importante notare che queste architetture non sono "sfere separate", nel senso che queste tecniche possono essere combinate tra loro sfruttando i reciproci vantaggi.

3.4.1 Client Hosted (host authority)

Un **listen server** è un server "hostato" da uno dei client, che quindi interpreta entrambi i ruoli. In pratica, questo significa che tale client avrà il controllo sul mondo di gioco e altri client potranno connettersi ad esso. In questo caso, ipotizzando che l'ambiente sia Unity, lo stesso engine si compor-

ta da **socket server** e tutti gli altri client si devono connettere ad esso per entrare nella sessione.

Come è facile immaginare, un approccio del genere ha diversi problemi in termini di performance, sicurezza e costi. Prima di tutto, la macchina che ospita il server è anche client, di conseguenza si occupa del rendering e dei processi di un utente giocante e questo può **appesantire la computazione** e rendere qualitativamente scarse le prestazioni della rete e della sessione di conseguenza. In molte applicazioni può anche sorgere un problema di **latenza degli altri client rispetto all'host**, che può dare una sorta di vantaggio a quest'ultimo. Dal punto di vista della sicurezza, inoltre, l'host ha il controllo completo del mondo di gioco (essendo lui anche il server) e questo può comportare manomissioni e problemi di sicurezza critici per molte applicazioni. Come se non bastasse, la persistenza del mondo di gioco (sezione 3.6) è difficile o quasi impossibile, dato che lo stato dell'applicazione è strettamente legato all'host, ruolo spesso flessibile. Infine, forse il principale problema resta quello noto come la **migrazione dell'host**. Essendo l'host un client come un altro, nella maggior parte dei casi non dispone di un sistema robusto o costantemente attivo. Se si dovessero verificare disconnessioni dell'host e/o in generale problemi di rete, tutta l'architettura risulterebbe compromessa, costringendo l'ambiente di rete a trovare un nuovo host, *migrando* le comunicazioni e la gestione verso quest'ultimo. Si tratta di un procedimento delicato e complesso, che può verificarsi più volte di quanto ci si aspetti.

Visti gli svantaggi appena presentati, l'opzione client-hosted sembrerebbe essere una strada da scartare automaticamente. In realtà, questo modello si adatta bene a diverse situazioni. Innanzitutto non richiede costi particolari, ne infrastrutture in più dei semplici dispositivi dove vengono eseguite le applicazioni. Funziona, inoltre, nei casi in cui non è importante la latenza tra dispositivi e permette anche sessioni **locali**, cioè con tutti i dispositivi in una rete locale anche senza accesso all'esterno (caso interessante visto che applicazioni in realtà mista colocated possono funzionare in questo scenario).

Questo tipo di architettura è molto estendibile e modificabile. Se i problemi già citati dovessero diventare troppo un motivo di rottura per questa strada, esiste anche l'opzione del server **headless** (senza testa), ovvero aggiungere al modello un **server dedicato**, spostando il concetto di host al suo interno e risolvendo i problemi di prestazioni (infatti il server avrà il motore Unity al suo interno ma non dovrà renderizzare niente perchè è headless appunto) e di sicurezza (e permettendo di realizzare la persistenza, memorizzando nel server le informazioni). Questa opzione, tuttavia, aumenta notevolmente i costi: è possibile affidarsi a servizi esterni o costruire il proprio backend, ma si tratta di un'operazione potenzialmente costosa (deployment, distribuzione delle sedi, auto-scaling etc.) e che non gode di alcuni vantaggi del client-hosted spiegati

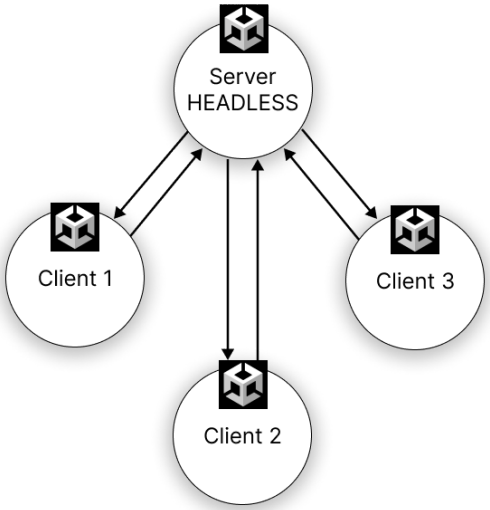


Figura 3.2: Aggiunta del Server dedicato all'architettura

prima.

E' stato, però, tralasciato un aspetto importantissimo dell'architettura client-hosted: come connettere tra loro i client non in reti locali? Essi, infatti, non potendo riferirsi ad un server pubblico, necessitano di modi per stabilire una connessione. Modi comuni per farlo sono ad esempio il **port forwarding**, ovvero il reindirizzamento di una porta pubblica sul proprio router verso il dispositivo con il "listen server". Questa tecnica è poco realistica e spesso irrealizzabile: non sempre i dispositivi hanno accesso al proprio router (caso dei dispositivi mobili), inoltre in caso di host migration diventa complesso se non impossibile. Per finire, è poco sicuro. Un'altra tecnica è ad esempio il **NAT Punchthrough**, ovvero l'utilizzo di strumenti come STUN e ICE che permettano di aprire una connessione diretta tra client "scavalcando" i NAT. Questo metodo, tuttavia, può non funzionare a seconda dei tipi di NAT presenti nella comunicazione e, soprattutto nel caso dei videogiochi multigiocatore, questa tecnica non è popolare proprio perchè si vuole permettere a chiunque di partecipare. Un'ulteriore tecnica di maggior interesse è, invece, il **relay**, che verrà affrontato nella sezione successiva.

3.4.2 Relay e Relay avanzato

Un **relay server** è un server esterno che fa da intermediario tra i vari client. Invece di aprire una connessione diretta tra i vari client, ogni client singolarmente apre una connessione verso il relay server che poi redirige la comunicazione verso il client interessato. Questa tecnica è utilizzabile in una

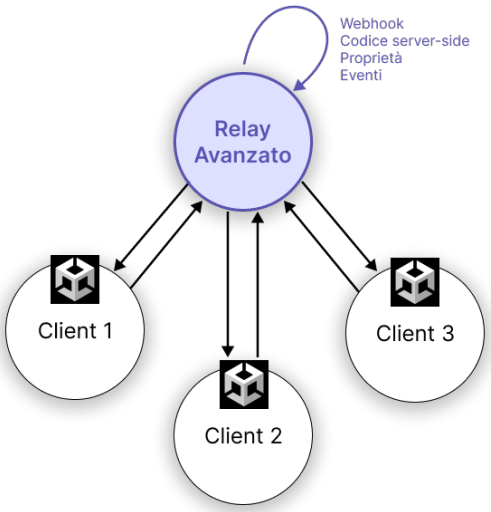


Figura 3.3: Diagramma di un'architettura con relay avanzato

situazione client-hosted per connettere i vari client tra di loro ed utilizzare la tecnica dell'host. Non solo: a seconda della complessità del relay server, si possono **aggiungere funzionalità** per "inspessire" il ruolo di intermediario di quest'ultimo, e intervenire in certe situazioni, ad esempio la disconnessione dell'host (e meglio operare, quindi, con la migrazione dello stesso). Aggiungendo sempre più complessità in questo "polo", questo diventa il caso del così chiamato **Relay Avanzato**, ovvero un'architettura in cui tutti i client sono paritetici (non c'è più il concetto di host) e la parte centrale, il relay, è un server "leggero" (che non richiede molte risorse per funzionare adeguatamente) il cui principale scopo è connettere i client e intervenire nelle comunicazioni tra gli stessi effettuando, se necessario, altre operazioni dipendenti dal tipo di comunicazione mandata (ad esempio, una chiamata http ad un altro backend/-servizio esterno, webhook). Il relay server non contiene, perciò, alcuna istanza di Unity in esecuzione, ma codice server-side leggero e di bassa complessità. Da un certo punto di vista, non contenere Unity può essere uno svantaggio, perché non si esegue codice server-side direttamente all'interno dell'engine, ma allo stesso tempo questa disposizione permette un server più leggero.

Come si è visto, un'architettura con relay avanzato ha molti vantaggi e consiste anche di una soluzione a basso costo e possibilmente facile da prototipare ed estendere. Nonostante i meriti e i vantaggi appena spiegati, tuttavia, possiede anche notevoli svantaggi. Sia per relay normale che relay avanzato esiste un problema di **latenza**: ogni passaggio di pacchetti di rete, infatti, deve passare per un relay server, e da lì andare verso possibili altri server esterni, il che incrementa notevolmente, nei casi peggiori, la latenza. Per questo motivo,

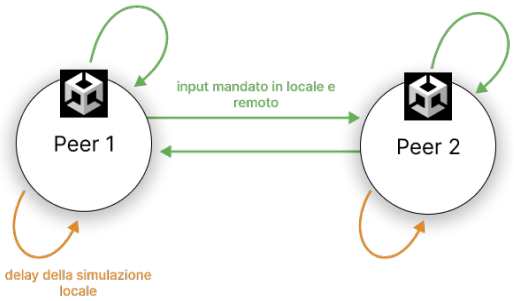


Figura 3.4: Diagramma di un'architettura deterministica

infatti, il relay normale nel caso client-hosted **cerca sempre prima di effettuare una connessione diretta verso l'host**: se questo non è possibile, allora si affida al relay. Nel caso del relay avanzato questo però non è possibile. Non solo: in quest'ultimo caso non è neanche possibile effettuare sessioni locali, perché la presenza esterna del relay server è *sempre* necessaria.

3.4.3 Full Client Authority (o deterministico)

Nonostante non sia stato considerato particolarmente adatto a questa esperienza, si è ritenuto opportuno citare anche questo tipo di soluzione, perchè comune a diversi videogiochi multiplayer. L'idea alla base di questa architettura è quella di **non condividere lo stato dell'applicazione, bensì solo gli input dei suoi partecipanti**. Nell'architettura "deterministica", la simulazione avviene interamente all'interno dei client, che si scambiano tra loro l'input di ciascuno e **aspettano la "risposta"** dei client per confermare l'avvenuta operazione, simulando il resto localmente. Questa tecnica permette ai client di essere totalmente sincronizzati tra di loro, non importa la latenza degli utenti coinvolti. Quello che fa il client può essere riassunto quindi in tre passi ordinati: invia l'input, aspetta la risposta, simula localmente. I vantaggi sono principalmente la **bassa richiesta di banda di rete**, la possibilità di avere un **grande numero di entità coinvolte nella simulazione** e la **facilità con cui si può ripercorrere un'intera simulazione avvenuta precedentemente (replay)**.

I problemi di questo approccio sono, tuttavia, multipli e fondamentali per la realtà mista: primo fra tutti, il **lag della simulazione**. Dato che la simulazione interna viene ritardata per aspettare la risposta degli altri client, uno può vedere la propria azione avere effetti sul mondo di gioco con un po' di ritardo, *assolutamente* da evitare in applicazioni real-time. Inoltre, se uno degli utenti si ritrova con alta latenza di rete, l'esperienza di tutti gli altri peggiora e si adatta a quella dell'utente con peggior latenza. Per finire, le applicazioni vanno

rese deterministiche per funzionare, e **Unity non è deterministico**, motivo per cui il motore della fisica, animazioni e altri aspetti vanno resi deterministici in modo autonomo. Alcuni di questi problemi possono essere mitigati usando un relay per ottimizzare le comunicazione con alta latenza, ma non basta a risolverli del tutto.

3.5 La sfida dell'ancoraggio nello spazio

Fino ad ora non è stato trattato il problema dell'ancoraggio nello spazio, concetto chiave nelle esperienze condivise. Come è stato accennato nella sezione 1.3.3 Spatial Mapping del capitolo 1, il dispositivo MR ha il compito di aggiornare continuamente la sua coscienza del mondo fisico per meglio "simulare" i comportamenti degli ologrammi in risposta alla conformazione dell'ambiente. Come è possibile per il dispositivo comprendere meglio l'ambiente fisico e che cosa sono allora le ancore spaziali?

