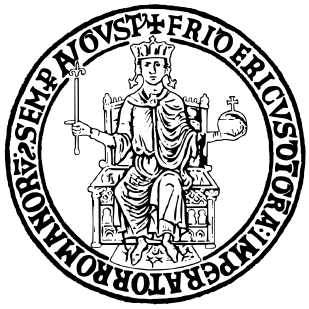
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Triennale in Informatica

Sviluppo di un modulo per la gestione astratta di input e interazioni tramite Hand Tracking, con applicazione ad un sistema di Naked Hand Recognition

Relatore
Prof. Porfirio Tramontana

Co-relatore
Dott. Maurizio De Nino

Candidato
Fabio Fascia
N86003288

Anno Accademico 2022–2023

Sinossi

Le tecnologie di Hand Tracking rappresentano un punto fondamentale nell'evoluzione delle applicazioni in Realtà Virtuale. Le mani come metodo di input hanno un enorme potenziale espressivo e interattivo, in quanto permettono di inserire all'interno di un sistema un grande numero di interazioni differenti, tutte però basate su una rappresentazione comune.

I grandi limiti di queste tecnologie sono stati per molto tempo l'ingombranza dei dispositivi di tracciamento e la loro scomodità per l'utente, problemi tipici degli approcci che prevedono l'utilizzo di guanti di vario tipo (descritti in [3]). Per questo motivo i sistemi di hand tracking basati su *computer vision*, spesso chiamati *sistemi di Naked Hand Recognition*, risultano essere molto promettenti: la visione computazionale sta infatti interessando gran parte della ricerca di settore, con risultati tangibili, e le relative tecnologie stanno diventando sempre più accessibili al grande pubblico. Tecnologie NHR vengono infatti già utilizzate in prodotti attualmente in commercio, ad esempio da Meta, con il suo visore *Oculus Quest 2*.

Con la diffusione di queste tecnologie si rende però necessaria anche un'integrazione efficace delle stesse nei sistemi software preesistenti, oltre che ovviamente in quelli futuri. In particolare è importante fare in modo che i nuovi sistemi di hand tracking possano coesistere in maniera naturale con i sistemi di input più "tradizionali" delle applicazioni XR, come i classici controller XR (*Valve Index*, *Oculus Quest Controller*, ...).

Inoltre queste tecnologie di tracciamento dovranno essere utilizzate come sistemi di interazione uomo-macchina: questo implica l'ulteriore necessità di convertire i dati di tracciamento in interazioni, rappresentate da gesti statici che dovranno poi essere utilizzati all'interno dei prodotti software che vorranno utilizzare la NHR. L'interfaccia per definire e riconoscere le pose statiche dovrà poi essere sufficientemente espressiva, astratta sull'input e semplice da utilizzare per gli sviluppatori.

In questo contesto si inserisce questo progetto di tesi il quale, partendo da un obiettivo specifico (l'integrazione di un sistema NHR all'interno di una applicazione VR) ha allargato gradualmente l'ambito del problema, portando allo sviluppo di un modulo software adatto a qualunque tecnologia di hand tracking.

Indice

Introduzione

Obiettivi della Tesi

L'obiettivo di questo elaborato è illustrare il processo di analisi, progettazione e sviluppo di un modulo software per interfacciarsi in maniera astratta con sistemi di hand tracking per la realtà estesa, fornendo un'interfaccia per i dati i tracciamento e una per il riconoscimento di pose statiche, il tutto nel contesto dello Unity Engine e del suo framework. Il sistema verrà poi applicato ad uno specifico caso d'uso: l'integrazione del servizio di Naked Hand Recognition Ultraleap all'interno di un prodotto software XR, sviluppato appunto in Unity. Il prodotto in questione è un Proof Of Concept (POC) in cui all'utente è richiesto di eseguire diverse operazioni in un ambiente VR, diversificate e di complessità variabile, con l'obiettivo finale di riparare un macchinario virtuale sostituendone un componente danneggiato.

Contesto aziendale

Il lavoro oggetto del presente elaborato è stato sviluppato presso l'agenzia **Digitalcomoedia s.r.l.**, azienda impegnata nel settore delle esperienze digitali, in particolare quelle XR. L'azienda sfrutta queste tecnologie, in combinazione con tecniche derivanti dall'ambito del *game design*, per sviluppare prodotti software che esulano dall'ambito ludico per essere utilizzati in contesti aziendali o pubblici, per fini ad esempio pubblicitari, addestrativi, educativi, culturali.

Con il team del Dott. Maurizio De Nino, che si dedica alla sperimentazione di varie tecnologie XR sperimentali e all'avanguardia all'interno delle applicazioni sopra citate, è quindi nata l'idea di portare avanti un progetto di natura analoga, relativo alla tecnologia di hand tracking a mani nude Ultraleap.

Condizioni e richieste di partenza

Il progetto nasce dall'idea del Dott. De Nino di integrare nei prodotti del suo team la tecnologia Ultraleap. Tuttavia nessun membro aveva esperienza con il sistema in questione, ed era necessario valutare le potenzialità della tecnologia e quantificare l'effettiva possibilità di utilizzo.

È stato quindi richiesto inizialmente di approcciare la tecnologia e le possibilità di integrarla in una applicazione Unity, dato che quest'ultimo è il motore real-time principale utilizzato dal team.

Successivamente a questa fase di studio, per una serie di motivi che verranno approfonditi nell'elaborato (vedi la sezione 1.3.3), è nata l'idea di definire un'interfaccia tra Ultraleap e lo Unity Engine, che è poi maturata nello sviluppo di un'interfaccia per generici sistemi di tracciamento delle mani. Il resto dell'attività di lavoro è stata quindi dedicata alla progettazione e allo sviluppo di suddetta interfaccia.

Attività svolte

La prima fase del lavoro è stata dedicata all'analisi del problema oggetto della tesi: sono state affrontate le problematiche legate all'acquisizione, la codifica, la normalizzazione e l'astrazione di dati di tracciamento in tempo reale (restrizioni dei software in tempo reale, semplicità ed espressività della codifica, estensibilità del sistema...), e quelle associate alla definizione e al riconoscimento real time di pose statiche (ulteriori restrizioni temporali, interfaccia semplificata per la definizione di nuove pose).

Una volta definiti i requisiti (funzionali e non) del sistema, si è passati allo studio delle tecnologie da utilizzare. Alcune di queste erano già note (Varjo XR-3, Ultraleap, Unity Engine), mentre per altre è stata necessaria una lunga fase di ricerca e confronto tra le varie tecnologie. Ciò è stato necessario in particolare per quanto riguarda la scelta dell'interfaccia per i dispositivi di input, che è ricaduta infine come vedremo sull'Input System di Unity. Dopo aver definito l'insieme delle tecnologie da utilizzare, si è passati a studiarne le API e approfondirne il funzionamento.

Fatto ciò si è proseguito con la progettazione del sistema: sulla base dei requisiti di cui sopra, e studiando vari sistemi totalmente o parzialmente affini a quello richiesto, si è scelto di suddividere il sistema in due moduli, rispettivamente per l'input e le interazioni.

Ultima attività svolta è stata l'implementazione del sistema software progettato, il cui funzionamento è stato validato sul campo lungo tutto il corso dello sviluppo. Una volta ultimato il sistema, questo è stato poi inserito come da obiettivo nell'applicazione VR descritta precedentemente, anche in questo caso svolgendo test sul campo.

Contenuti della tesi

Il presente elaborato di testi è strutturato come segue:

- Nel Capitolo 1 viene analizzato il problema in questione, dal cui studio verranno raccolti e analizzati i requisiti richiesti al sistema oggetto della

tesi.

- Il Capitolo 2 fornisce una panoramica sullo stato dell'arte dei sistemi XR/MR e delle tecnologie di hand tracking, con un approfondimento sulla tecnologia Ultraleap, oggetto della tesi.
- Nel Capitolo 3 viene descritto il processo decisionale che ha portato alla scelta dell'API da utilizzare per gestire i dati di input, per poi andare a descrivere il framework di input su cui si è deciso di costruire il sistema, lo Unity Input System
- Il Capitolo 4 illustra la fase di progettazione del sistema dal punto di vista dell'architettura software: verranno mostrati i due moduli da sviluppare e il modo in cui questi si integrano nell'architettura del prodotto finale.
- Nel Capitolo 5 vengono invece approfondite le funzionalità dei singoli moduli, e le soluzioni architetturali e algoritmiche messe in atto per soddisfare i loro requisiti.
- Infine, il Capitolo 6 tratta l'implementazione del sistema software, l'ambiente di sviluppo utilizzato, e l'integrazione del sistema in un progetto dato, al fine di verificarlo e validarlo.

Ai capitoli sopra elencati seguono Conclusioni, Appendice, Bibliografia e Ringraziamenti.

Capitolo 1

Analisi del Problema

1.1 Lo Unity Engine

Unity è un motore per applicazioni real-time 3D interattive, comunemente anche detto **game engine**, sviluppato da Unity Technologies. Un game engine è un ambiente che semplifica lo sviluppo e la distribuzione di vari prodotti software interattivi, quali videogiochi, modelli architettonici, animazioni 3D in tempo reale e altri.

La principale forza di Unity rispetto ai suoi competitor è rappresentata dalla sua architettura *pluggable* il nucleo di Unity è rappresentato solo dalle funzionalità essenziali (motore grafico, motore fisico, motore di illuminazione, game loop, oggetti di scena) mentre tutte le altre funzionalità sono gestite tramite pacchetti software detti **plugin**, installabili direttamente dallo Unity Editor. Questo permette in primis di ridurre lo spazio occupato in memoria dai progetti, evitando di utilizzare plugin non necessari, ma in secundis rende anche semplice il lavoro di estensione del sistema da parte della community. Sono moltissimi infatti i plugin di terze parti (gratuiti e non) che gli utenti dell'engine mettono a disposizione in rete, e la loro integrazione tramite plugin è semplicissima.

Unity è molto diffuso nello sviluppo di applicazioni XR, popolarità dovuta ad una serie di plugin, messi a disposizione gratuitamente dagli sviluppatori di Unity Technologies, dedicati proprio allo sviluppo di applicazioni VR/AR e alla gestione della comunicazione tra Unity e i suddetti sistemi di realtà estesa. I suddetti plugin semplificano enormemente lo sviluppo di applicazioni di questo tipo, sia per i neofiti che per le aziende del settore.

1.2 Varjo XR-3

Varjo è un'azienda operante nel mercato delle tecnologie XR, con un focus specifico sulla ricerca e lo sviluppo di visori per la realtà mista di ultima generazione, aventi come target principale le industrie e i team di ricerca.

Nel corso di questo progetto di tesi è stato utilizzato il **Varjo XR-3**, ultimo modello della loro serie di visori dedicati alla realtà mista. Il visore è stato messo a disposizione dall'azienda ospitante, che lo utilizza attivamente per testare e validare i propri prodotti.

Figura 1.1: Il visore Varjo XR-3

L'utilizzo di questo visore era richiesto come parte dei requisiti del progetto, cosa naturale dal momento che è attualmente uno dei migliori visori MR sul mercato, per varie ragioni:

- Con un Video Pass-Through[1] da 12Mpx, permette di creare ambienti MR ad alta risoluzione con latenze molto basse
- Monta un LiDAR[2] integrato che permette una depth estimation efficiente in tempo reale
- Ha anche un Leap Motion Controller integrato, e fornisce quindi un supporto nativo per la tecnologia Ultraleap, fulcro del progetto di tesi (approfondita nella prossima sezione di questo capitolo)
- Il tutto è gestibile tramite un client per desktop, *Varjo Base*, che semplifica di molto la configurazione del visore e la sua integrazione nel motore di gioco

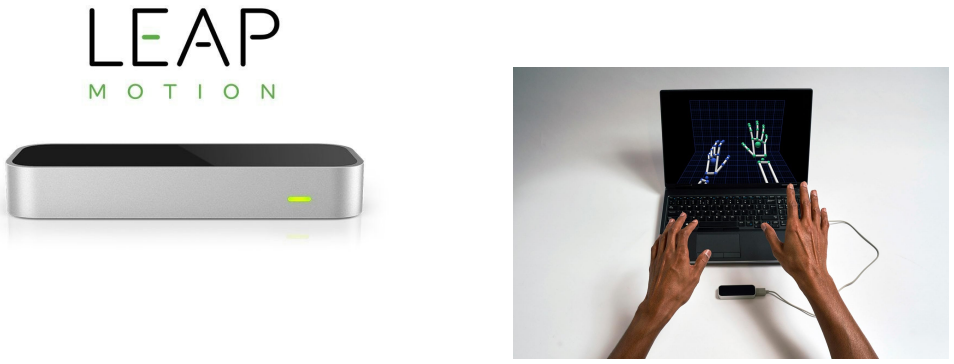
Il Varjo XR-3 offre quindi una serie di ottime funzionalità, ma a proposito va evidenziato il suo target commerciale: il visore ha un costo decisamente elevato, è molto ingombrante (richiede un cavo di alimentazione, 2 cavi HDMI per l'input verso la macchina e 2 cavi DisplayPort per l'output verso gli schermi interni), e richiede una grande potenza di calcolo da parte del sistema che lo utilizza, motivi per cui risulta poco adatto ad applicazioni di natura più commerciale.

[1] capacità di trasportare dati video HDMI attraverso il sistema del computer direttamente verso i suoi output video, senza compromettere la qualità dell'immagine

[2] acronimo di Light Detection And Ranging, è uno strumento di telerilevamento che permette di determinare la distanza di un oggetto utilizzando un impulso laser

1.3 Leap Motion e Ultraleap

La Leap Motion è una startup del settore tecnologico, specializzata nello sviluppo di tecnologie di rilevazione del movimento umano, con applicazione all'interazione uomo-macchina. I loro servizi si basano tutti su una periferica USB da loro progettata, chiamata **Leap Motion Controller**: dotata di 2 telecamere e 3 LED infrarossi, e con un raggio d'azione di circa 1 metro in un'area semisferica, è progettata per identificare principalmente avambracci, mani e dita con una precisione di 0,01 mm. Questa tecnologia è molto simile al *Kinect* di Microsoft, con la differenza che il Leap lavora in un'area di funzionamento più piccola e con maggiore precisione.



(a) (b)

Figura 1.2: (a): il Leap Motion Controller; (b): esempio di utilizzo di Ultraleap

Uno dei servizi principali offerti da Leap Motion è **Ultraleap**, software di hand tracking che comunicando con il Leap Motion Controller permette di eseguire un tracciamento delle mani in tempo reale. Gli sviluppatori possono poi usufruire dei dati di hand tracking all'interno delle loro soluzioni software tramite interfacce apposite, integrando quindi l'hand tracking Ultraleap nelle applicazioni più disparate.

1.3.1 Ultraleap Unity Plugin

Gli sviluppatori di Ultraleap mettono a disposizione, gratuitamente sul loro sito web, una serie di pacchetti per integrare immediatamente la loro tecnologia all'interno di vari ambienti di sviluppo general-purpose. Tra questi, forniscono un plugin che permette l'utilizzo di Ultraleap all'interno di applicazioni sviluppate in Unity, l'**Ultraeap Unity Plugin**. Oltre a permettere di utilizzare il sistema di hand tracking come metodo di input nelle applicazioni, il plugin fornisce varie altre funzionalità, come l'animazione procedurale dei modelli delle mani associati al tracciamento, oppure la definizione di oggetti di scena che interagiscano alle mani di Ultraleap in maniera più complessa.

1.3.2 Struttura del Plugin

Il fulcro del plugin è il cosiddetto **Leap Service Provider**: uno script che comunica direttamente con il servizio Ultraleap esterno, e ne mette a disposizione i dati all'interno di Unity. Tramite le sue interfacce, è infatti possibile accedere ai dati di tracciamento del frame corrente, con le informazioni su tutte le mani riconosciute dal sistema.

Gran parte degli altri script del plugin di appoggiano ad un *Leap Service Provider* per svolgere i loro compiti. Esempi di altre funzionalità che il plugin fornisce sono:

- Animazione procedurale di modelli 3D delle mani in base ai dati di tracciamento forniti da Ultraleap
- Interazione degli oggetti di scena con i modelli animati delle mani (afferrare oggetti, spostarli ecc.)
- Definizione custom di interazioni più complesse, sulla base delle API fornite

Tutte queste funzionalità richiedono la presenza di un *Leap Service Provider* in scena, e l'utilizzo di script forniti direttamente dal plugin. Inoltre, vengono messi a disposizione una serie di *prefab* Unity, utili a semplificare il primo setup di un prototipo, oppure per velocizzare l'inserimento di alcuni script utilizzati di frequente.

1.3.3 Problematiche riscontrate

Il ruolo e le modalità di utilizzo del *Leap Service Provider* pongono già una prima limitazione al plugin: per utilizzarlo è necessario avere un oggetto in scena che non ha altra utilità se non permettere l'utilizzo del plugin stesso. La necessità di avere molti oggetti di scena con funzionalità di controllo è di per sé una cattiva idea di design per applicazioni di questo tipo, a maggior ragione se sono associati unicamente a funzionalità non centrali per il sistema, quale è l'utilizzo delle mani nude per l'input.

Tuttavia la problematica fondamentale riscontrata nell'utilizzo del plugin può essere riassunta in due punti: cattiva integrazione in Unity e pervasività all'interno del sistema. In primo luogo, il plugin non si appoggia ad alcuna delle interfacce fornite dall'API di Unity, e allo stesso modo l'immissione dei dati di tracciamento nel sistema avviene non tramite strumenti forniti dall'engine, ma attraverso un modulo software specifico (il *Leap Service Provider*). Questo rende lo sviluppo di applicazioni con questo plugin fortemente legato all'utilizzo di Ultraleap come unico metodo di input: una parte del progetto che utilizzi il plugin per Ultraleap difficilmente potrà essere riutilizzata per funzionare con modalità di input differenti. E da qui si passa al secondo aspetto della problematica: il plugin si inserisce nell'intera architettura software, dall'input passando per la logica di interazione, fino ad arrivare alla logica di gioco. Di conseguenza per poter utilizzare le API fornite dal plugin, *tutto* il progetto dovrà farne uso, rendendolo in definitiva inadatto allo sviluppo di applicazioni di più ampio respiro.