Una **Spatial Anchor** (oppure appunto ancora spaziale) rappresenta un punto "importante" del mondo fisico di cui il sistema deve tenere conto per "coordinarsi" quando l'esperienza MR arriva ad occupare uno spazio più grande di una piccola stanza. Dato che ogni ancora possiede un proprio sistema di coordinate, l'applicazione potrà piazzare gli ologrammi nei loro sistemi di riferimento per fare in modo che anche se l'utente si allontani l'ologramma non perda la sua posizione grazie al fatto che è piazzato vicino ad un riferimento, l'ancora appunto. Senza ancore, più un ologramma viene piazzato lontano dall'utente e minore sarà la precisione della sua posizione, dato che il sistema fa fatica a capire com'è fatto l'ambiente circostante dopo una certa distanza. Muovendosi verso quell'ologramma, quindi, il visore cercherebbe di migliorare la concezione che ha dello spazio fisico, aggiustandone alcuni aspetti ma causando irrimediabilmente uno sfasamento nella posizione degli ologrammi, che apparirebbero in posizioni mutevoli rispetto al mondo o tra di loro.

Come vedremo, le ancore tornano utili anche in tanti altri contesti.

3.6 La sfida della persistenza

La persistenza è definita come la continua o prolungata esistenza di qualcosa. Un gioco online è persistente quando il mondo di gioco continua ad esistere dopo che il giocatore locale ha concluso la sua sessione. Altri giocatori stanno ancora usando il mondo di gioco persistente, e azioni significative possono avvenire e alterare la partita locale di un utente quando la sua sessione sarà ripresa successivamente.

In una realtà mista, l'esistenza degli ologrammi dev'essere garantita a prescindere dalla presenza o meno di client connessi. In un applicazione dev'essere possibile salvare lo stato attuale degli oggetti, ma quando questo stato si riferisce a qualcosa di condiviso, le cose si complicano. Non solo: è necessario mantenere anche **informazioni sul mondo reale**, visto che l'applicazione lo richiede, e associarle alle informazioni sul mondo virtuale.

Prendiamo d'esempio un applicazione locale: un utente potrebbe spostare gli ologrammi, eliminarli, aggiungerne di nuovi, modificarne altre caratteristiche ecc. Quando l'utente esce dall'applicazione, il dispositivo salva lo stato attuale dell'applicazione in una specie di file/database per poi ritrovare l'applicazione in uno stato esattamente identico alla prossima apertura.

In un ambiente condiviso la gestione dev'essere diversa: gli utenti possono collegarsi, scollegarsi, le modifiche possono avvenire da parte di entità diverse e, soprattutto, **può non esserci neanche un client connesso**, portando la memorizzazione altrove, per esempio nel server. Come spiegato in 3.4, però, non è affatto detto che esista un server liberamente accessibile: ad esempio, in un'architettura distribuita o full-client cessa di esistere un entità centralizzata, quindi i client "rimbalzano" semplicemente l'informazione tra loro. Dove può trovarsi un'informazione che dev'essere persistente?

Partiamo dal modello classico: client-server. In questo caso il server conserva tutte le informazioni e resta sempre attivo pronto a servire i client fornendo loro lo stato dell'applicazione con tutti i suoi ologrammi. Come è possibile rendersi conto, in ambito pratico un'architettura così rigida non serve bene i requisiti di un sistema simile ai real-time, soprattutto per questioni di sincronizzazione (ma verrà comunque esaminato nei capitoli successivi). Se, invece, ci occupiamo di un'architettura distribuita, la soluzione potrebbe essere più complessa. Ogni utente ha una copia locale del mondo? L'ultimo ad uscire mantiene la copia e la condivide quando il prossimo utente entra?

Ricordando la struttura dei sistemi distribuiti, anche questi sistemi fanno uso di **middleware**. Un middleware è un software che funge da intermediario tra diverse applicazioni: si utilizza spesso quando il campo d'implementazione può essere molto vario e software diversi possono essere usati assieme per risolvere un problema. In questo caso, alcuni strumenti usati nella comunicazione di rete per la MR condivisa sono proprio dei middleware (es. Photon) e permettono di utilizzare altri servizi in combinazione con essi, proprio per salvare informazioni. E' ovvio, però, che in quel caso l'architettura scelta per gestire i client in rete potrebbe subire cambiamenti, adattandosi per appoggiarsi ad un'entità esterna.

Esistono altre informazioni da mantenere oltre gli ologrammi? Se noi volessimo, ad esempio, piazzare un ologramma di un vaso sopra un tavolo fisico, al nostro prossimo rientro sapremmo che esisteva un vaso, ma come mante-

nere la conoscenza del tavolo fisico? Altre informazioni vanno mantenute per avere un rapporto 1:1 con il mondo fisico, ovvero le **ancore**. Le ancore come gli ologrammi devono essere persistenti: devono essere locali o condivise? La risposta dipende dal tipo di esperienza: se gli utenti si trovano nello stesso ambiente (colocated) è probabile che vadano condivise, altrimenti può variare a seconda del tipo di applicazione. Soluzioni per questo tipo di persistenza sono ad esempio **Azure Spatial Anchors**, prossimamente introdotto, che permette di mantenere i riferimenti spaziali (le ancore) persistenti, salvandole in cloud.

3.7 La sfida della rappresentazione dell'utente

La rappresentazione di altri utenti che partecipano alla stessa sessione collaborativa è un altro aspetto da approfondire. Abbiamo visto che sono possibili diversi tipi di esperienze, alcune in cui non è detto che gli stessi utenti si trovino nella stessa stanza. Come gestire graficamente, quindi, la presenza di altri utenti nell'applicazione?

In generale chiamiamo **avatar** un ologramma che rappresenta un altro utente umano all'interno dell'applicazione. Come vedremo, si potrà definire all'interno dell'applicazione una specie di **origine dello spazio condiviso**. In caso di esperienze in remoto servirà una coscienza della relativa posizione *rispetto all'origine* per poter collocare degli avatar. Di fatto, non è importante per un'esperienza remota, assistere allo stesso spazio reale (fisicamente impossibile) perciò l'importante è avere la stessa coscienza degli uni rispetto agli altri e rispetto agli ologrammi della parte virtuale. Esempio: se un utente A vuole spostare un ologramma verso un altro utente B, mentre si trovano in luoghi fisici diversi, l'importante è che A sappia riconoscere in che posizione B sta vedendo l'ologramma, cioè quale sia la posizione di B nel mondo virtuale rispetto alla posizione di A nel mondo virtuale.

In un'esperienza colocated la rappresentazione è più complessa. L'ologramma dell'avatar deve coincidere con la persona che rappresenta, originando strane fusioni tra ologramma e utente reale. Non solo: visto che la concezione dello spazio condivisa (spiegata più avanti) si basa su stime o comunque su procedure non precise al 100%, l'ologramma dell'avatar apparirebbe "spostato" o comunque piazzato in maniera storta rispetto alla persona. In questo caso si possono adottare fondamentalmente due strategie. La prima, non mettere nessun avatar e lasciare che il riconoscimento avvenga tramite la vista attraverso gli ologrammi dell'altra persona. In questo caso, tuttavia, ci si accorge che potrebbe diventare difficile capire quali azioni sta compiendo l'altro utente, con quali ologrammi sta interagendo o semplicemente capire se è allineato

rispetto al mondo virtuale. Secondo, utilizzare un avatar poco invasivo che appare occasionalmente e che è strettamente sufficiente a capire le intenzioni dell'utente.

3.7.1 Come realizzare un avatar?

L'ologramma che avrà funzione di avatar può variare: nel caso di **Microsoft Mesh**, l'applicazione/piattaforma importantissima e riferimento di questo progetto che permette una moltitudine di esperienze condivise, l'avatar è una raffigurazione stilizzata simile alla persona che rappresenta, dal torso fino alla testa e che fa comparire delle mani in caso di manipolazione di ologrammi da parte della stessa persona. In altri casi può essere semplicemente un icona 3D. In generale, le app sviluppate tendono ad una rappresentazione **non realistica** della persona, questo per evitare l'effetto "uncanny valley" (ovvero l'inquietudine generata nelle persone di fronte a robot o automi antropomorfi troppo somiglianti alla figura umana) e, logicamente, limitazioni tecniche (ologrammi troppo complessi e con troppi poligoni). In generale, più l'esperienza include un maggior numero di persone più si rende necessario distinguerle tramite avatar più accurati ma non troppo complessi, mentre più l'esperienza si rende colocated (ovvero la maggior parte delle persone si trova nello stesso luogo fisico) e più gli avatar assumono un stile più minimalista.

3.8 La sfida degli ambienti diversi

A volte, un'esperienza condivisa può non dipendere dall'ambiente fisico in cui si svolge. Finora è stato spiegato il concetto di realtà mista come qualcosa che ha un rapporto 1:1 con il mondo reale: se un ologramma è stato piazzato in un certo punto fisico, se l'utente ritorna in quello stesso punto l'ologramma si troverà di nuovo lì. Altre volte, tuttavia, l'ambiente virtuale deve potersi ad adattare, per motivi pratici, a diversi ambienti fisici, **migrando gli ologrammi già esistenti**.

Esempio principale di questi casi è l'esperienza remota: gli ologrammi persistenti devono essere mantenuti in posizioni tali da non snaturare ciascuno degli ambienti fisici diversi in cui viene eseguita l'applicazione condivisa. Di nuovo tornano d'aiuto le **ancore**: ogni ancora dev'essere piazzata in un luogo che è possibile considerare l'origine del sistema di coordinate (un punto centrale della stanza, per esempio). L'ancora in questione va piazzata dall'utente per garantire che il nuovo luogo sia "calibrato" per l'accoglienza degli ologrammi, dopodiché la migrazione avviene tenendo conto dell'ancora come punto di riferimento. Nel caso in cui l'ambiente di partenza sia grande abbastanza da

necessitare di più ancore, si procede con il piazzamento di ogni singola ancora, pur tenendo conto che in questo caso l'applicazione non si presta bene, probabilmente, allo spostamento in un altro luogo.

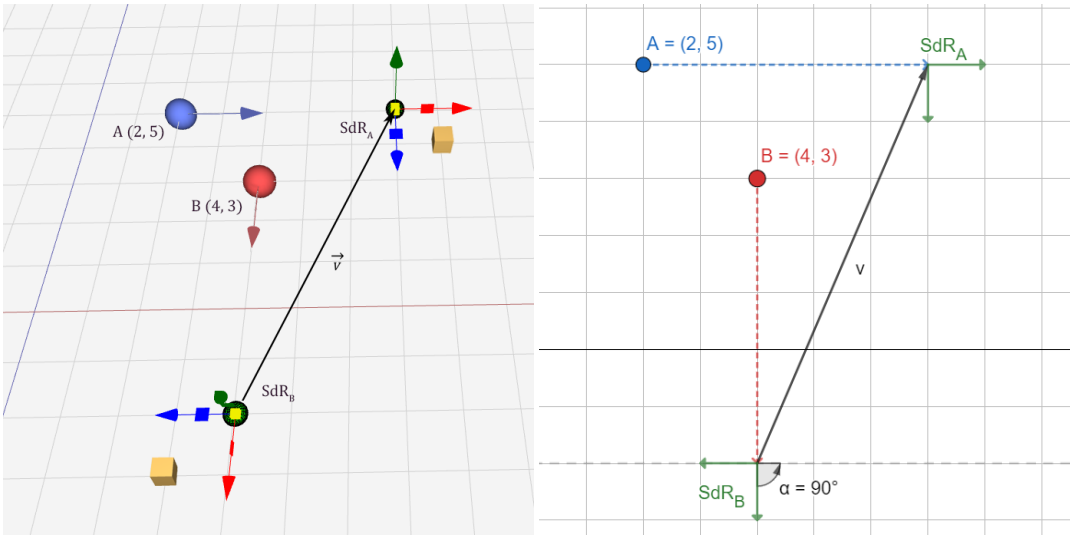
Questo aspetto non è stato esplorato nel caso di studio, in quanto non strettamente legato solo alle esperienze condivise e non punto cardine del progetto.

3.9 La sfida dell'ownership

Come vedremo, molti framework per la comunicazione di rete gestiranno, tra le varie cose, anche l'**ownership degli oggetti in rete**. Cosa si intende per ownership?

Ogni oggetto in una gestione unificata dell'ambiente condiviso deve possedere un identificativo e la **proprietà da parte di un client**. Lo scopo del primo è ovvio: in un'applicazione dove esistono degli oggetti replicabili, esistenti sia in forma "locale" che condivisa tra tutti gli utenti, è necessario un modo per distinguere gli oggetti in maniera univoca in tutti i client. Lo scopo dell'ownership potrebbe essere, invece, meno chiaro: in generale si può dire che l'ownership serve ad evitare potenziali bug e problemi che si possono verificare in ambienti condivisi attraverso una rete, in cui possono avvenire modifiche contemporanee o ritardi nell'effettiva "apparizione" di modifiche. L'ownership è, letteralmente, la "proprietà" di quell'oggetto da parte di uno dei client o del server. Chi opera su un oggetto, può vedere ristrette alcune operazioni a causa della mancanza di ownership sullo stesso.