Nel complesso dunque, l'*Ultraleap Unity Plugin* risulta essere valido solo nello sviluppo di progetti Unity relativamente piccoli, poco flessibili e statici nella loro evoluzione, come possono essere dei piccoli prototipi; non si presta invece bene a soluzioni enterprise più complesse, dinamiche e riutilizzabili. Questa criticità costituisce la ragione d'essere di questo elaborato di tesi, o meglio del progetto che questa va a documentare.

1.4 Requisiti del Sistema

A partire dalle problematiche sopracitate, è possibile delineare l'insieme dei requisiti, funzionali e non, del sistema che si è voluto sviluppare per una più valida integrazione di Ultraleap nello Unity Engine:

Requisiti funzionali

F.1 Astrazione dell'Input Il sistema deve astrarre i dispositivi dall'utilizzo che se ne fa nell'applicativo: una stessa applicazione Unity deve poter utilizzare tanto Ultraleap quanto un controller tradizionale

F.2 Estensibilità Deve essere possibile definire nuove sorgenti per i dati di input: questo permetterà di definire un fornitore di dati che faccia da tramite con il servizio Ultraleap, ma anche in futuro di integrare eventuali nuove tecnologie tramite le stesse interfacce

F.3 Interazioni I dati di hand tracking devono essere utilizzati per interagire con l'ambiente virtuale dell'applicativo

F.3.1 Riconoscimento di Pose Statiche Il sistema deve quindi riconoscere determinate pose delle mani, tramite le quali l'utente agirà sul sistema

F.3.2 Definizione di Pose Statiche Le pose da riconoscere dovranno essere definite dall'utilizzatore del sistema, così da adattare le interazioni possibili al contesto specifico dell'applicazione

Requisiti non funzionali

NF.1 Unity Engine I moduli del sistema devono essere integrabili nell'ambiente dello Unity Engine

NF.2 Supporto Ultraleap Il servizio di hand tracking Ultraleap deve essere compatibile con il sistema

Capitolo 2

Panoramica sulle tecnologie utilizzate

Prima di entrare nel merito del sistema sviluppato, è opportuno approfondire il dominio in cui il progetto si muove: quello delle applicazioni XR è un ambiente molto ampio, con una storia già relativamente lunga, un grande interesse culturale e antropologico oltre che scientifico, e che comprende sistemi anche molto diversi tra loro. Nel seguente capitolo verrà quindi fornita una panoramica sulla storia e le principali problematiche legate al mondo della Realtà Estesa, approfondendo i sistemi di Realtà Mista e le tecnologie a loro più utili. Verrà inoltre affrontato il problema dell'hand tracking, fornendo per quest'ultimo vari tipi di approcci descritti dalla letteratura di settore.

2.1 Sistemi XR

Con la terminologia **Realtà Estesa**, in inglese **Extended Reality (XR)** si fa riferimento a tutti gli ambienti software che combinano elementi reali e virtuali, e alle tipologie di interazioni uomo-macchina generate dalla tecnologia informatica e dai dispositivi indossabili, in inglese *Wearable*. L'acronimo XR è anche un gioco di parole, in cui la X può essere vista come una variabile che può rappresentare una qualunque tecnologia di calcolo spaziale. Nel complesso quindi, XR è un termine canovaccio che comprende tutto ciò che può essere ottenuto applicando le tecnologie informatiche all'amplificazione, estensione o modifica dell'esperienza sensoriale dell'utente.

È da sottolineare come le applicazioni XR rappresentino un sottoinsieme delle applicazioni in tempo reale estremamente critico dal punto di vista dell'ottimizzazione: se infatti in molte applicazioni real-time un leggero rallentamento del gioco può tuttalpiù risultare in un calo qualitativo dell'esperienza utente (si pensi al caso dei videogiochi), invece nelle applicazioni XR in cui la componente virtuale è preponderante, come nella realtà virtuale, un calo di prestazioni dell'applicazione può avere ripercussioni ben peggiori sull'utente finale, che vanno

dal leggero fastidio ottico alla **motion sickness**. Le applicazioni XR sono quindi da considerare strettamente time-critical, con una componente di ottimizzazione che impatta sulla qualità del prodotto finale molto più che in altre applicazioni real-time.

I prossimi paragrafi saranno dedicati ad una breve panoramica sulle principali famiglie di prodotti XR, per poi entrare nel dettaglio delle tecnologie di questo tipo interessate dal progetto.

2.1.1 Realtà Virtuale

Le applicazioni che seguono il paradigma della **Realtà Virtuale (Virtual Reality, VR)** simulano virtualmente ambienti realistici, e ne permettono l'esplorazione mediante l'ausilio di interfacce apposite, principalmente un **visore**, ovvero un casco o strumento analogo dotato di schermi posti vicino agli occhi dell'utilizzatore, che annullano il mondo reale dalla visuale dell'utente sostituendolo all'ambiente virtuale. In generale quindi, si parla di realtà virtuale ogni qual volta il la percezione che l'utente ha del mondo reale viene sostituita con l'esperienza di un ambiente virtuale.

Le origini delle tecnologie VR possono essere fatte risalire agli anni '60 del XX secolo, quando il regista Morton Heilig, che già negli anni precedenti aveva teorizzato lo sviluppo di quello che chiamava "cinema esperienza" ("Experience Theater"), capace di coinvolgere tutti i sensi dell'utilizzatore, sviluppò e presentò un prototipo della sua idea: nel 1962, Heilig presentò il *Sensorama*, un apparecchio che permetteva di assistere a 5 differenti film, ciascuno dei quali forniva stimoli multisensoriali allo spettatore. In momenti specifici dei film, l'utente veniva stimolato in vari modi: vento artificiale prodotto da un sistema di ventole, tremori dati da una sedia vibrante, suoni binaurali generati da speaker stereo, perfino odori provenienti da appositi condotti.

Figura 2.1: Il Sensorama di Heilig

Tuttavia per il primo progetto che si possa propriamente definire di realtà virtuale bisogna andare avanti al 1977, con il software *Aspen Movie Map* sviluppato

dal MIT. Il simulatore permetteva gli utenti di visitare la cittadina americana di Aspen, in Colorado, camminando per le sue strade con 3 modalità visuali differenti: estate, inverno e poligonale. Le prime due consistevano nella replica di filmati delle strade della città, mentre l'ultimo la riproduceva virtualmente con una poligonazione tridimensionale (con ovvi limiti grafici dovuti alle tecnologie dell'epoca).

Figura 2.2: Immagini dell'Aspen Movie Map

Nei decenni successivi queste tecnologie hanno mantenuto uno sviluppo più o meno costante, anche grazie all'interesse per i loro ambiti di applicazione, ma la loro grande diffusione risale solo all'ultimo decennio (2010-2020) grazie alla diffusione in massa dei primi visori commerciali, come l'*HTC Vive*, l'*Oculus Rift* e il *PlayStation VR*, e quindi l'approdo di queste tecnologie al grande pubblico.

2.1.2 Realtà Aumentata

Il paradigma chiamato **Realtà Aumentata (Augmented Reality, AR)** corrisponde all'arricchimento della percezione sensoriale dell'utente mediante informazioni virtuali; ne è un esempio la visualizzazione di elementi virtuali nell'ambiente inquadrato dalla videocamera di uno smartphone. Nelle applicazioni AR, gli elementi virtuali, in genere chiamati anche *ologrammi*, vengono sovrapposti all'ambiente circostante, aggiungendo al mondo reale informazioni e/o interazioni altrimenti non presenti. Gli ologrammi AR possono essere oggetti realistici, testi, effetti sonori, ma anche pannelli e menu interattivi.

Il primo sistema considerabile AR è datato 1968, quando il ricercatore informatico Ivan Sutherland, in collaborazione con il suo studente Bob Sproull, sviluppò un dispositivo costituito da due tubi catodici, che proiettando immagini su un visore le sovrapponevano al mondo reale. Il limite principale di questo dispositivo, considerabile il primo visore propriamente detto mai sviluppato, era costituito dal

suo peso: tale era la mole del dispositivo, che era necessario sostenerlo per mezzo di un braccio meccanico agganciato al soffitto. Fu proprio questa peculiarità a valergli il nome in codice di *Spada di Damocle*.

Figura 2.3: A sinistra: Sutherland con indosso il suo visore; a destra: la *Spada di Damocle* con il suo supporto

Al giorno d'oggi i visori AR sono ben più maneggevoli ed hanno un mercato in crescita: i principali competitor sul mercato al momento in cui si scrive sono il visore *Hololens* di Google e le lenti AR *Meta 2*, dell'omonima compagnia.

2.1.3 Il Reality-Virtuality Continuum

All'inizio degli anni '90 del XX secolo, la conoscenza della comunità scientifica riguardo VR e AR era già abbastanza sviluppata: lo stesso termine *Realtà Virtuale* venne coniato nel 1989 da Jaron Lanier, pioniere dell'ambito XR e fondatore del team di ricerca *Virtual Programming Languages Research*. Inoltre nel corso degli anni '80 l'idea di *cyberspazio* iniziò ad entrare a gamba tesa nella cultura popolare, grazie ad opere come i romanzi dello scrittore statunitense William Gibson, film come *Tron* e opere di altro tipo, ad esempio il gioco di ruolo *Cyberpunk 2022*. Questo interesse della società per il soggetto senza dubbio aiutò nel produrre un rinnovato interesse nell'ambito, che infatti vide grandi sviluppi in quei decenni. Tuttavia, a valle di quel grande sviluppo delle tecnologie, si iniziò a sentire la necessità di fare ordine riguardo la natura di questi progetti, che venivano associati tutti al termine VR nonostante avessero nature molto diverse tra loro.