Se un oggetto, tuttavia, necessita di essere modificato agilmente da più client (interoperabilità), l'ownership dev'essere trasferita continuamente per permettere a chi sta compiendo la modifica di avere prima di tutto la proprietà dell'oggetto. Non tutti i framework, tuttavia, funzionano allo stesso modo: in alcuni casi l'ownership potrebbe essere **fissa**, e le modifiche sull'oggetto potrebbero avvenire solo tramite delle chiamate di rete specifiche, atte a operare su un server o un client che possiede già l'ownership. In altri casi, invece, essa potrebbe essere modificabile tramite delle **richieste di ownership**, che però rendono la modifica dinamica scomoda per lo sviluppatore. Infine, l'ownership potrebbe essere trasferita automaticamente, rendendo il framework più ad alto livello. Vedremo come alcuni possiedono tutte e tre queste possibilità (Photon) mentre altri solo alcune (Netcode for Gameobjects).



(a) In 3D (b) In 2D

Figura 3.5: Diverso sistema di coordinate

3.10 La sfida della concezione di spazio

La concezione dello spazio virtuale di due dispositivi scollegati differisce sempre di una quantità più o meno grande a seconda delle condizioni di partenza di un'applicazione. Quando un visore come Hololens avvia un'applicazione, comincia a scannerizzare l'ambiente e piazza l'origine dello spazio di coordinate in un punto dello spazio fisico che dipende dalla posizione iniziale del visore e della direzione in cui è puntato. E' difficile che due visori riescano ad allinearsi in questo modo quando avviano la stessa applicazione (oltre ad essere scomodo e insolito), di conseguenza è necessario un modo per far si che **entrambi vedano l'origine nello stesso punto fisico**.

Immaginiamo di trovarci in una stanza che, per comodità in questo esempio, dotiamo di un sistema di coordinate. Chiamiamo A e B due utenti diversi che si trovano in un punti distinti all'interno della stanza, rispettivamente (2,5) e (4,3). B è girato di 90 gradi rispetto ad A. Entrambi fanno partire l'applicazione, prima A e poi B. Successivamente il visore di ciascuno crea l'origine del sistema di riferimento dello spazio *virtuale* che userà per collocare gli ologrammi all'interno dell'applicazione. In questo caso, per semplicità, mettiamo che il visore collochi il sistema di riferimento a 5 unità di distanza dalla posizione dell'utente (cioè anche del visore). Ecco che non solo i sistemi di riferimento sono traslati, ma addirittura anche ruotati l'uno rispetto all'altro. Ora, se dovessimo piazzare, ad esempio, l'ologramma di un cubo giallo, la posizione *relativa* del cubo nei sistemi di riferimento sarà la stessa ma, dato

che si trovano nello stesso ambiente fisico, i due utenti vedranno lo stesso ologramma in due posizioni completamente diverse. Non solo: se gli utenti possiedono degli avatar, la posizione che ognuno vedrà dell'avatar reciproco sarà completamente sbagliata.

Per portare entrambi i sistemi di riferimento nello stesso punto fisico posso, ad esempio, trovare un vettore $\vec{v}$ insieme ad una matrice di rotazione $R(\alpha)$ (in questo caso d'esempio, assumendo che i due visori non si trovino ad altezze diverse) che mi trasporti uno dei sistemi di riferimento nella stessa posizione fisica dell'altro scelto come "master". In questo modo lo "slave" dei due sarà forzato ad adattarsi alla concezione fisica dello spazio dell'altro. Questo metodo, tuttavia, si rivela piuttosto scomodo e poco sensato: come calcolo, innanzitutto, $\vec{v}$ e R? Dovrei in qualche modo sapere la posizione relativa degli utenti. Inoltre in caso di più utenti che partecipano, ognuno dovrebbe fare riferimento ad un master, quindi tenere traccia di chi si è "sincronizzato" oppure no. Per finire, non si può applicare in caso di esperienze remote.

Un metodo migliore consiste non nel "aggiustarsi a vicenda" tra visori, quanto piuttosto nel aggiustare ogni singolo client fissando un punto nello spazio. Quando abbiamo parlato di ancore spaziali, abbiamo accennato ad una loro importantissima funzione: usandole come punto di riferimento per sistemi di coordinate condivisi, possiamo fare in modo che **un punto nello spazio virtuale di ogni client corrisponda immediatamente ad un punto nello spazio fisico**. Quando un visore riconosce il punto fisico in questione, ricalcola tutto il sistema di riferimento interno basandosi sul fatto che il punto fisico è accurato.

Per ottenere questo effetto, è necessario un modo per riconoscere un punto fisico con precisione sufficientemente elevata e in maniera semplice. Uno può essere quello dei famosi e ormai molto utilizzati **QR Code** (o marker in generale): se un marker statico che si trova in un luogo viene inquadrato da tutti i visori, lì verranno fissati gli origini dei sistemi di riferimento. Questo può funzionare anche per esperienze remote: non è infatti necessario che il luogo preciso dove si trova il QR code sia lo stesso in questo caso, l'importante è fissare un origine comodo per tutti gli utenti. Il principale svantaggio di questo metodo, tuttavia, è il bisogno del codice cartaceo o non e il doverlo inquadrare manualmente.

Un altro modo che sfrutta le capacità di un visore come Hololens è quello di **Azure Spatial Anchors**: ogni ancora è persistente e associata a delle informazioni sul mapping fisico operato dal visore, di modo che ogni ancora si trovi sempre nello stesso punto anche per altri visori che accedono alle stesse ancore in cloud. Questo metodo è automatico e veloce, ma potrebbe offrire una scarsa precisione.

3.11 Onerosità di calcolo e sistema distribuito

Dopo quanto spiegato ed introdotto, un'osservazione sottile potrebbe essere fatta a riguardo: quanto effettivamente gli HMD (head-mounted display) e i visori in generale dipendono da servizi e piattaforme esterne? La struttura stessa dell'esperienza condivisa presuppone un ambiente distribuito, così come la persistenza della realtà mista necessita di infrastrutture per memorizzare dati virtuali associati a riferimenti fisici. La presenza di un servizio come Azure fa intendere come Microsoft abbia già intuito, come lanciatore del prodotto, le necessità di un prodotto come Hololens. Gli stessi algoritmi di visione artificiale così come tutte le tecniche basate sul machine learning per scannerizzare, mappare, "capire" la realtà fisica non sono affatto leggere computazionalmente parlando.

Hololens, dispositivo scelto per questo progetto, è nato come dispositivo che si basi sul **cloud computing**, per far fronte alla sua bassa capacità di calcolo e per necessità di implementazione. Anche Azure Spatial Anchors rappresenta una soluzione cloud per il problema della ancore persistenti e condivise. Ma cosa comporta un simile scenario?

Prima di tutto, costringe la maggior parte degli sviluppatori a pagare per contratti e servizi che stanno su un cloud o a sfruttare una propria infrastruttura (anche se in certi casi non è possibile). Secondariamente pone l'utilizzo di Hololens come dispositivo funzionante unicamente se connesso in rete, il che rappresenta una debolezza soprattutto in caso di problemi e malfunzionamenti. Si potrebbe argomentare questa critica sottolineando il fatto che l'esperienza condivisa può effettivamente funzionare solo se c'è connessione di rete. Ma allo stesso tempo non esistono solo applicazioni per esperienze condivise. Infine ci sono ovviamente da considerare le latenze dovute alla rete.

3.12 Privacy e Sicurezza

Essendo Hololens un dispositivo mobile che accede con connessioni a internet illimitate e con canali di comunicazione aperti, il problema della privacy diventa subito rilevante. Soprattutto lo spatial mapping genera pensieri allarmanti: se un dispositivo mappa l'ambiente circostante tramite la visione artificiale, questi dati possono essere usati da hacker per identificare la struttura di una stanza e tutto ciò che sta al suo interno. Non solo: è stato provato [14] che sia possibile mappare volti di persone e associarli con profili di social networks pubblici per associare volto a nome ed altri dati personali. I visori per la realtà mista sono per lo più HMD, quindi, avendo delle telecamere, rischiano di "emettere" dati sensibili.

Per quanto concerne gli aspetti di sicurezza appena citati, è il visore stesso, nella maggioranza dei casi, che deve garantire privacy e mostrare affidabilità, e Hololens 2 in particolare dimostra miglioramenti sotto questi aspetti. Nel caso delle esperienze condivise, invece, il problema è un altro. In questa esperienza può essere di maggiore interesse valutare aspetti relativi alla **protezione delle sessioni condivise**, la restrizione di certe operazioni sugli ologrammi o visibilità ridotta per certe informazioni e dati visivi. Tuttavia, non verranno particolarmente esplorati in questo progetto.

Capitolo 4

Implementazione di Esperienze Condivise in Realtà Mista: confronto fra architetture di riferimento

In questo capitolo verranno implementate due diverse soluzioni per l'applicazione presentata all'inizio del capitolo 3. Questi due diversi progetti perseguiranno lo stesso obiettivo, ovvero quello di creare un'esperienza condivisa in MR che permetta di condividere degli ologrammi sia nel loro stato, che in questi esempi sarà il **colore** dell'ologramma, sia per caratteristiche fisiche, ovvero il Transform dell'ologramma (posizione, rotazione e scala). Queste applicazioni, con dispositivo target Microsoft Hololens (analizzato nel capitolo 2), dovrà permettere agli utenti di "sincronizzarsi" facilmente, sia in esperienze colocated che remote. Dal punto di vista implementativo, verranno testate due diverse tecnologie per il lato networking, soggetti del confronto di questo capitolo. Queste sono **Netcode for GameObjects** e **Photon**.

Verranno prima affrontati certi loro aspetti in comune, secondariamente avranno una sezione dedicata a ciascuno. In questo capitolo verranno esaminate queste tecnologie inizialmente da un punto di vista di "gaming", per le similarità già discusse, e successivamente applicate al contesto della realtà mista.

Per questo progetto, un grande esempio che vale la pena citare è **Microsoft Mesh**, una piattaforma per la collaborazione e condivisione sviluppata da Microsoft. Punta a dispositivi olografici per VR, AR e MR, unificando il tutto su più piattaforme come smartphone, laptop o visori veri e propri. L'applicazione che verrà sviluppata prende ispirazione proprio da questo progetto di Microsoft, che si prevede assumerà sempre più importanza in futuro per

l'extended reality in generale.

4.1 Netcode for GameObjects

Netcode for GameObjects è una libreria di alto di livello per Unity che permette di astrarre la logica di rete. Detto in parole povere, permette di mandare GameObjects e dati del mondo virtuale tra sessioni di rete diverse e a multipli utenti nello stesso tempo. Il suo scopo è quello di facilitare lo sviluppo di giochi multiplayer senza che lo sviluppatore si debba preoccupare dei protocolli di basso livello e di framework di rete.

Prima di Netcode, Unity ha lavorato a **UNet**, una versione precedente, ormai deprecata, e a **MLAPI**, una libreria considerata "mid-level", anch'essa deprecata. Netcode, come i suoi precedenti, lavora seguendo un'architettura **client-hosted** (vedi 3.4.1), ma supporta anche altre tecniche come il relay.

4.1.1 Unity Game Services

In realtà, Netcode è inserito all'interno di un contesto più grande. Come vedremo per Photon, sono diversi i servizi che Unity come piattaforma cloud ha deciso di mettere a disposizione. Pur essendo, di fatto, client-hosted, Netcode supporta anche meccanismi come il relay. Unity, infatti, mette a disposizione **Relay**, un servizio che esiste per connettere i giocatori che usano Netcode sfruttando appunto la tecnica del relay (diventando quindi simili a Photon). Insieme a Relay, Unity ha tanti altri strumenti che aiutano a scalare con il videogioco che si sta creando, per esempio **Lobby**, che serve per connettere i giocatori *prima* che la sessione di gioco inizi (nelle cosiddette lobby appunto). Esistono anche collaborazioni tra Photon e Unity, come ad esempio **Multiplay**, un servizio di game hosting in cloud. La scelta, insomma, tra Photon e Netcode si deve basare anche sull'ecosistema che entrambi si portano dietro.

4.1.2 Network Manager

Il funzionamento di un'applicazione con Netcode ruota intorno ad un GameObject, il **Network Manager**: esso contiene diversi script di libreria tra cui ovviamente il **NetworkManager** e lo **Unity Transport**. NetworkManager è un componente unico (non ce ne devono essere altri su GameObject diversi) che serve a gestire l'avvio della sessione di rete, insieme ad altri parametri della stessa e di Netcode. Tra gli aspetti più importanti ci sono i **NetworkPrefabs**, ovvero dei prefab dotati di componenti particolari (più avanti spiegati) che saranno generabili (spawn) dal manager e condivisi in rete. Aspetto fondamentale di questo componente è che **tramite esso si può inizializzare**

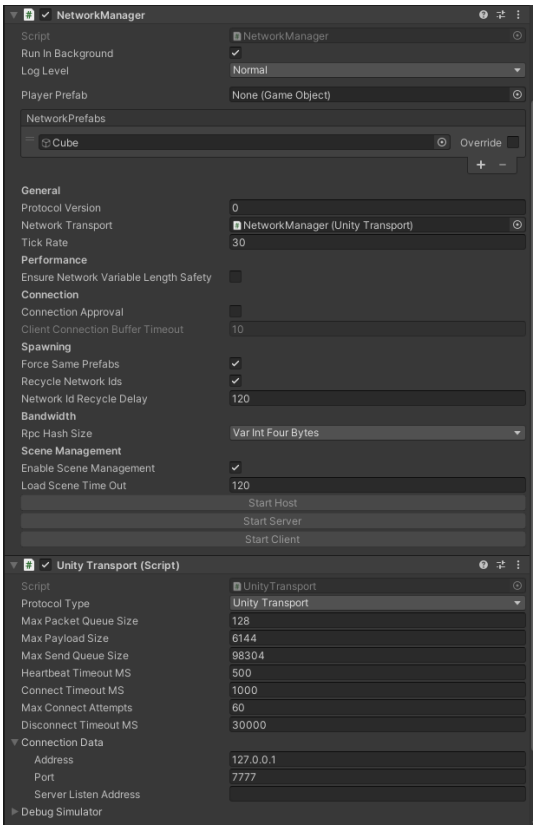


Figura 4.1: NetworkManager e Unity Transport visti dall'Inspector di Unity

la connessione come host o come client (specificando anche i dati della connessione, cioè indirizzo IP e porta). Nella nostra applicazione questo verrà scelto grazie a dei pulsanti cliccabili dall'utente.

Altro componente che fa coppia con il precedente è Unity Transport. Esso è forse ancora più importante, perché specifica **come avviene la connessione tra l'host e gli altri client**, per esempio regolando l'ordine dei pacchetti di rete, controllo della congestione o assicurando l'integrità dei dati. Netcode permette all'interno di questo componente di scegliere il tipo di transport che desideriamo: è possibile anche definirsi un proprio transport. Nel nostro progetto, verrà utilizzato il transport di default fornito da Netcode, chiamato anch'esso **Unity Transport (UTP)**, basato su sockets UDP.

4.1.3 Network Object

Tutti gli strumenti spiegati in seguito, il sistema RPC, lo spawning di oggetti e le Network Variables si basano su GameObject che hanno due componenti particolari:

- NetworkObject
- NetworkBehaviour

Ogni oggetto che va condiviso in rete possiede un NetworkObject e almeno un NetworkBehaviour. Anche altri componenti fondamentali come il NetworkTransform necessitano di NetworkObject per funzionare.

A che cosa serve? Ogni NetworkObject esiste per identificare univocamente quell'oggetto in rete. Ad ogni NetworkObject, infatti, viene assegnato un "global object ID hash" al momento dello spawn: esso può essere usato successivamente per vari modi, ad esempio da un peer che può dire "invia questa chiamata RPC all'oggetto con ID 180".

I NetworkBehaviour sono invece script di varia natura che possono dare all'oggetto vari comportamenti e aggiungere logica di rete allo stesso. In concreto, NetworkBehaviour è una classe astratta che estende MonoBehaviour di Unity e che viene estesa da altri componenti più specifici (es. NetworkTransform).

Ownership

Quando si parla di NetworkObject, entra in gioco il discorso della **ownership** già affrontato in precedenza (sezione 3.9). In Netcode, infatti, essa viene calcolata proprio a livello di NetworkObject, dove ognuno appartiene ad un client o all'host. Come cita la documentazione di Netcode for GameObjects [19] "Netcode è server authoritative, di conseguenza quest'ultimo controlla (o meglio, è l'unico autorizzato) lo spawn o de-spawn di NetworkObject". Per questo motivo metodi come `Spawn()` di NetworkObject permettono di generare un oggetto assumendo l'ownership server-side. Questa "proprietà", tuttavia, può anche essere cambiata tramite i metodi `ChangeOwnership()` e `RemoveOwnership()` sempre di NetworkObject, e vedremo come questo sarà fondamentale.

PlayerObjects

Interessanti sono anche i PlayerObjects, registrabili nel NetworkManager, che sono essenzialmente dei NetworkObject la cui istanza, però, è **unica per client**, cioè che un client non può generare più volte il PlayerObject, ma solo una volta (almeno finché ne possiede l'ownership). Questo può tornare comodo per creare gli **avatar** (sezione 3.7), ovvero GameObjects che sono unici per client e che lo rappresentano nello spazio.

4.1.4 NetworkVariable e RPC

Le NetworkVariable, o più semplicemente variabili di rete, sono semplicemente dei contenitori per delle proprietà (variabili) il cui accesso viene effettuato in rete. Grazie ad esse è possibile sincronizzare lo stato di una variabile tra server e tutti i client. Nel nostro caso, il **colore del cubo** sarà mantenuto tramite una NetworkVariable, di modo che tutti i client lo condividano. E' importante notare che, di default, **solo il server può modificare il valore di una NetworkVariable**. Questa impostazione può anche essere regolata in modo che chi ha l'ownership possa modificarlo.

Il concetto di RPC (Remote Procedure Call) esiste anche al di fuori di Netcode, nella produzione di software in generale, particolarmente nei sistemi distribuiti. Una RPC consiste nella chiamata di una procedura o subroutine attivata su un diverso spazio di indirizzi (tipicamente su un altro computer in una rete condivisa) da quello chiamante. Dal punto di vista del codice si comporta come una normale procedura in modo che il programmatore non debba preoccuparsi effettivamente dell'implementazione dell'interazione remota. Quando un oggetto effettua una RPC, l'SDK si prenderà carico della chiamata e invierà anche i dettagli del "chiamante" sulla rete.

Netcode ha due varianti delle RPC: **ServerRpc** e **ClientRpc**. La prima permette di eseguire la procedura dentro il server/host, la seconda all'interno di una selezione di clients (di default tutti i client). In Netcode è possibile dichiarare una RPC utilizzando l'annotazione `[ServerRPC]` o `[ClientRPC]` seguita dalla procedura (che a sua volta deve avere un identificatore che finisce con "ServerRpc" o "ClientRpc" a seconda dell'annotazione usata).

Possiamo usare quello visto finora per effettuare uno scambio di ownership di un NetworkObject quando ne sentiamo il bisogno. L'implementazione risulta simile a questa:

```
// ChangeOwnershipScript.cs

public void TakeOwnership()
{
    // Altre azioni da eseguire nel client
    // ...
    // Richiedo ownership
    ChangeOwnershipServerRpc();
}
```

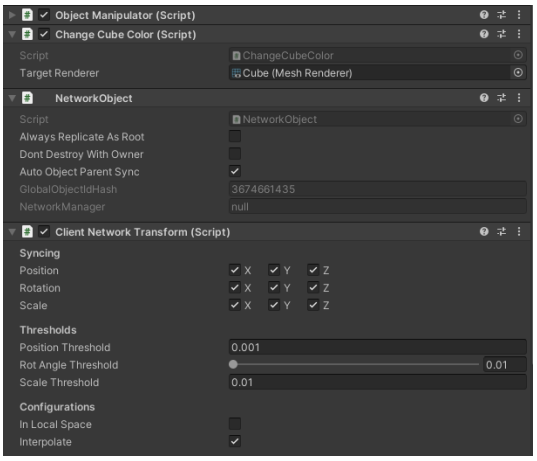


Figura 4.2: Script presenti nel prefab di un ologramma condiviso

```
[ServerRpc(RequireOwnership = false)]
public void ChangeOwnershipServerRpc(ServerRpcParams serverRpcParams
    = default)
{
    // Altre azioni da eseguire nel server
    // ...
    // Affido ownership
    GetComponent<NetworkObject()
        .ChangeOwnership(serverRpcParams.Receive.SenderClientId);
}
```

La proprietà `RequireOwnership` serve a specificare se questa chiamata RPC può essere effettuata solo da client che hanno l'ownership sul NetworkObject associato al NetworkBehaviour dove la procedura viene usata. In questo caso viene messa a `false` per ovvi motivi (la ownership è proprio quella che stiamo cercando di ottenere, quindi questo metodo non deve averla come requisito). Insieme ad essa troviamo anche un'altra incognita, `ServerRpcParams`: esso è un parametro *opzionale*, che è molto utile per capire chi (quale client) ha effettuato la chiamata. Infatti viene usato per trasferire l'ownership passando come parametro a `ChangeOwnership` proprio il client ID del chiamante.

E' opportuno evidenziare che in questo codice non esistono particolari misure di sicurezza: qualunque client può richiedere e ottenere l'ownership dell'oggetto in questione senza problemi.

4.1.5 Network Transform

Netcode fornisce un importantissimo componente, fondamentale per questo progetto, ovvero **NetworkTransform**. Normalmente in Unity posizione, rotazione e scala degli oggetti sono contenuti in un componente chiamato **Transform**: la "versione network" consente di sincronizzare questo componente nei NetworkObject tra tutti i client. In Netcode, infatti, il Transform viene sincronizzato solo quando l'oggetto viene generato, mentre questo permette di continuare la sincronizzazione anche durante tutta la sessione.

Questo componente possiede molte proprietà: una è ad esempio quella delle **thresholds**, soglie di modifiche al Transform dell'oggetto entro cui la sincronizzazione non avviene. Un'altra proprietà fondamentale è l'**interpolazione**: senza di essa, visto che i dati su posizione, rotazione e scala non vengono sincronizzati in maniera "continua" bensì tramite buffer scansionati da dei "tick", l'interpolazione permette alla trasformazione di essere più dolce e continua. Netcode sceglie di implementare questa tecnica solo client-side: per movimenti più regolari anche su host e server, viene lasciata agli utenti la scelta di implementarla anche server-side.

ClientNetworkTransform

Come cita la documentazione di Netcode: "NetworkTransform sincronizza sempre le posizioni dal server verso i client e cambiamenti di posizione nei client non sono permessi". Qui torna il discorso dell'ownership: per proteggere il concetto di ownership server-side, NetworkTransform concentra i cambiamenti di posizione solo all'interno del server. Questo significa che se un client volesse innescare una modifica in un oggetto, **non potrebbe farlo neanche avendone l'ownership**, perché su questo componente i cambiamenti avvengono solo dal server verso i client. L'unico modo sarebbbe quindi quello di inviare dati al server (per esempio tramite una RPC) che descrivano la modifica da operare, e successivamente operarli all'interno del server.

Nonostante questo modo di agire possa andar bene per progetti più generici, dove il movimento degli oggetti avviene tramite script personalizzati che prendono come input tastiera o mouse (e che quindi invece di modificare direttamente il Transform devono solamente inviare dei dati al server per attivare lì la modifica) non è assolutamente comodo per progetti con altri tipi di input, come in questo caso: l'interazione avviene, infatti, tramite script esterni provenienti da MRTK, di conseguenza sarebbe troppo complesso intervenire nella modifica del Transform per informare il server della modifica che il client vuole operare.

Come spiegato in 4.3, sarà possibile rendere tutto questo più semplice utilizzando un altro script fornito da Netcode, ovvero **ClientNetworkTransform**.

Come suggerisce il nome, questo componente rende possibili manipolazioni **da client verso server**, solo se tale client possiede (ha l'ownership) l'oggetto.