Fu quindi nel 1994 che, nel contesto di due ricerche separate [**Mil+94**], l'americano Paul Milgram e il giapponese Fumio Kishino definirono in contemporanea il **Reality-Virtuality Continuum**: quest'ultimo è un diagramma monodimensionale che rappresenta su un asse continuo il passaggio da un ambiente totalmente reale ad uno totalmente virtuale; i valori intermedi dell'asse indicano quindi tutte le possibili combinazioni di realtà e virtualità, secondo un grado che indica quanto l'ambiente virtuale si sovrappone a quello virtuale.

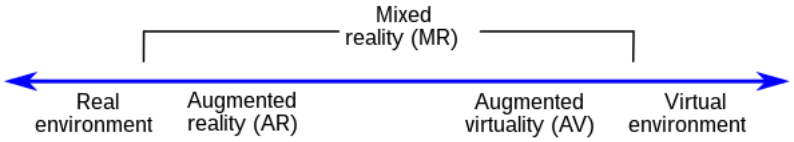


Figura 2.4: Diagramma del Reality-Virtuality Continuum

Seguendo quindi questa interpretazione, l'estremo negativo dell'asse rappresenta l'assenza di virtualità, mentre l'estremo positivo è un ambiente in cui i dati virtuali sostituiscono in toto l'esperienza sensoriale dell'utente.

Il Mediality-Virtuality Continuum

Nel 2002 il ricercatore canadese Steve Mann propose un'evoluzione di questa idea: introdusse un'asse dimensionale aggiuntivo al grafico, corrispondente al grado con cui la percezione reale dell'utente viene alterata dalla tecnologia. Questo asse viene chiamato di *medialità*, e trasforma il Reality-Virtuality Continuum in un diagramma bidimensionale, chiamato appunto **Mediality-Virtuality Continuum**.

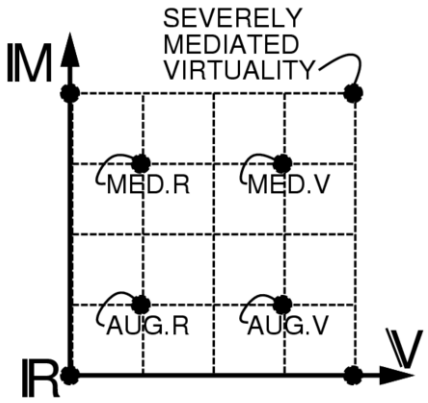


Figura 2.5: Diagramma del Mediality-Virtuality Continuum

Un elemento che emerge da questo nuovo diagramma è quello che Mann definì **Realtà Mediata (Mediated Reality)**, posizionato in alto a sinistra nella figura 2.5, con cui si indicano tutti gli ambienti in cui la percezione umana viene alterata dalla tecnologia, che sia estendendola, modulandola oppure riducendola. Secondo questa estensione della categorizzazione proposta a suo tempo da Milgram e Kishino, il mondo XR (rappresentato dall'asse delle ascisse del diagramma M-V, figura 2.5) sarebbe quindi solo una parte delle esperienze multimediali possibili, con tante altre soluzioni ancora parzialmente o totalmente inesplorate.

2.1.4 Realtà Mista

L'approccio XR che più interessa l'argomento della tesi è senza dubbio la **Realtà Mista**, o **Mixed Reality (MR)**. Con questa nomenclatura si indicano i sistemi che prevedono una mescolanza di ambienti reali con ambienti digitali, per produrre nuovi ambienti ibridi in cui oggetti fisici e virtuali interagiscono tra loro in tempo reale. Nei sistemi MR, l'utente ha la possibilità di interagire contemporaneamente sia con l'ambiente reale circostante, che con oggetti virtuali posti realisticamente all'interno di quest'ultimo, e in alcuni casi può sperimentare anche un'interazione tra un oggetto reale ed uno virtuale. Per come è definita, la Realtà Mista può essere posizionata in un punto intermedio non meglio specificato del diagramma R-V di Milgram e Kishino, punto che varierà da sistema a sistema in base al grado di virtualità integrata nello stesso.

Un esempio di sistema MR potrebbe essere uno in cui all'utente è richiesto di utilizzare attrezzature reali (come un cacciavite) per riparare un macchinario virtuale, visualizzato tramite un visore o strumentazioni di altro tipo.

Allo stesso modo, è un sistema MR uno in cui l'utente interagisce con l'ambiente virtuale utilizzando le proprie mani nude, senza fare ricorso a periferiche di input. È quindi evidente l'importanza che ricoprono le tecnologie di riconoscimento e tracciamento delle mani nude all'interno dello spettro delle soluzioni MR.

2.1.5 Applicazioni delle tecnologie XR

La grande famiglia delle soluzioni XR ha una altrettanto estesa gamma di applicazioni differenti, alcune delle quali già ampiamente industrializzate e commercializzate:

- **Gaming**: già da parecchi anni le soluzioni AR vengono utilizzate in ambito videoludico (basti pensare al recente fenomeno mobile *Pokémon GO*), mentre il mercato dei videogiochi VR è in continua crescita grazie anche a tecnologie dedicate (PlayStation VR/VR2 e altri)
- **Musei**: applicazioni XR possono estendere l'esperienza dei visitatori, arricchendo le opere di informazioni mostrate a video oppure descrizioni mirate a stimolare l'attenzione dell'utente ad alcuni dettagli dell'opera
- **Siti storici**: la realtà virtuale permette di effettuare riproduzioni fedeli di siti di partimonio storico e culturale, aggirando problemi di scarsa manutenzione dei siti storici oppure rendendoli disponibili anche durante chiusure temporanee
- **Simulazione realistica**: il mondo XR è stato fin dagli albori utilizzato per lo sviluppo di simulatori quanto più realistici possibile, in particolare simulatori di volo per addestrare nuovi piloti, e simulatori automobilistici per l'industria e per lo sviluppo di vetture competitive

- **Istruzione**: tramite sistemi XR, vari dati multimediali possono essere sovrapposti all'ambiente reale di uno studente. Ad esempio un libro di testo potrebbe incorporare dei *marker*, che quando scansionati da un dispositivo AR possono fornire informazioni supplementari all'argomento trattato

2.2 Tecnologie MR

Nel mondo XR vi sono una serie di tecnologie che ricoprono un'importanza fondamentale nel sottoinsieme dei sistemi di realtà mista. È importante quindi riconoscerle e comprenderne le criticità, per poter valutare correttamente la bontà di una soluzione MR.

2.2.1 Video See-Through

Il *Video See-Through*, o VST, è forse la tecnologia che più di tutte rappresenta il mondo delle applicazioni MR, in quanto consiste proprio nell'insieme delle tecniche algoritmiche finalizzate a combinare, in fase di rendering, i dati provenienti dalle stereocamere del visore con quelli relativi all'ambiente virtuale, in modo da fornire all'utente l'impressione che gli oggetti virtuali siano posizionati nello spazio fisico che lo circonda.

Questa premessa apparentemente semplice cela però una serie di problematiche più o meno complesse:

- **Prospettiva**: l'ambiente in realtà mista deve essere coerente dal punto di vista prospettico, cosa non semplice in quanto richiede un rendering degli oggetti virtuali basato su una moltitudine di variabili, tra cui la posizione dell'oggetto rispetto all'ambiente reale e all'utente, e l'angolo con cui il visore inquadra l'oggetto
- **Illuminazione**: gli oggetti virtuali devono poter interagire con l'illuminazione dell'ambiente reale, e viceversa l'effetto dell'illuminazione virtuale deve riflettersi sull'ambiente reale. Questa è tendenzialmente la problematica più complessa, dato che implica la necessità di riconoscere le sorgenti di luce reali, convertirle in illuminazione virtuale, e applicare quest'ultima agli oggetti nell'ambiente. Inoltre, altre problematiche associate sono la gestione delle ombre, e delle superfici opache o riflettenti.

Tutto ciò contribuisce ad aumentare enormemente la complessità computazionale del VST, che tuttavia rimane ancora gestibile se considerato nel contesto dell'algoritmo a sé. Non bisogna però dimenticare che questo insieme di algoritmi va tipicamente integrato in applicazioni real-time, il che rende la loro ottimizzazione critica.

Per evitare confusione, è necessario sottolineare la differenza tra *Video See-Through* e *Video Pass-Through*: il secondo corrisponde alla possibilità di trasportare il segnale video (in genere HDMI) in entrata del calcolatore direttamen-

te verso l'esterno, senza elaborazione intermedia, evitando quindi un possibile calo di qualità dell'immagine. Per creare un buon VST si fa tipicamente uso di questa funzionalità, che quindi è utile all'implementazione del VST ma non ne è assolutamente un sinonimo.

2.2.2 Depth Estimation

Con il termine **Depth Estimation** ("stima della profondità") si intende il problema di associare a ciascun pixel di un'immagine la sua distanza dalla videocamera (o videocamere) che l'ha generata. Tale problema ha un'applicazione fondamentale nelle applicazioni MR: in un ambiente misto, come già detto in precedenza la prospettiva deve essere unificata tra ambiente reale e virtuale; ma ciò implica che un oggetto reale posto tra l'osservatore e un oggetto virtuale dovrà coprire la visuale di quest'ultimo, e viceversa. La distanza di un oggetto virtuale è relativamente semplice da valutare, dato che basta considerare la sua distanza dal visore nel sistema di riferimento dell'ambiente virtuale. A risultare complessa è invece la valutazione della distanza di un oggetto reale: è questo il problema principale indirizzato dalle soluzioni di Depth Estimation.

Esistono vari approcci possibili alla Depth Estimation, suddivisibili generalmente in due famiglie:

- **Computer Vision**: utilizzando unicamente le immagini ottenute dalle videocamere, si applicano algoritmi di Computer Vision per dedurre la distanza di ciascun pixel dalla videocamera. In questa categoria rientrano una moltitudine di approcci differenti, ad esempio basati su stereocamere oppure deep learning
- **Sensor Fusion**: i dati di profondità vengono raccolti tramite strumentazioni apposite (in genere un LiDAR), e in seguito associati opportunamente all'immagine. Questo approccio risulta molto più efficiente a livello computazionale, ma richiede strumentazione aggiuntiva, con tutte le problematiche che ne derivano (costo, ingombranza ecc.).

2.3 Tecnologie di Hand Tracking

Sono detti sistemi di **hand tracking** tutte quelle tecnologie atte a digitalizzare i dati posizionali di mani e/o braccia umane in un ambiente, rappresentandoli in un data model predefinito. Queste soluzioni hanno un ampio spettro di applicazioni, ma tutte sono riconducibili in sostanza al loro utilizzo nell'ambito dell'interazione uomo-macchina, nello specifico in relazione al riconoscimento di gesti umani come forma di comunicazione e di interazione con l'ambiente.

Nel corso degli anni l'evoluzione di questi sistemi ha portato alla loro suddivisione in categorie più o meno numerose: di seguito viene quindi fornita una breve descrizione delle principali categorie di sistemi di hand tracking identificabili nella letteratura di settore [4].

2.3.1 Approcci basati su guanti strumentali

I sistemi per il tracciamento delle mani nascono con l'invenzione dei cosiddetti **guanti strumentali**, meglio noti in letteratura come **data glove sensor**, ovvero dei "guanti-sensori" parafrasando dall'inglese. Questi sono dei dispositivi indossabili dall'utente sotto forma di guanti, forniti di una serie di sensori di varia natura la fusione dei cui dati permette di identificare posizione e posa della mano.

L'invenzione del primo data glove viene in genere attribuita ai ricercatori Daniel J. Sandi e Thomas DeFanti [**DS77**], membri e fondatori dell'Electronic Visualization Laboratory della University of Illinois Chicago, che basandosi su un'idea del ricercatore Richard Sayre svilupparono nel 1976 il **Sayre Glove** (qui in figura 2.6).

Figura 2.6: Il Sayre Glove di Sandi e DeFanti

Questi guanti erano basati su dei meccanismi elettromagnetici: su ciascun dito era legato un tubo flessibile con una sorgente di luce ad un'estremità e un fotorivelatore[1] all'altra. Flettendo il dito, l'intensità della radiazione rilevata diminuiva, fornendo una misura dell'abduzione delle dita. Questo sistema forniva quindi solo dati relativi alla flessione delle dita (e non ad esempio alla posizione della mano nell'ambiente), ma fu comunque sufficiente a stimolare la ricerca nella direzione di queste tecnologie.

In questi sistemi il riconoscimento delle pose avviene necessariamente tramite analisi sui dati di tracciamento: le informazioni ricevute dai sensori vengono utilizzate per ricavare un modello della mano, il quale viene analizzato per decidere a livello software quale gesto sta venendo eseguito dall'utente. Questo approccio è utilizzabile (e in effetti utilizzato) anche in sistemi privi di sensori e basati sulla visione artificiale (vedi sezione 2.3.2), ma in quel caso vi sono molte altre soluzioni possibili, non disponibili nel caso dei guanti strumentali.

Ancora oggi l'approccio basato su guanti strumentali viene utilizzato attivamente, questo per una serie di vantaggi incontrovertibili:

[1] dispositivo in grado di rilevare la radiazione elettromagnetica, fornendo in uscita un segnale avente un'intensità di corrente o una differenza di potenziale proporzionale all'intensità della radiazione rilevata

- **Precisione**: un sistema basato su sensori può facilmente fornire le coordinate rototraslazionali esatte del palmo e delle dita della mano in maniera consistente
- **Sensori aggiuntivi**: un guanto strumentale può essere integrato con sensori aggiuntivi per ottenere dati di varia natura, mirati ad aggiungere nuove funzionalità oppure relativi al dominio applicativo interessato
- **Feedback aptico**: l'evoluzione recente dei sistemi di hand tracking indossabili si sta concentrando sul fornire dei feedback sensoriali virtuali all'utente, in particolare feedback tattile (o *aptico*, dall'inglese *haptic*) di varia natura, come piccole vibrazioni o scariche elettriche sulle punte delle dita, oppure cavi che quando tesi imprimono resistenza al moto di abduzione delle dita

Tuttavia questi sistemi portano con sé anche una serie di problematiche non indifferenti, principalmente legate alla loro usabilità:

- **Ingombranza**: i dispositivi indossabili (anche quelli di ultima generazione) tendono ad essere ingombranti e pesanti, il che li rende non solo scomodi ma anche poco adatti ad applicazioni dedicate ad utenti con difficoltà motorie (e.g. applicazioni di supporto alla riabilitazione di pazienti del tipo sopracitato)
- **Comunicazione**: i guanti hanno la necessità di comunicare con la macchina tramite un canale proprio, che necessiterà poi di un'elaborazione dedicata. Nelle prime soluzioni sperimentali questa comunicazione era necessariamente cablata, il che creava non pochi problemi alla mobilità dell'utente. In seguito con l'evoluzione dei protocolli di comunicazione wireless (Wi-Fi, Bluetooth, ...) i problemi di usabilità sono stati in parte risolti, ma le limitazioni legate al supportare questa comunicazione restano (ad esempio il dispositivo non può allontanarsi troppo dal calcolatore/ricevitore del segnale).
- **Autonomia**: l'idea di utilizzare guanti strumentali con alimentazione cablata è stata fin da subito vista come problematica per la mobilità, per cui allo stato dell'arte odierno queste tecnologie montano un'alimentazione interna; questo implica però una loro subordinazione ai tempi di autonomia

Con il tempo è nata quindi la necessità di sviluppare sistemi di hand tracking che fornissero maggiori libertà di movimento e autonomia. In questo senso non si sono fatte attendere le soluzioni orientate più al software che all'hardware.

2.3.2 Approcci basati su Computer Vision

Con **visione artificiale**, o **computer vision**, si intende l'insieme dei processi mirati a ricavare un modello approssimato del mondo reale a partire da immagini bidimensionali. In particolare rientrano nella categoria della computer vision quei sistemi che puntano a identificare e modellare componenti del mondo reale

a partire da un'immagine. È quindi evidente che queste soluzioni siano perfette per implementare sistemi di hand tracking, in particolare quando affiancati da visori XR, le cui immagini possono essere usate come input per gli algoritmi di riconoscimento.

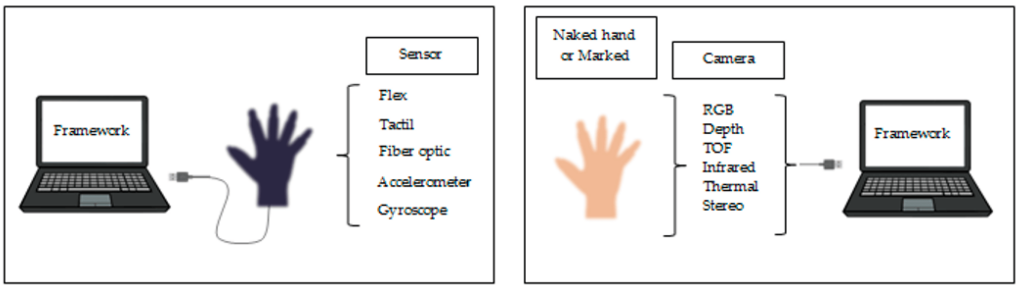


Figura 2.7: Confronto tra i sistemi di hand tracking basati su guanti e quelli basati su computer vision

I sistemi di hand tracking basati su computer vision hanno una serie di vantaggi intrinsechi rispetto all'utilizzo di guanti strumentali:

- **Indipendenza dall'hardware** : gli algoritmi di riconoscimento possono essere utilizzati su immagini provenienti da vari tipi di videocamere, ad esempio monoculari, grandangolo, a tempo di volo (TOF)[2], infrarosse (IR) e altre. Ciò permette di effettuare hand tracking anche su dati forniti da sorgenti molto diverse tra loro, senza incrementare ulteriormente la complessità del sistema
- **Libertà di movimento**: non essendo legato ad un hardware specifico, ma solo alle immagini dell'ambiente, il tracciamento può avvenire senza che l'utente utilizzi altri dispositivi oltre a quello necessario a ricavare le immagini stesse
- **Economicità**: una riduzione delle tecnologie dedicate implica una riduzione dei costi di produzione del sistema, che si spostano in gran parte sullo sviluppo software

L'utilizzo di algoritmi di computer vision apre inoltre la strada ad un'ampia gamma di strategie differenti per il riconoscimento di gesti:

- **Basate sul colore (Color-Based Recognition)**: le varie parti della mano vengono tracciate sulla base del loro colore, sia esso quello della pelle oppure di un guanto di tela colorato indossato appositamente (detto *Glove Marker*)
- **Basate sull'aspetto (Appearance-Based Recognition)**: il riconoscimento della mano avviene sfruttando le caratteristiche dell'immagine

[2]permette di stimare la distanza di un oggetto dalla telecamera, calcolando il tempo impiegato da un impulso luminoso a percorrere il tragitto camera-oggetto-camera (il "tempo di volo" appunto)

in input (e.g. la variazione di valore tra pixel adiacenti), che vengono confrontate con un modello della mano basato su caratteristiche analoghe

- **Basate sul movimento (Motion-Based Recognition)**: in questi sistemi il modello della mano viene estratto dal confronto tra più frame consecutivi, riconoscendo e analizzando il suo movimento; ideale per modellare gesti dinamici
- **Basate sullo scheletro (Skeleton-Based Recognition)**: la mano viene riconosciuta sulla base di vincoli geometrici imposti sui dati in input, concentrando quindi il modello finale sulle proprietà statistiche della mano umana e sulla sua geometria, piuttosto che sul reale aspetto dell'elemento tracciato
- **Basate sulla profondità (Depth-Based Recognition)** [8]: questi approcci utilizzano una telecamera di profondità per ottenere immediatamente informazioni sulla geometria tridimensionale dell'oggetto, e su di essa applicare gli algoritmi di riconoscimento
- **Basate su deep-learning**: come il nome suggerisce, sono quelle soluzioni che sfruttano le reti neurali e le loro capacità di riconoscere pattern per eseguire il tracciamento della mano

Ciascuna di queste strategie di riconoscimento ha dei suoi pro e contro, dei contesti più o meno vantaggiosi per l'utilizzo, dei tipi di immagini in input su cui lavorano meglio o peggio ecc. L'elenco precedente va visto solo come una breve panoramica degli approcci attualmente in uso: si possono trovare in letteratura molte altre soluzioni più o meno variegate, e il problema è attualmente ancora oggetto di studio [7].

L'utilizzo della visione computazionale porta quindi con sé una serie di ovvi vantaggi. Tuttavia vanno fatte presenti anche le difficoltà implementative e i limiti che questo approccio comporta:

- **Imprecisione**: gli algoritmi di computer vision effettuano una *stima* della posa della mano, ed includono quindi necessariamente un certo grado di errore da cui non è possibile prescindere
- **Elementi nascosti**: risulta molto difficile, per gli strumenti della visione artificiale, ricostruire il modello di un oggetto solo parzialmente visibile nell'immagine, con la necessità di fare inferenza sulle parti mancanti; tracciare un oggetto non inquadrato è poi totalmente fuori dalle capacità di questi sistemi
- **Ostacoli al tracciamento**: le condizioni ambientali possono influenzare negativamente le capacità di riconoscimento degli algoritmi. Superfici riflettenti, variazioni di illuminazione, occlusione ambientale possono risultare un grosso ostacolo al riconoscimento corretto della mano dall'immagine

- **Overhead computazionale**: spostare la complessità del tracciamento dall'hardware al software comporta un aumento della complessità di tempo computazionale del tracciamento, che va quindi bilanciata con la risoluzione dell'immagine per rendere il tracciamento valido da utilizzare in applicazioni real-time

Capitolo 3

Framework per Input e Interazioni

Il primo problema che è stato necessario affrontare in fase di progettazione è stata la scelta di un'interfaccia per l'input che fosse supportata da Unity, sulla quale strutturare poi l'intero sistema. Questo capitolo approfondisce prima le problematiche legate a questa scelta, poi le motivazioni dietro la decisione presa. Dopodichè verrà approfondita l'interfaccia scelta, così da fornire nei prossimi capitoli una migliore comprensione della progettazione del sistema.

3.1 Studio iniziale per la scelta del framework di Input

Le primissime fasi della progettazione sono state dedicate come detto alla scelta del framework che avrebbe fornito le fondamenta per inserire. La scelta è ricaduta infine sullo **Unity Input System**, framework messo a disposizione dallo stesso Unity Engine per la gestione dei dispositivi di input. Prima di entrare nel dettaglio di questo framework, verrà però fornita una spiegazione delle motivazioni dietro questa scelta, e del processo decisionale che ha portato a questa scelta progettuale.

3.1.1 Requisiti richiesti al framework

Dall'analisi dei requisiti di sistema si possono evincere varie funzionalità che l'interfaccia per i dispositivi deve necessariamente fornire. In particolare, dovranno essere rispettati i requisiti F.1, F.2 e F.3, dai quali si possono trarre le seguenti conclusioni:

- Per implementare al meglio il requisito F.2, è necessario un framework che gestisca i vari dispositivi collegati in maniera ordinata, seguendo una ontologia riconoscibile: in caso contrario, l'aggiunta di molti dispositivi fini-

rebbe per rendere l'utilizzo del sistema eccessivamente confuso, rallentando così lo sviluppo

- Il requisito F.3 sottintende la necessità di un'interfaccia che faccia da ponte tra i dati grezzi di input e la logica di gioco. Se questa non fosse presente, gli utilizzatori del sistema dovrebbero necessariamente lavorare con i dati di input nella loro forma primitiva, il che non sempre è l'opzione migliore, specialmente quando i dati in input sono soggetti a forti fluttuazioni (come nel caso di dati provenienti da sensori)

Inoltre vanno presi considerazione i requisiti non funzionali già descritti (NF.1, NF.2).

3.1.2 Opzioni considerate

Una delle prime opzioni considerate è stata di appoggiarsi al plugin Unity per **SteamVR**, sistema basato su *OpenVR*, specifica proprietaria di Valve per definire le interfacce di comunicazione con dispositivi XR. La scelta era motivata principalmente da fattori di produzione: nel contesto aziendale in cui si è mosso il progetto, SteamVR era largamente utilizzato, il che avrebbe semplificato l'utilizzo del framework. Questa scelta tuttavia non solo avrebbe comportato l'introduzione di una API proprietaria nel progetto, ma avrebbe reso poco elegante il design finale del software, che si sarebbe basato sull'utilizzo di interfacce in un contesto estraneo a quello per cui erano state pensate.

Successivamente si è quindi valutata la possibilità di utilizzare plugin sviluppati direttamente dal team di Unity, in modo da lavorare con interfacce pensate per un utilizzo più general-purpose. La scelta più ovvia ricadeva quindi sul framework che Unity mette a disposizione proprio per l'input di applicazioni XR: lo **Unity XR Framework**. Questo avrebbe permesso di accedere ad una serie di funzionalità specifiche per il dominio considerato, il che avrebbe semplificato enormemente il lavoro degli utenti dell'Input Adapter.

Purtroppo però l'*XR Framework* è uno strumento relativamente nuovo nell'ecosistema Unity, e in quanto tale non fornisce ancora la possibilità di estenderlo ad alto livello. Il sistema sarebbe quindi dovuto essere definito dal basso livello, implementando le specifiche **OpenXR**, specifica standard analoga ad OpenVR, ma che a differenza di quest'ultima non sono proprietarie ma definite da un consorzio apposito. Questa soluzione avrebbe tuttavia reso molto più complessi sia lo sviluppo che l'utilizzo del sistema finale, motivo per cui il framework è stato scartato.

3.1.3 Motivazioni della scelta dello Unity Input System

La scelta è ricaduta infine sullo **Unity Input System**, framework che gli sviluppatori di Unity mettono a disposizione per la gestione dei dati di input. L'Input System fornisce interfacce per definire vari livelli dell'architettura software, dall'interfaccia con le periferiche fino alla definizione di eventi in-game. La sua

struttura intuitiva, adatta all'estensione e all'utilizzo in contesti anche "singolari" (come l'introduzione di dati non provenienti da un dispositivo hardware), lo rende ideale per l'utilizzo nel caso d'uso interessato. Le sue API sono poi estendibili e implementabili ad alto livello, il che semplifica enormemente la progettazione e lo sviluppo del sistema.

3.2 Unity Input System

Lo Unity Input System è stato introdotto ufficialmente nella versione 2019.4 dell'engine (in forma sperimentale già nella 2018.3) per sostituire la classica API di input di Unity, fornita dal package *UnityEngine.Input*, con un'interfaccia più potente, flessibile e configurabile. Le sue API permettono di gestire l'acquisizione dei dati di input dall'interfaccia con l'hardware fino all'utilizzo nella logica dell'applicativo.