4.1.6 Controllo dell'ologramma

Per quanto riguarda gli altri dati, posizione, rotazione e scala, abbiamo già script come il ClientNetworkTransform che ci permettono di condividerli. Come è stato già descritto, la variabile "stato" per gli ologrammi condivisi sarà il **colore** degli ologrammi stessi. Come possiamo condividere il colore? Come spiegato nella sezione 3.3, ci conviene condividere una variabile più semplice del colore, per esempio il **valore di uno slider che rappresenta l'input dell'utente**, quindi condividere l'input del client piuttosto che lo stato. Per questo motivo, possiamo sfruttare le **NetworkVariable** di Netcode, contenenti un numero floating point (float) da 0 a 1 rappresentante la percentuale dello slider. Successivamente, impostiamo la funzione *callback* dell'evento dello slider "Value Updated" (valore aggiornato) ad una funzione del cubo che cambi il colore. Siccome **solo il server può modificare il valore della variabile**, è necessario che tutti i client notifichino il server del valore aggiornato. In questo caso ci aiutano le **RPC**, in modo simile a quanto visto per l'ownership (sezione 4.1.4). Mettendo tutto assieme, il codice diventa simile a questo:

```
[SerializeField]
GameObject cube;

private NetworkVariable<float> sliderValue =
    new NetworkVariable<float>(0.5f);

private float oldValue = 0.0f;

//callback dello slider
public void OnSliderUpdated(SliderEventData eventData)
{
    ChangeColorServerRpc(eventData.NewValue);
}
[ServerRpc(RequireOwnership = false)]
void ChangeColorServerRpc(float newValue)
{
    sliderValue.Value = newValue;
}
```

```
    private void Update()
    {
        if (IsClient && oldValue != sliderValue.Value)
        {
            oldValue = sliderValue.Value;
            cube.GetComponent<Renderer>().material.color =
                Color.HSVToRGB(oldValue, 1.0f, 1.0f);
        }
    }
```

Partendo dall'alto, `[SerializeField]` è utile per rendere un campo della classe disponibile nell'Inspector di Unity, in questo modo possiamo assegnare il GameObject del cubo a questo script. `oldValue` serve a mantenere "localmente" l'ultimo valore registrato, è utile per aggiornare il colore del cubo *solo* se differisce dal precedente. `OnSliderUpdated` è la funzione assegnata come callback dell'evento di aggiornamento dello slider. Per il momento, ignoriamo `SliderEventData`, una classe di MRTK. All'aggiornamento del valore dello slider, viene chiamata la funzione `ChangeColorServerRpc` che aggiorna la NetworkVariable internamente al server. Infine, dentro `Update`, funzione chiamata da Unity ad ogni frame, viene controllato prima di tutto se **ci troviamo dentro un'istanza di Unity come client** e se il valore della NetworkVariable è stato aggiornato. Se sì, `oldValue` viene aggiornato e con `cube.GetComponent<Renderer>().material.color = Color.HSVToRGB(oldValue, 1.0f, 1.0f);` sfruttiamo delle componenti di Unity per cambiare la *hue* (tinta) del colore del cubo usando il valore dello slider.

E' importante notare che questa non è l'unica implementazione possibile. Un'altra implementazione consiste, ad esempio, nel sostituire la funzione `Update` con una funzione annotata con `[ClientRpc]`, eseguita perciò su tutti i client, chiamata dal server quando un client aggiorna il valore dello slider, cioè dentro `ChangeColorServerRpc`. A scopo dimostrativo è stata scelta quest'altra via, invece, perché ritenuta più esemplificativa. In casi reali, andrebbe, per esempio, verificato **quanto spesso avviene l'aggiornamento dello slider**, per capire quanto spesso questo evento genererebbe RPC (nel caso dell'implementazione con `[ClientRpc]`), e valutare se il controllo ogni frame della NetworkVariable `sliderValue` appesantisce invece di più la comunicazione.

4.2 Photon

Photon Engine è un backend as a service (SaaS) pensato per applicazioni e videogiochi multiplayer sincroni o asincroni multi-piattaforma. Si basa su un'infrastruttura cloud che gestisce tanti prodotti diversi per offrire diverse

soluzioni. Queste soluzioni sono usate in molti videogiochi al giorno d'oggi e sfruttano diverse delle architetture già viste in precedenza.

4.2.1 PUN

PUN (Photon Unity Networking) è la soluzione offerta da Photon per lo sviluppo di videogiochi su Unity specificatamente. Microsoft stessa consiglia l'utilizzo di Photon PUN per le esperienze multi-utente in realtà mista (come visto nella documentazione di Microsoft [7]), per questo motivo verrà usata in questo progetto. Collegandosi ad un servizio di relay esterno, PUN si appoggia al servizio offerto da Photon Cloud, che è comunque gratuito ma pone alcune limitazioni sul traffico di rete (comunque irrilevanti nel caso di queste applicazioni di piccole dimensioni).

Photon PUN re-implementa e potenzia certe features del networking già presenti in Unity. Inoltre, essendo l'API di PUN costruita simile a quella di Unity, è fatto in modo da facilitare lo spostamento degli sviluppatori su questa API. Esiste anche un convertitore automatico che aiuta il porting di progetti multiplayer già esistenti.

4.2.2 Altre soluzioni

Photon offre tanti altri prodotti, tra cui **Fusion**, evoluzione di PUN e **Bolt**, un altro loro prodotto. Nonostante sia mostrato da Photon come un prodotto migliore dei precedenti sotto tutti i punti di vista, è stato scartato perché ancora troppo recente e soggetto a possibili aggiornamenti. Il citato Bolt è un altro prodotto simile a PUN, ma orientato al client-hosted, architettura per cui è già stato visto Netcode. Per il lato deterministico offrono un prodotto apposito, **Quantum**, ma come già detto questa architettura non verrà testata.

4.2.3 Introduzione a PUN

PUN si base su "stanze" chiamate "Photon Rooms". Sui servizi cloud di Photon, infatti, è possibile creare delle app scegliendo il tipo (in questo caso PUN) e tutte le applicazioni su Unity che faranno riferimento a quell'app su cloud potranno accedere alle stesse stanze. E' possibile anche creare il proprio **Photon Server** a cui gli utenti si connetteranno invece di usare la versione in cloud.

Che cos'è una stanza? In giochi multiplayer questo termine appare spesso, di norma ci si riferisce ad una **sessione di rete** a cui partecipa un numero relativamente basso di giocatori (per esempio 16). E' il caso anche della nostra applicazione, per questo motivo è comodo "ragionare in stanze".

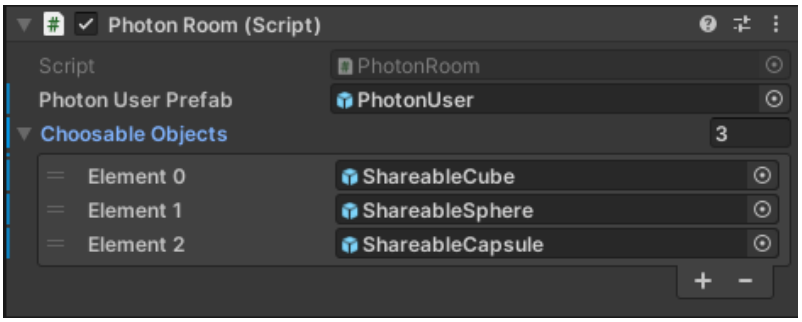


Figura 4.3: Il componente PhotonRoom nell'Inspector

Esiste anche un altro concetto, quello di **Lobby**: la lobby è una zona in cui i giocatori possono scegliere in quale stanza entrare. Nel nostro caso non ne faremo uso perché lavoreremo sempre con una sola stanza, facendoci entrare automaticamente l'utente.

4.2.4 PhotonRoom

Le stanze di Photon possono essere già esistenti o possono essere create. Un client che entra in una stanza viene chiamato **attore**. Nel nostro caso è sufficiente chiamare `PhotonNetwork.JoinRandomRoom()`, una funzione che permette al nostro client di entrare in una stanza casuale: notare che la stanza è casuale perché in questo progetto non si prevede di usare multiple stanze. `PhotonNetwork` è una classe statica che permette di usare il *plugin* di Photon, accedendo alla maggior parte delle sue funzioni. E' importante notare una cosa: l'utente in questo caso non deve fare nulla, solo aprire l'applicazione per entrare automaticamente nella stanza, questo perchè l'applicazione **è già collegata al relay server** tramite il collegamento con il cloud di Photon (autenticato tramite una chiave inserita dallo sviluppatore).

PhotonNetwork

`PhotonNetwork` permette tante altre azioni. Mentre in Netcode esiste il metodo `Spawn` per generare oggetti in rete **dopo averli già istanziati localmente**, su Photon questa classe si occupa di tutto. In Unity normalmente si utilizza `Instantiate` della classe `Object` per creare oggetti nella scena, analogamente su Photon basta chiamare `PhotonNetwork.Instantiate()`

Su Netcode esistono i **Network Prefabs**, ovvero gli unici oggetti che possono essere condivisi in rete. Anche Photon ha un comportamento simile: dentro `PhotonNetwork` è presente la **Prefab Pool**, cioè una *pool* di oggetti che possono

essere riusati e rigenerati. Possiamo quindi aggiungerci oggetti all'aprirsi della stanza di Photon per determinare quali prefab potremo istanziare.

Lo script di `PhotonRoom` diventa così:

```
public class PhotonRoom : MonoBehaviourPunCallbacks, IInRoomCallbacks
{
    public static PhotonRoom Room;

    [SerializeField] private GameObject photonUserPrefab = default;
    [SerializeField] private GameObject[] choosableObjects = default;
    private Player[] photonPlayers;
    private int playersInRoom;
    private int myNumberInRoom;

    ...

    private void Start()
    {
        if (photonUserPrefab != null)
            pool.ResourceCache.Add(photonUserPrefab.name,
            photonUserPrefab);

        if (ChoosableObjects != null)
        {
            foreach(GameObject gm in ChoosableObjects)
            {
                pool.ResourceCache.Add(gm.name, gm);
            }
        }
    }

    public override void OnJoinedRoom()
    {
        base.OnJoinedRoom();

        photonPlayers = PhotonNetwork.PlayerList;
        playersInRoom = photonPlayers.Length;
        myNumberInRoom = playersInRoom;
        PhotonNetwork.NickName = myNumberInRoom.ToString();

        StartGame();
    }
}
```

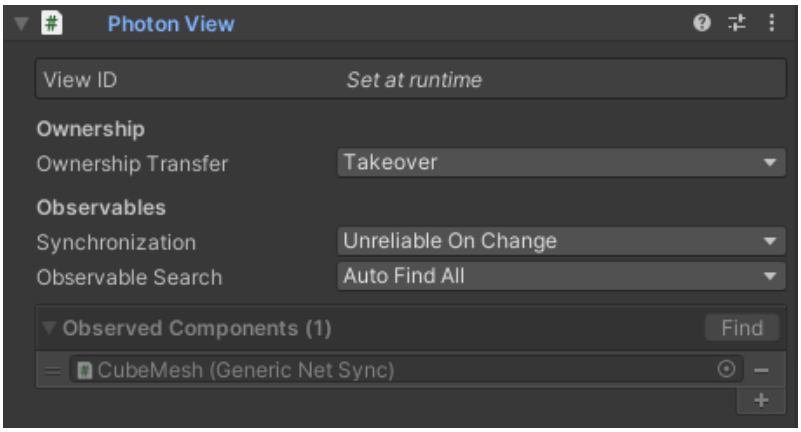


Figura 4.4: Componente PhotonView nell'Inspector

Sono state omesse alcune parti perché ritenute poco importanti ai fini della spiegazione. Per semplificare, il metodo `StartGame` non fa altro che creare l'oggetto "giocatore", cioè `PhotonUserPrefab`, mentre il resto del codice è costituito da metodi callback di alcune funzioni di Photon secondarie (provenienti dalla superclasse `MonoBehaviourPunCallbacks` e interfaccia `IInRoomCallbacks`) già implementate nel progetto d'esempio fornito da Microsoft.

I primi due campi vengono assegnati nell'Inspector, così che lo sviluppatore possa comodamente riassegnare i prefab del giocatore (l'avatar) e degli ologrammi. Il resto dei campi rappresenta dati della stanza, come il numero di giocatori, l'indice assegnato al client che sta eseguendo quel codice e un array di giocatori dentro la stanza.

Dentro il metodo di Unity `Start`, viene gestita la pool di oggetti e inseriti i prefab assegnati nell'Inspector all'interno della `ResourceCache`, un dizionario che mappa il nome del prefab nel prefab effettivo per facilitare e velocizzare l'istanziazione di oggetti successivamente.