3.2.1 Struttura dello Unity Input System

Control, Device e Layout

Nello Unity Input System, l'unità elementare che costituisce un dispositivo di input è il **Control**. Un Control definisce un tipo di dato che i dispositivi di input esterni possono mettere a disposizione del sistema. Ad esempio, un gamepad può fornire dati relativi alla pressione dei suoi vari pulsanti (D-Pad, ABXY...) e alla posizione delle sue levette analogiche e trigger analogici. Un Control è identificato da:

- Il tipo di dato che fornisce al sistema (intero, reale, vettore 2D/3D, quaternione...).
- Il modo in cui questo dato è codificato (la pressione di un pulsante è codificata con un intero oppure una bitmap?).
- Altre caratteristiche di varia natura, come ad esempio se è soggetto a oscillazioni aleatorie (*noisy*), oppure il suo valore di default.

I Control possono essere sia **elementari** (pulsanti, levette analogiche, sensori di posizione/rotazione) che **composti** (i dati di tracciamento relativi ad una mano, composti dai dati di tracciamento delle varie ossa). I Control composti sono tipicamente utilizzati nel caso in cui il dato restituito sia un oggetto.

Tramite i Control è possibile definire i dispositivi di input, che Unity chiama per l'appunto **Device**. Un Device è descritto dall'insieme dei Control che mette a disposizione del sistema. Questo insieme di Control è descritto tramite un'interfaccia, detta **Layout**. Ad esempio, un Layout di tipo Gamepad può fornire dei Control relativi alla pressione dei suoi vari pulsanti (D-Pad, ABXY...) e alla posizione delle sue levette analogiche e trigger analogici.

Ciascun layout è poi descritto da varie proprietà, tra cui un nome identificativo,

la classe di dispositivi a cui fa riferimento ed altri identificativi (opzionali) di varia natura, tipicamente legati a interfacce di basso livello.

Control Hierarchy

Seguendo questa definizione di Device e Control, i Layout possono essere organizzati in quelle che Unity chiama **Control Hierarchies**. Queste sono gerarchie ad albero, in cui la radice è rappresentata dal Layout di un Device, e i cui figli possono essere di due tipi:

- Layout che descrivono un Control, che rappresentano i dati di input che quel dispositivo mette a disposizione; un Layout Control può a sua volta avere Layout figli, nel caso in cui il Control in questione sia composto.
- Layout che descrivono altri Device, che rappresentano specializzazioni del Layout padre (ad esempio, il Layout *Gamepad* può essere specializzato nel modello di gamepad specifico).

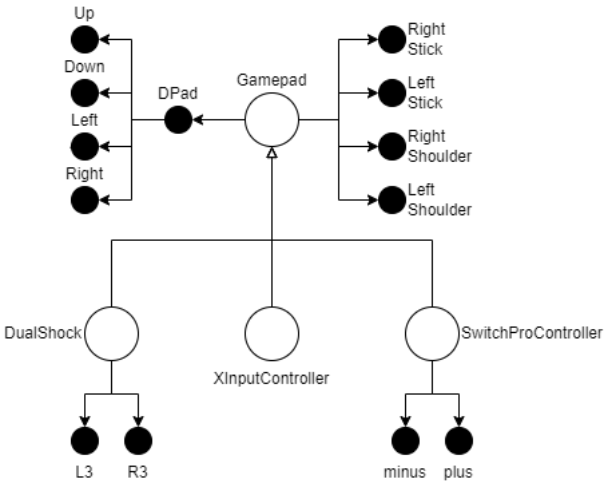


Figura 3.1: Esempio di Control Hierarchy dell'Input System per un Layout di tipo "Gamepad"

Questa classificazione dei dispositivi e dei loro Control permette al sistema di selezionare il Layout più adatto ad ogni dispositivo in base alla similarità tra i due: il sistema cercherà sempre di associare ad un dispositivo il Layout più specifico tra quelli adatti (quello più in profondita nell'albero).

Memory Layout

L'Input System dedica ad ogni Device un blocco di memoria, che viene usato per scambiare i dati di input tra il dispositivo e il sistema. Questo blocco conterrà variabili dedicate per ciascuno dei Control del Device, allocate in maniera contigua, così da rendere sempre a tempo costante la lettura dei dati forniti dai Control. Essendo da allocare contiguamente, tale blocco di memoria deve avere

una struttura statica e immutabile. Questa struttura, chiamata **Memory Layout**, va associata a ciascun Layout: se un Device viene associato ad un certo Layout, il Memory Layout corrispondente verrà utilizzato per il blocco di memoria a lui dedicato.

Più Layout possono essere associati allo stesso Memory Layout, dato che se due dispositivi utilizzano gli stessi Control, non ha senso definire due Memory Layout differenti. Inoltre una stessa variabile del Memory Layout può contenere i dati di più Control. Un semplice esempio di questo caso sono pulsanti la cui pressione è memorizzata tramite una bitmap: più pulsanti sono descritti dalla stessa cella di memoria.

Infine, va fatto presente che anche i Control devono avere un loro Layout. Questo è banale nel caso dei Control elementari, dato che il Memory Layout sarà costituito da tipi di dato forniti da Unity o dal linguaggio (tipi elementari, *Vector2*, *Vector3*, *Quaternion*), ma diventa notevole nel caso di Control composti: riprendendo l'esempio precedente, se vogliamo definire un Control che fornisca i dati di tracciamento della mano, il suo Memory Layout dovrà contenere variabili dedicate ai vari Control specifici di ciascun dito.

Actions e Action Maps

Layout e Control definiscono l'interfaccia tra il sistema e le periferiche, ma non il modo per utilizzare effettivamente l'input all'interno dell'applicazione. A questo scopo l'Input System definisce un livello ancora più alto di astrazione, rappresentato dalle **Action**.

Una Action rappresenta un generico input dell'utente visto dalla prospettiva della game logic. Se i Control costituiscono un wrapper per il significato *fisico* degli input ("pulsante x") le Action incapsulano il significato *logico* degli input (azione "conferma"). Le Action possono essere definite tramite una GUI apposita dello Unity Editor; possono poi essere invocate e referenziate dagli stessi script di Unity, tramite componenti apposite oppure da script C#. Così facendo l'input dell'utente può finalmente arrivare alla logica di gioco, senza però che quest'ultima abbia bisogno di conoscere alcunché sulla provenienza dei dati.

Ogni Action è descritta da:

- Un nome identificativo, in genere rappresentativo del suo significato logico
- Un tipo di dato, che indica il genere di dati di input associati all'Action. Questo può essere tanto un flag booleano, nel caso di un pulsante, quanto un valore reale, un vettore o un dato strutturato

L'associazione tra una Action e un Control è poi reificata tramite gli **Input Bindings**. Un Input Binding contiene informazioni su un Control (specificato mediante la sua posizione nella Control Hierarchy, in forma di URI) e sulla Action a cui il Control fornisce dati. La reificazione dell'associazione permette una associazione molti a molti tra Action e Control: un Control può fornire dati a più Action (lo stesso input può scatenare più eventi), così come una Action può

ricevere dati da più Control (più input possono scatenare lo stesso evento). Agli Input Bindings possono anche essere associati dei filtri, chiamati **Processors**, che si frappongono tra il Control e l'Action elaborando i dati prima che raggiungano la game logic, ad esempio normalizzando i dati vettoriali.

Il sistema della Action fornisce poi ulteriori strumenti per aumentare la flessibilità dell'applicativo: le Action sono raggruppate in Action Maps, permettendo di modificare rapidamente gruppi interi di Action in base alla necessità, ad esempio nel caso in cui il sistema dovesse poter girare su piattaforme diverse (desktop, mobile, console).

3.2.2 Funzionalità dell'Input System

Inserimento di un nuovo Device nel sistema

Quando un nuovo dispositivo viene riconosciuto da Unity in maniera automatica o manuale (tramite codice ad alto livello), le sue caratteristiche verranno confrontate con quelle dei Layout riconosciuti dall'Input System. Dopodichè, se almeno uno dei Layout corrisponde alle caratteristiche del dispositivo, ovvero tutti i suoi attributi corrispondono agli attributi del device, quest'ultimo verrà associato a quel Layout, permettendo di accedere ai suoi dati di input tramite l'interfaccia corrispondente. In caso di corrispondenze multiple, il sistema sceglie il Layout con più attributi (e che quindi descriverà un tipo più specifico di dispositivo).

Scelto il Layout da utilizzare, viene allocato il blocco di memoria dedicato ai dati di input del Device, utilizzando il Memory Layout associato al Layout scelto. Dopodichè il Device potrà utilizzare il proprio blocco di memoria per immettere nell'Input System i propri dati di input.

Input Feeding

Come già detto i dati di input di un Device passano all'Input System tramite il suo Memory Layout. Di default l'input feeding dei Device collegati al sistema avviene ad ogni frame, seguendo la logica del Game Loop: all'inizio del ciclo di game loop, che rappresenta un singolo frame dell'applicazione, i dati che il Device mette a disposizione vengono memorizzati nelle variabili dedicate del suo Memory Layout, secondo la codifica specificata. Il codice che gestisce l'input feeding può essere definito in due modi differenti:

- **A basso livello** tramite gli stessi ID di basso livello che permettono a Unity di riconoscere i dispositivi collegati in maniera automatica, in particolare la *Human Interface Device* (HID). Tale interfaccia permette di gestire configurazione, input e output delle periferiche fisiche standard, e quindi viene usata da Unity per gestire a tutto tondo i Device standard in maniera automatica (più dettagli su HID in [riferimento alla documentazione Windows di HID]).