Dentro `OnJoinedRoom`, callback per quando l'utente entra nella stanza, viene chiamato prima di tutto il metodo della superclasse `MonoBehaviourPunCallbacks`, classe che contiene una `PhotonView` (spiegato successivamente) e una collezione di callbacks chiamabili da PUN (come appunto questo metodo). Al suo interno vengono semplicemente assegnati i campi dell'istanza **locale** di PhotonRoom e chiamata la funzione `StartGame`.

4.2.5 PhotonView e Photon Transform View

Una **PhotonView** identifica un oggetto nella rete e configura come il client che lo controlla aggiorna le istanze remote degli altri client. Conti-

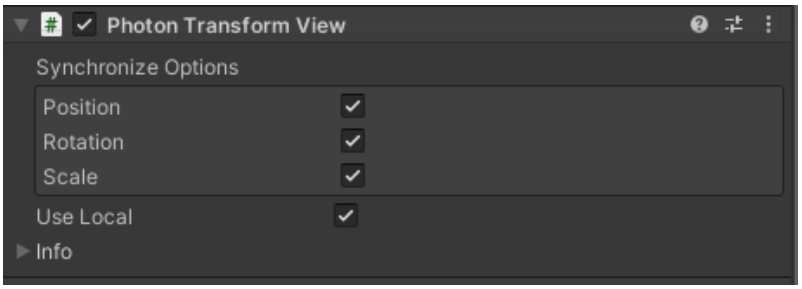


Figura 4.5: Componente PhotonTransformView nell'Inspector

nuando il paragone con Netcode, possiamo vedere PhotonView come l'analogo di NetworkBehaviour o NetworkObject.

Ogni PhotonView ha **un creatore, un proprietario e un "controllore"**. L'attore che chiama `Instantiate` sull'oggetto diventa automaticamente il suo creatore. Ogni PhotonView ha un identificatore chiamato **ViewID**, e questo viene generato usando l'identificatore dell'attore. Il proprietario della PhotonView indica il suo controllore di default. Va notato che gli oggetti possono anche essere **di proprietà della stanza** (networked room objects). Il controllore è l'attore che controlla la PhotonView: eccetto per casi particolari, coincide con il proprietario.

Dal componente di PhotonView è possibile scegliere altri due aspetti: uno è la **synchronization**, che specifica come i dati degli **Observables** (altri script che permettono di condividere certi aspetti dei GameObject, tramite appunto Observables, proprietà che vengono "osservate" e condivise in seguito a certi eventi) vengono sincronizzati tra tutti i client. In questo caso impostiamo `Unrealiable On Change`, che significa che gli aggiornamenti degli observables vengono inviati solo in caso di cambiamento (l'ultimo dato non viene inviato se è uguale al precedente) e che i dati sono ricevuti in ordine ma alcuni potrebbero venir persi. Il secondo aspetto è **Observable Search**, che specifica come questo componente trova gli observables in questo GameObject e nei oggetti figli. Lo impostiamo su `Auto Find All`, che trova automaticamente tutti gli observables nei GameObject figli di questo (compreso lui stesso).

PhotonTransformView è un componente che serve, come il corrispettivo NetworkTransform di Netcode, per sincronizzare il Transform del GameObject associato. A differenza di Netcode, tuttavia, **l'ownership dell'oggetto viene gestita a livello di PhotonView**.

Ownership

In Photon, avere l'ownership di un oggetto significa essere **il controllore del suo PhotonView**. L'ownership può essere di tre tipi:

- **Fissata**: non può essere cambiata. Il creatore è anche il proprietario
- **Cambiabile tramite richiesta**: qualunque attore può richiedere l'ownership tramite `PhotonView.RequestOwnership()`. Questo attiva una callback nell'owner che può trasferire quindi l'ownership tramite `targetView.TransferOwnership(requestingPlayer)` (simile a Netcode)
- **Takeover**: qualsiasi attore può prendere il controllo della view senza avere il consenso del proprietario.

Nel nostro caso non è molto importante controllare se un utente o no può modificare l'oggetto, quindi verrà impostata su "Takeover".

Come è facile notare, Photon rende molto più facile trasferire il controllo automaticamente, semplicemente perchè la parte di passaggio dell'ownership è già presente ed implementata.

4.2.6 RPC e controllo dell'ologramma

Anche in PUN è facile implementare delle RPC. Usando l'attributo `[PunRPC]` su un metodo è possibile chiamarlo successivamente come RPC usando `photonView.RPC("NomeDelMetodo", RpcTarget.All)`. Come si può notare, non esiste la distinzione presente in Netcode tra Client e Server RPC proprio perché Photon non è Client-Hosted (quindi non esiste l'host). Il secondo parametro, invece, serve a specificare i "bersagli" della RPC, ovvero chi la riceve. Esistono diverse opzioni (others, master client ecc.) ma nel nostro caso dev'essere ricevuta da tutti, quindi la impostiamo sul `All`. Seguendo la stessa logica vista per Netcode, possiamo ora re-implementare lo script per il cambio del colore.

```
public class ColorChange : MonoBehaviour
{
    [SerializeField]
    GameObject cube;

    float sliderValue = 0.0f;
    float oldValue = 0.0f;

   public void OnSliderUpdate(SliderEventData data)
    {
        PhotonView photonView = PhotonView.Get(this);
        sliderValue = data.NewValue;
```

```
        if (oldValue != sliderValue)
        {
            photonView.RPC("SendColor", RpcTarget.All, sliderValue);
        }
    }

    [PunRPC]
    void SendColor(float value)
    {
        oldValue = value;
        cube.GetComponent<Renderer>().material.color =
            Color.HSVToRGB(value, 1.0f, 1.0f);
    }
}
```

La struttura risulta molto simile a quella già vista in Netcode. Non dovendo modificare nessuna variabile condivisa, semplicemente ogni client aggiorna la propria variabile locale modificando quindi il colore del cubo localmente. Come forse si può notare, risulta simile al caso affrontato in Netcode (sezione 4.1.6) come implementazione alternativa.

4.3 Integrazione con MRTK

Abbiamo visto con entrambi gli strumenti, Netcode e Photon, come è possibile condividere stato e Transform degli oggetti condivisi. Ora ci occuperemo meglio della parte Mixed Reality usando l'MRTK, vedendo come integrarlo con il framework utilizzato. Nonostante nel progetto questa parte possa rivelarsi marginale grazie alla facilità d'uso di MRTK e alla ricchezza di funzioni che esso offre, nell'obiettivo iniziale di questa esperienza questo aspetto è fondamentale per **capire che livello di astrazione fornisce la libreria di rete utilizzata rispetto alla Mixed Reality**, ovvero quanto facilmente l'aspetto di rete può apparire separato dall'aspetto della realtà mista, durante lo sviluppo.

4.3.1 Object Manipulator

L'ObjectManipulator è un componente di MRTK molto importante in questo progetto. Assegnandolo a dei GameObject è possibile spostarli, scalarli e ruotarli usando una o due mani. Funziona con le gestures di Hololens 2 come le articolazioni della mano e gli "hand rays". Per far sì che l'ogget-

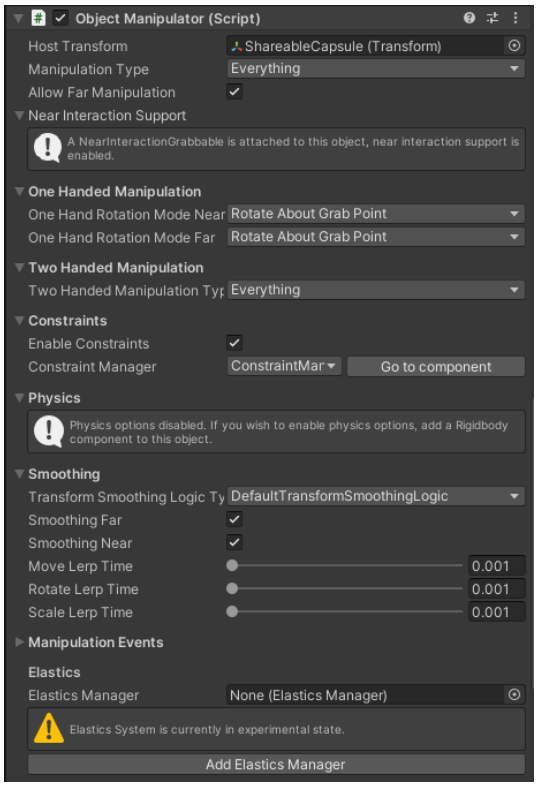


Figura 4.6: Componente Object Manipulator nell'Inspector

to risponda all'input di una mano vicina, è necessario aggiungere anche il `NearInteractionGrabbable`, un altro script di MRTK.

Netcode

Usando semplicemente gli script di MRTK e Netcode assieme in questo caso non funziona: quando si prova usando script come `NetworkTransform`, si nota che **solo l'host è in grado di apportare cambiamenti usando l'ObjectManipulator**. Questo avviene per motivi di ownership: per risolvere questo problema ci torna utile lo script visto alla sezione 4.1.4, per ottenere l'ownership tramite delle RPC. Usando inoltre i "manipulation events" di ObjectManipulator (callbacks assegnabili chiamate in seguito a eventi di manipolazione) è possibile far in modo che questo cambio di ownership avvenga solo **all'inizio della manipolazione**. Nota: è qui che torna utile `ClientNetworkTransform` perché è l'unico che permette cambiamenti di posizione nel client, quindi ottenuta l'ownership anche il client ha accesso a queste modifiche.

Sebbene la risoluzione di questo problema appaia semplice sul lato pratico dopo questa spiegazione, questo ha complicazioni enormi all'interno del confronto portato avanti finora. All'interno dello script di `ObjectManipulator`, infatti, abbiamo dovuto inserire elementi riguardanti il lato networking, **violando l'astrazione del livello MR da quello di rete**.

Photon

In Photon è necessario solo un accorgimento per far funzionare i componenti assieme. PhotonTransformView funziona tramite callbacks di Photon che in seguito a modifiche trasmettono l'informazione sul Transform ai restanti client. Questo può essere fatto, anche in questo caso, solo se lui è **owner** dell'oggetto. Fortunatamente, Photon mette a disposizione l'opzione "Takeover" spiegata nella sezione 4.2.5, permettendo di trasferire automaticamente l'ownership quando avviene la modifica. E' sufficiente quindi impostarlo in questo modo.

A differenza di Netcode, in questo caso è stato sufficiente inserire un nuovo componente per regolare il comportamento di rete dell'ologramma, seguendo la politica "modulare" di Unity. Notiamo che in questo caso MRTK e Photon non sono coscienti della reciproca esistenza, e funzionano mantenendo il livello di astrazione intatto.

4.3.2 Altri componenti

MRTK è molto ricco di componenti utilizzabili per altre forme di input.

Prefab condivisi

I prefab condivisi, ovvero gli ologrammi dell'applicazione, consistono di semplici solidi con uno slider integrato che permette di cambiarne il colore. Alcuni esempi sono quelli visibili nella figura 4.7. In questo semplice progetto gli oggetti condivisi sono questi, ma il vantaggio di questo approccio è che **qualsiasi modello 3D con gli stessi script e registrato tra i "prefab condivisibili" viene sincronizzato in maniera corretta**. Questo aumenta enormemente le potenzialità e lo rende estendibile.

Slider

Lo slider è un componente di MRTK chiamato "PinchSlider", modificabile tramite gestures delle mani.

Come si può vedere in nella figura 4.8a viene assegnata la callback di `OnValueUpdated` allo script del prefab a cui è assegnato questo slider, ovve-

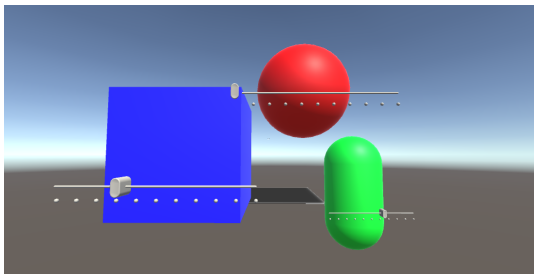
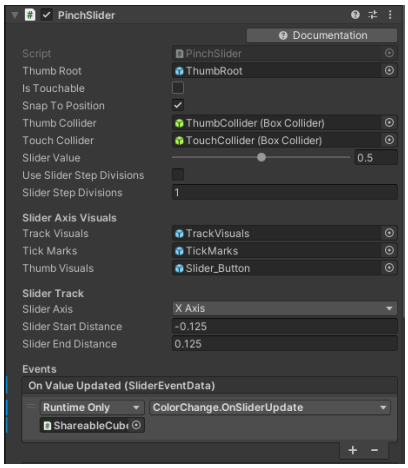
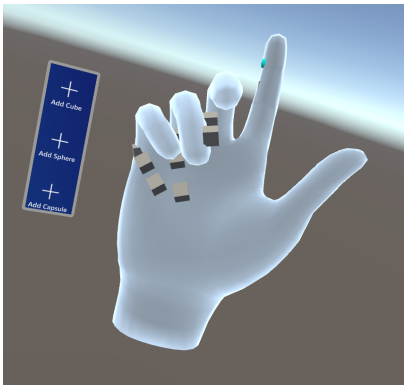


Figura 4.7: Ologrammi generabili nell'applicazione



(a) Componente del Pinch Slider (Inspector)



(b) Componente del Hand Menu

Figura 4.8: Componenti di MRTK utilizzati

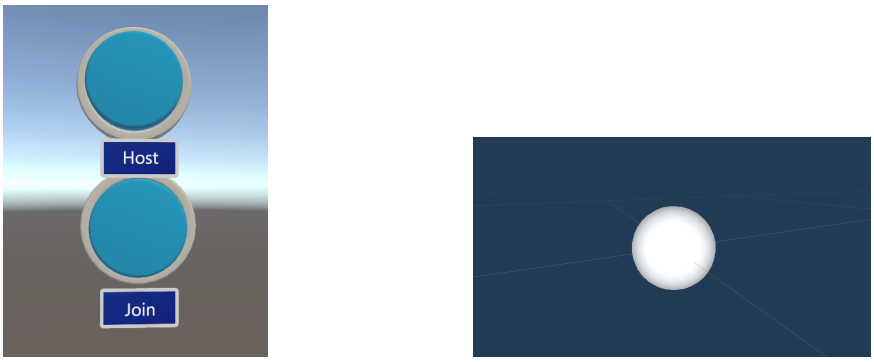
ro `ColorChange`, in modo che attivi la funzione callback già esaminata in 4.1.6 per Netcode e in 4.2.6 per Photon.

Hand Menu

E' stato aggiunto anche un Hand Menu, ovvero un componente di MRTK che permette un semplice e comodo menù che appare sul polso dell'utente. In questo caso è utilizzabile per generare nuovi ologrammi condivisi.

Pulsanti e altri Input

In entrambi i casi analizzati, Netcode e Photon, non abbiamo un sistema di "lobby" che permetta agli utenti di entrare nelle sessioni (stanze) condivise a loro scelta. Nel primo caso, tuttavia, l'accesso alla stanza non è automatico (Netcode non si collega automaticamente ad un servizio esterno), bensì c'è una



(a) Pulsanti presenti nell'applicazione Netcode

(b) Avatar dell'utente

Figura 4.9: Pulsanti e avatar

scelta da compiere, sul dispositivo host e i restanti client. Per questo motivo, solo nel caso di Netcode, sono stati aggiunti due pulsanti per permettere agli utenti di scegliere se diventare l'host oppure entrare in una sessione esistente. Nel secondo caso, è stato aggiunto anche un altro componente di MRTK che richiami la tastiera di Hololens per inserire eventualmente l'IP della sessione a cui unirsi.

Avatar

Per questo progetto di esempio è stata ritenuta di secondaria importanza la realizzazione di un avatar definitivo. Come prova, è stata usata una semplice sfera bianca come rappresentante dell'utente nello spazio virtuale.

4.4 Allineamento fisico dello spazio virtuale

Aspetto fondamentale ancora rimasto da implementare è proprio la "sincronizzazione" dello spazio virtuale con quello fisico. Anche se gli ologrammi sono condivisi ed esiste l'interoperabilità su di essi, non appaiono ancora nella stessa posizione fisica. Se due utenti nella stessa stanza avviano l'applicazione su due visori separati in due posizioni diverse, vedranno tutto il mondo virtuale in posizioni diverse (concetto spiegato nella sezione 3.10). Sono quindi necessari metodi per allinearli, e in questo progetto ne sono stati testati due: uno prevede l'utilizzo di **codici QR** mentre l'altro della piattaforma **Azure Spatial Anchors**.

4.4.1 QR code

Condividere la stessa concezione dello spazio significa condividere un origine e un sistema di coordinate. Un primo metodo intuitivo per fare questo risulta essere il fissaggio di un punto nello spazio fisico, di modo che entrambi i visori, riconoscendo quel punto, si allineino. Il **codice QR** entra in gioco per questo ruolo. Codice a barre bi-dimensionale (matrice) già utilizzato come **marker** in tante applicazioni.

In MRTK è possibile ottenere un progetto d'esempio che utilizza alcune utilities per la lettura di QR code, utilizzabili su visori Windows Mixed Reality (come appunto Hololens). Integrando il pacchetto QR di Mixed Reality, scaricabile usando NuGet (gestore di pacchetti), è possibile integrare questi script direttamente nel progetto.

Il pacchetto si basa su tre script in particolare: **QRCodesManager**, **QRCodesVisualizer** e **QRCode**. Il primo si occupa di integrare un componente del MRTK, il **QRCodeWatcher**, con i restanti script. Quest'ultimo richiede l'accesso alla fotocamera di Hololens, quindi è necessario chiamare in maniera asincrona `await QRCodeWatcher.RequestAccessAsync()` per ottenere il permesso. Dopodiché, all'interno della funzione `Update` di Unity verifichiamo se il permesso è stato accordato con `accessStatus == QRCodeWatcherAccessStatus.Allowed`, chiamando una funzione che serve per fare il setup delle funzioni di callback del QRCodeWatcher. In questo caso gli eventi a cui prepariamo una risposta sono `qrTracker.Added`, `qrTracker.Updated` e `qrTracker.Removed` (`qrTracker` è l'oggetto della classe QRCodeWatcher), ovvero quando un codice QR viene rispettivamente aggiunto (come nuovo), aggiornato e rimosso.

Qui entra il gioco lo script `QRCodesVisualizer`: il suo scopo è quello di gestire le callback impostate nel QRCodesManager per rispondere agli eventi spiegati precedentemente. Questo viene gestito usando una **coda di eventi**, in cui vengono inserite le azioni added, updated e removed. Ad ogni `Update` di Unity viene chiamata la funzione `HandleEvents` che gestisce poi gli eventi presenti nella coda.

```
private void HandleEvents()
{
    // blocca l'accesso ad altri thread sulla coda
    lock (pendingActions)
    {
        while (pendingActions.Count > 0)
        {
            var action = pendingActions.Dequeue();
```

```
            if (action.type == ActionData.Type.Added)
            {
                // Instanzia un nuovo oggetto QR Code
            }
            else if (action.type == ActionData.Type.Updated)
            {
                if (!qrCodesObjectsList.ContainsKey(action.qrCode.Id))
                {
                    // Aggiorna la posizione del QR Code esistente
                }
            }
            else if (action.type == ActionData.Type.Removed)
            {
                if (qrCodesObjectsList.ContainsKey(action.qrCode.Id))
                {
                    // Rimuove l'oggetto QR Code se presente nella
                        lista degli oggetti
                }
            }
        }
    }
}
```

Nel nostro caso non è importante *aggiungere* nuovi QR code, perché quello utilizzato sarà solo uno. Sarà tuttavia possibile spostarlo per aggiornare il sistema di coordinate.

L'oggetto QRCode è un semplice GameObject contenente degli assi cartesiani per indicare l'orientamento del sistema di coordinate. Dentro l'oggetto QRCode esiste l'omonimo script che gestisce il posizionamento e altri dettagli del GameObject associato. Per fare in modo che esso gestisca l'allineamento del sistema di coordinate, è stato aggiunto a questo oggetto anche un riferimento ad un altro GameObject, ovvero `anchor`, cioè l'ancora appunto. Quest'ultimo è riferito ad un GameObject che racchiude **tutti gli ologrammi condivisi** come GameObject figli, di modo che se `anchor` viene traslato o ruotato, tutti gli ologrammi condivisi vengono modificati di conseguenza, **aggiornando il loro transform per restare sullo stesso spazio locale del GameObject padre**. Se l'ancora viene posizionata e ruotata per trovarsi nella stessa posizione del QR code, quindi, automaticamente possiamo dire che **il sistema di coordinate di tutti i visori è allineato**. Dentro lo script QRCode, quindi, possiamo fare in modo che all'aggiornamento della posizione dell'oggetto, "trasciniamo" con esso anche l'ancora.

```
void UpdatePropertiesDisplay()
{
    // Aggiorniamo le proprieta' quando cambiano
    if (qrCode != null && lastTimeStamp !=
        qrCode.SystemRelativeLastDetectedTime.Ticks)
    {
        // Aggiorniamo dati testuali mostrati sull'oggetto QR code in
            scena
        // ...

        // Aggiorniamo il time stamp dell'ultima modifica
        lastTimeStamp = qrCode.SystemRelativeLastDetectedTime.Ticks;

        if (anchor != null)
        {
            // Se l'anchor e' stata assegnata, aggiorniamo la
                posizione dell'ancora
            anchor.transform.position = this.transform.position;
            anchor.transform.rotation = this.transform.rotation;
            anchor.transform.Rotate(new Vector3(90, 0, 0));
        }
    }
}
```

Come si può notare, all'ancora viene impartita una rotazione di 90 gradi lungo uno degli assi. Questo è stato fatto per correggere un bug di posizionamento per il quale l'ancora risultava appoggiata in un diverso orientamento.

La funzione `UpdatePropertiesDisplay()` viene chiamata ogni frame di Unity e permette di aggiornare la posizione dell'ancora ogniqualvolta il QR code subisce un aggiornamento. In questo modo, è stato raggiunto un primo semplice allineamento del sistema di coordinate.

4.4.2 Azure Spatial Anchors

Con la presentazione della realtà mista lato Hololens, Microsoft ha spesso voluto mettere in luce un suo progetto, **Azure Spatial Anchors (ASA)**. E' ormai conosciuta da diverso tempo la piattaforma cloud di Microsoft, Azure, e ASA altro non è che un servizio offerto dalla stessa. Come è stato già detto, le ancore tornano molto utili in realtà mista. Usando quindi ASA è possibile ottenere **lo stesso effetto ottenuto con i marker (QR code), ma in maniera automatica**. Mentre il primo metodo utilizza simboli presenti nello spazio fisico, che vanno quindi inquadrati con una fotocamera per esse-

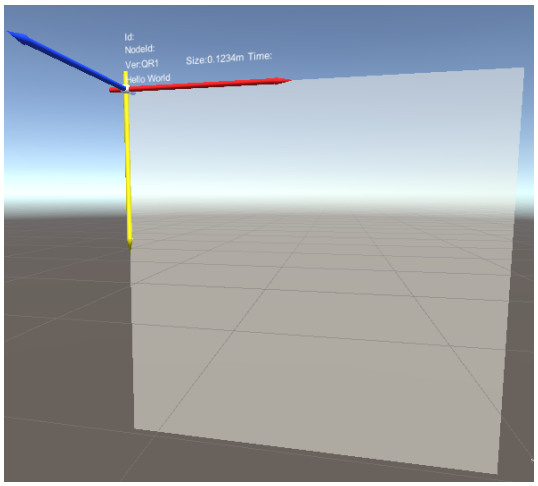


Figura 4.10: GameObject del QR Code istanziato nella posizione rilevata

re scansionati, un servizio come ASA utilizza a pieno le capacità del visore, **scansionando lo spazio e salvando dati relativi a delle ancore, la cui locazione può essere condivisa, tutto in maniera automatica**. Se è possibile condividere un punto dello spazio fisico tra visori, allora è possibile anche condividere un sistema di coordinate. Espressa in questo modo, è ovvio che appaia come metodo migliore rispetto ai marker: anche ASA, tuttavia, non è perfetto. La precisione raggiunta utilizzando questo strumento non è sempre ottimale, di fatto è calcolata tramite delle stime fatte scansionando lo spazio. Questo preclude ad eventuali applicazioni che hanno bisogno di alta precisione il loro utilizzo. Dipende inoltre da un servizio esterno: caricare le ancore aggiornate rompe l'ipotesi già fatta riguardo ad un'esperienza locale. E' importante specificare che entrambe queste argomentazioni hanno possibili risoluzioni: ad esempio aumentare le ancore o scansionare meglio lo spazio per la maggior precisione e implementare sistemi di cache o memoria interna che mantengano l'ultima versione delle ancore salvata in locale per slegarsi dalla connessione esterna.

Per poter utilizzare ASA, è necessario aggiungere al progetto una serie di pacchetti appartenenti a MRTK e accedere tramite il portale cloud di Azure per salvare le ancore sulla stessa piattaforma. Sfortunatamente, l'implementazione dell'allineamento spaziale tramite ASA è risultata incompleta, e quindi non si andrà in dettaglio su di essa. MRTK e Photon, però, mettono a disposizione numerosi script per aiutare nell'utilizzo delle ancore spaziali. Primo fra tutti, `SpatialAnchorManager`, script che incapsula molti dei passaggi necessari all'autenticazione e creazione di ancore (dalla complessità piuttosto elevata).

4.5 Considerazioni finali su Netcode e Photon

In relazione al caso di studio, sia Netcode che Photon risultano essere ottimi framework per lo sviluppo di esperienze condivise in Mixed Reality. Netcode riprende molti aspetti da alcune API già sviluppate sempre da Unity, quindi l'utilizzo appare solido e robusto. Photon cerca di imitare la struttura di queste API, nonostante lavori con un'architettura completamente diversa. Sicuramente la seconda è più **immediata**, in quanto semplifica moltissimi aspetti, come l'**ownership**, che su Netcode invece è necessario toccare più con mano. Anche certi meccanismi come le **RPC** risultano più immediate su Photon, ma resta il fatto che quest'ultimo è un **prodotto fra tanti, pensato in maniera speciale per l'architettura basata su relay avanzato**, mentre Netcode è un prodotto più flessibile ma quindi personalizzabile. Per quest'ultimo aspetto, alcune funzionalità (allineamento spaziale, trasferimento di ownership, interoperabilità tra client) hanno richiesto più tempo per essere implementate su Netcode rispetto a Photon. Così com'è stato implementato il nostro progetto, **Netcode ha raggiunto un livello di astrazione minore sul lato networking**, aspetto che costringe la libreria a dover essere utilizzata "ritoccando" i componenti e, come già visto, intervenire anche **mischiando componenti di MRTK con altri di Netcode**. Questo si rivela piuttosto critico nel nostro confronto perché non stiamo cercando un framework di rete qualsiasi, ma uno che **renda possibile un'integrazione agile con la realtà mista, rendendo però indipendenti le due parti**.

Come già specificato nella sezione 3.4, per questo caso di studio **non è stata trovata un'architettura dominante rispetto alle altre**. Sarà stato notato che in questo capitolo è stato parzialmente tralasciato l'aspetto della **persistenza**, questo perché **in applicazioni "client-based" senza servizi esterni, questa risulta troppo difficile da implementare**, senza ricorrere ad architetture alternative pensate esplicitamente per essa si intende(un esempio è il sistema degli snapshot di Croquet, sezione 2.1.4). Infatti, nel caso di applicazioni che danno molta importanza alla persistenza è necessario optare per un servizio più "server-based", in cui è presente un server, pur leggero che sia, ma che mantenga certe informazioni, oppure appoggiarsi a piattaforme esterne (come Azure Spatial Anchors, che punta molto sulla *persistenza* delle sue ancore). In casi come questo un'architettura client-hosted con server headless (sezione 3.4.1) risulta più realistica, ma non scagiona l'advanced relay, per la sua capacità di middleware di inserirsi in mezzo a più servizi. Per un approccio più "locale" torna in gioco di nuovo client-hosted, ma allo stesso tempo esistono i problemi riguardanti la **barriera di diversità dei vari ruoli, host e client**, come ad esempio l'host migration, barriera che l'advanced relay abbatte. Come si può notare, nessuna soluzione esclude l'altra. E'

necessario, però, affermare che per questa esperienza è stato preferito Photon, semplicemente per la maggiore semplicità. Anche in un caso di studio di bassa complessità come questo, il livello di astrazione fornito da Photon è stato ritenuto **più adatto alla realtà mista rispetto a Netcode**. Tutto questo potrebbe non valere in applicazioni meno prototipali e più ricche e complesse.

4.6 Alternative e sviluppi futuri

Come per le architetture non si può affermare che una risulti sempre migliore dell'altra, lo stesso si può dire degli strumenti utilizzati. Per concludere, quindi, si è ritenuto opportuno citare anche altri strumenti esistenti che possano sostituire i due già citati. Uno è ad esempio **Mirror**, una libreria di alto livello per Unity. Mirror nasce prendendo spunto da molte funzionalità di UNet, versione precedente di Netcode, ed è stato sviluppato originariamente per correggerne alcuni bug e migliorarne alcuni aspetti. E' uno strumento piuttosto utilizzato, sempre d'impianto client-hosted, ma, a differenza dei precedenti, **è completamente open-source** e nasce da una community molto vasta.

Un altro prodotto su cui sarebbe interessante lavorare in sviluppi futuri è **Fusion**, un'altra libreria proposta da Photon. Con Fusion, Photon si propone di evolvere i due principali prodotti da loro offerti (PUN e Bolt) per unire il tutto in un unico strumento potente e flessibile. Essendo ancora di recente uscita, ad oggi rappresenta ancora una soluzione instabile e povera di compatibilità con altri strumenti (aspetto che invece PUN copre ampiamente), ma sarebbe un'ottima fonte di ricerca per progetti futuri.

L'applicazione prodotta in questo esempio si tratta ovviamente di un caso semplice, con livello di complessità sufficiente a dimostrare alcuni aspetti delle librerie utilizzate. In progetti futuri sarebbe interessante integrare quanto appreso in questa esperienza in casi di maggiore complessità e applicazioni finalizzate ad uno scopo più concreto.

Conclusione

Alla fine di questo percorso, sono stati analizzati diversi aspetti di tecnologie prima sconosciute al sottoscritto. Molte di queste sono ancora di recente uscita, questo testimonia quanto giovane e in sviluppo sia la materia trattata in questa tesi. Non solo: personalmente è stato trovato un ambiente di sviluppo già avviato e più solido rispetto alle aspettative. La Mixed Reality rappresenta ancora un'esperienza riservata a pochi, come dimostrano i numeri e i casi di test effettuati sui visori da altri sviluppatori. La ricerca di soluzioni, metodi e inventive per risolvere certi problemi in questo campo può dare una notevole spinta d'insieme a tutta la prospettiva di applicazioni Mixed Reality nel vicino futuro. La tecnologia di Hololens, inoltre, pur presentando certi problemi e incertezze, si è dimostrata capace di offrire un'esperienza difficile da descrivere per chi non ha mai provato questo visore. L'immediatezza e l'agilità di certe sue funzionalità hanno dimostrato di sapere reggere la pressione di rappresentare attualmente il probabile leader dei visori Mixed Reality. Anche sull'aspetto multi-utente, l'ambiente di sviluppo esplorato, ovvero Unity con le librerie utilizzate, ha dato prova di essere potente e flessibile, portando lo sviluppo di un aspetto così complicato come il lato networking di applicazioni real-time cooperative in MR ad essere quasi facile ed intuitivo.

Detto questo, rimangono numerosi coni d'ombra. La Mixed Reality rappresenta un futuro probabile e non troppo lontano dalla sua piena realizzazione, tuttavia non mancano diverse incertezze: un utente abituato a lavorare e interagire nel mondo fisico, è pieno responsabile delle sue azioni e delle relative conseguenze. Quando questa interazione si sposta nel mondo virtuale, questo non è sempre vero. Personalmente viene infatti ritenuto pericoloso questo ruolo di "intermediario" assunto dal dispositivo. Proprio perché le interazioni nel mondo 3D risultano più naturali alle persone, se queste dovessero avere anche piccoli malfunzionamenti quando si opera in realtà mista il sistema apparirebbe molto più fragile e scomodo all'utente, creando disagio sullo stesso. Per apparire solidi, questi sistemi hanno bisogno di ottimizzazione e rifinitura non indifferenti per poter "uscire dal loro guscio".

Dal lato più tecnico, invece, gli strumenti utilizzati sono apparsi ancora scarsamente specializzati. Lo sviluppatore è costretto a mettere insieme tante

librerie, strumenti e software di provenienza diversa per ottenere un risultato presentabile. Lo stesso Unity esiste principalmente per lo sviluppo di videogiochi, e non per applicazioni in realtà mista. Questo si riflette in alcune incertezze e inadeguatezze durante l'utilizzo. In un certo senso, lo sviluppo per MR è stato percepito ancora come privo di una sfera propria. Allo stesso tempo, però, questo aspetto è frutto della ricerca di compatibilità e portabilità in questi sistemi. Spostando la creazione su ambienti certificati e molto utilizzati come Unity, ha permesso anche l'avvicinamento di sviluppatori provenienti da altri ambiti (e di conseguenza l'espansione del settore) e l'allargamento della panoramica di strumenti utilizzabili. Sarebbe interessante vedere la nascita in futuro di un motore grafico dotato di librerie per il lato networking pensato unicamente per la realtà mista, portabile su visori e dispositivi differenti.

Se il mondo della realtà mista saprà affrontare questi problemi, la sua crescita potrà sicuramente non far altro che aumentare. L'utilizzo di librerie come quelle viste in questa tesi, testimoniano il possibile avvicinamento della realtà mista a mondi che già conosciamo, come per esempio quello dell'industria video ludica. L'aumento esponenziale che questo potrebbe comportare unito ad una ricerca e sperimentazione profondi può sicuramente fare sbocciare questa tecnologia. Entro una decina d'anni potrebbe diventare indispensabile possedere un visore o un dispositivo che permetta a ognuno di noi di accedere alla realtà mista e, quando questo avverrà, la presenza di solide e compatte fondamenta per questa tecnologia ci consentirà un passo sicuro e agile in essa.

Bibliografia

[1] Pierre Wellner, Wendy Mackay, Rich Gold, *"Back to the real world"*, 1993

[2] Paul Milgram e Fumio Kishino, *"A Taxonomy of Mixed reality Visual Display"*, 1994

[3] Julie Carmigniani, *"Augmented reality technologies, systems and applications"*, 2011

[4] Ronald T. Azuma *"Survey of Augmented Reality"*

[5] Ronald T. Azuma, Yohan Baillot, Reinhold Behringer, Steven Feiner, Simon Julier, Blair MacIntyre *"Recent Advances in Augmented Reality"*, 2001

[6] A. Ricci, M. Piunti, L. Tummolini, C. Castelfranchi *"The MirrorWorld: Preparing for Mixed-Reality Living"*, 2015

[7] Documentazione di Microsoft, `https://learn.microsoft.com/it-it/windows/mixed-reality/discover/mixed-reality` *"Che cos'è la realtà mista?"*

[8] Unity official site `https://unity.com/`

[9] Unreal official site `https://www.unrealengine.com/en-US`

[10] Croquet documentation `https://croquet.io/docs/croquet/index.html`

[11] "Hololens 2 Tecnical Specifications" `https://www.microsoft.com/en-us/hololens/hardware`

[12] Magic Leap `https://www.magicleap.com/en-us/`

[13] Mark Billinghurst e Hirokazu Kato, *"Collaborative Augmented Reality"*, 2002

[14] Jaybie A. De Buzman, Kanchana Thilakarathna, Aruna Seneviratne, *"Security and Privacy Approaches in Mixed Reality: A Literature Survey"*

[15] OpenXR `https://www.khronos.org/openxr/`

[16] Repository del MRTK `https://github.com/microsoft/MixedRealityToolkit-Unity`

[17] Microsoft Mesh `https://www.microsoft.com/en-us/mesh`

[18] Unity Europe 2017 - Photon vs UNet: multiplayer architecture explained `https://www.youtube.com/watch?v=Y1my5bKhKJY`

[19] Documentazione di Netcode For GameObjects `https://docs-multiplayer.unity3d.com/netcode/current/about`

[20] Utilizzo di MLAPI con HoloLens `https://xr-physics-work-etc.hatenablog.com/entry/2021/07/24/163603`

[21] Documentazione di Photon PUN `https://doc.photonengine.com/en-us/pun/v2/getting-started/pun-intro`

[22] Progetto d'esempio per QR code in MRTK `https://github.com/microsoft/MixedReality-QRCode-Sample`