- **Ad alto livello** tramite le API dell'Input System stesso. Mediante interfacce specifiche (che verranno approfondite nei prossimi capitoli) è possibile accedere ai Control di ciascun Device e aggiornarne le relative variabili del Memory Layout. Spesso i Control dei Device così aggiornati vengono identificati dall'Input System come **sintetici** (tramite l'attributo *synthetic*), dato che tendono a non corrispondere a reali sorgenti di input fisiche.

Al termine del ciclo di game loop, il Memory Layout del dispositivo viene resettato, impostando tutti i Control ai valori di default. Questo comportamento può però essere bloccato per uno specifico Control, associandogli l'attributo *dontReset*. Va però comunque ricordato che il contenuto di una variabile del Memory Layout viene svuotato quando il valore del Control corrispondente viene letto, e tale comportamento non è sopprimibile in alcun modo.

Capitolo 4

Architettura del Sistema

4.1 Architettura a livelli

L'Input Adapter si innesta in un'**architettura a livelli** preesistente, nella quale le componenti del sistema possono essere raggruppate in base al livello di astrazione a cui elaborano i dati di input, oltre che naturalmente secondo le loro responsabilità. In particolare, possiamo identificare, in ordine crescente di astrazione, i seguenti livelli:

1. **Livello Fisico**: il layer con cui l'utente si interfaccia, rappresentato dai dispositivi di I/O
2. **Interfaccia HW/SW**: contiene i driver e i software di gestione specifici degli hardware utilizzati, come *Varjo Base* e il servizio Ultraleap
3. **Unity Entry-Point**: fornisce le interfacce per introdurre i dati esterni nel framework dello Unity Engine; a questo livello si posiziona il sistema software sviluppato
4. **Framework di Input**: livello a cui i dati vengono strutturati in maniera ordinata e intuitiva, per renderli user-friendly; qui ritroviamo l'Input System descritto nel capitolo precedente
5. **Logica dell'applicativo**: livello più alto in cui si colloca la game logic dell'applicativo; degno di nota anche lo *Unity Interaction System*, altro plugin dell'engine dedicato alla definizione delle interazioni tra gli oggetti di scena, perfetto per l'utilizzo in combinazione con l'Input System

La figura 4.1 mostra un diagramma riassuntivo dell'architettura del sistema, evidenziando la posizione e il ruolo del sistema Input Adapter sviluppato.

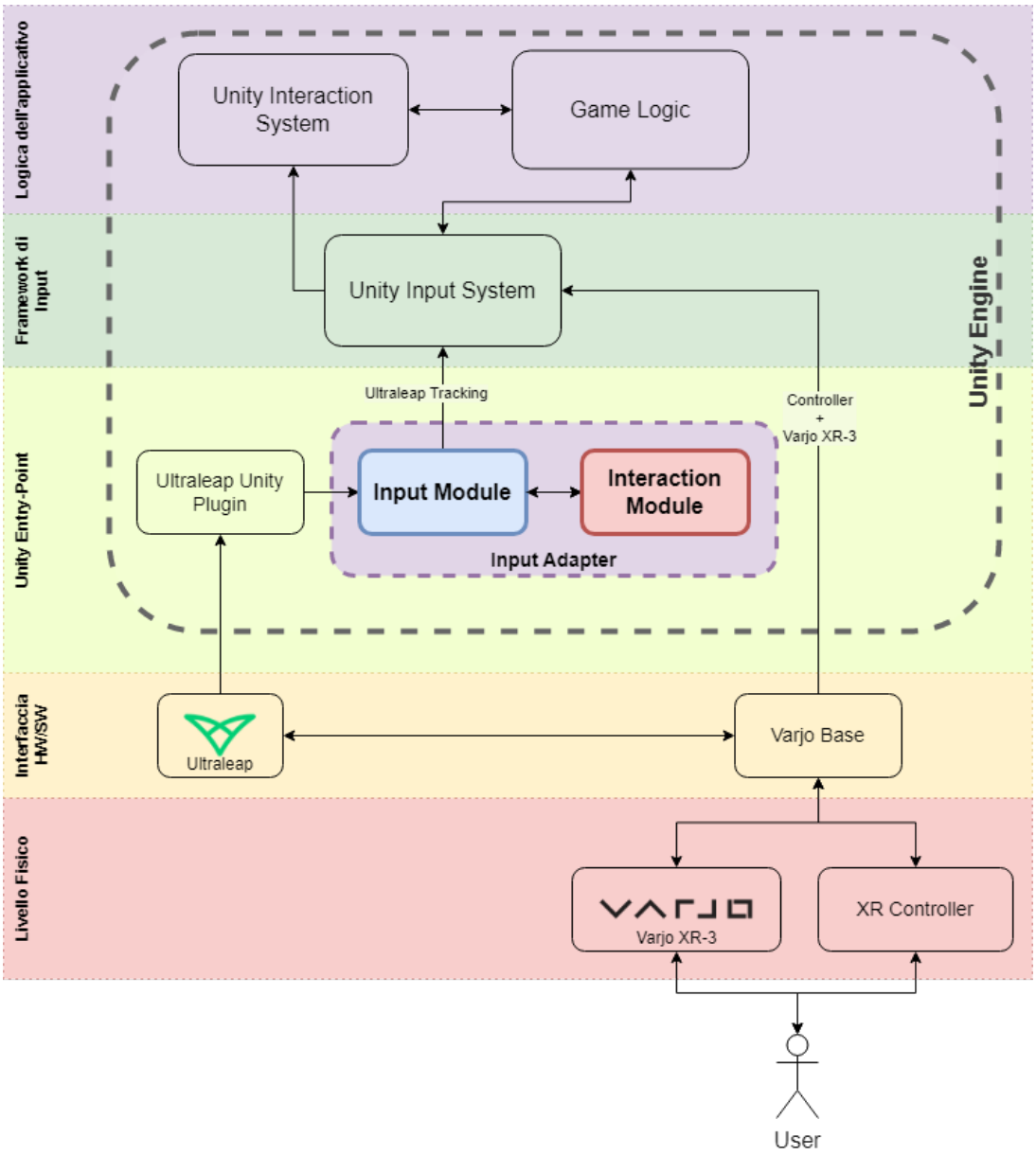


Figura 4.1: Diagramma dell'architettura del sistema

4.2 Moduli dell'Input Adapter

Il sistema sviluppato presenta due moduli principali:

- **Input Module** : fornisce l'interfaccia di comunicazione tra lo Unity Input System e i sistemi di hand tracking esterni, e gestisce la pipeline di acquisizione, conversione e astrazione dei dati di input, mettendo questi ultimi a disposizione di Unity, attraverso appunto le interfacce dell'Input System.
- **Interaction Module** : fornisce le interfacce e gli strumenti per definire delle pose statiche utili ad interagire nell'applicazione, e utilizza i dati

forniti dall'Input Module per effettuare un riconoscimento real-time delle stesse. In tal modo rende possibile utilizzare la gestualità delle mani per interagire nell'applicativo.

4.3 Struttura del Sistema

Il funzionamento complessivo del sistema può essere descritto ad alto livello tramite il percorso che dati di tracciamento seguono per andare dai dati grezzi di stereocamera/Leap Motion, passando per Ultraleap fino ad arrivare alla game logic di alto livello.

- Sorgenti dei dati
    1. I raw data provenienti dal Varjo XR-3 raggiungono il sistema tramite il software di gestione del visore, *Varjo Base*.
    2. Varjo Base gestisce il servizio Ultraleap integrato nel visore, manipolandoli ed analizzandoli tramite computer vision per ottenere i dati di hand tracking.
    3. I dati così ottenuti vengono immessi nello Unity Engine tramite l'Ultraleap Plugin per Unity.
- Input Module
    4. Le interfacce fornite dal Plugin vengono incapsulate in un wrapper, che converte i dati nelle strutture e nel formato adatto a Unity.
    5. I dati incapsulati raggiungono l'Input System di Unity tramite degli Input Device sintetici; i dati dei loro Control/Memory Layout vengono aggiornati ad alto livello con questi stessi dati.
- Interaction Module
    6. I dati di tracciamento vengono utilizzati come input per un algoritmo di riconoscimento di pose statiche. Il suo risultato è una posa in un insieme definito precedentemente dagli utilizzatori del sistema, corrispondente al gesto correntemente eseguito dalla mano tracciata.
    7. Anche il risultato dell'algoritmo di riconoscimento raggiunge l'Input System tramite gli Input Device di cui sopra.
- Game Logic
    8. I dati provenienti da entrambi i moduli dell'Input Adapter potranno essere utilizzati in Unity tramite il sistema di Action-Control dell'Input System.
    9. A questo punto i dati di tracciamento sono utilizzabili integralmente all'interno della logica di gioco dell'applicativo.

Naturalmente un percorso analogo verrebbe seguito nel caso di dati provenienti da sorgenti di natura differente: è semplice constatare che per sistemi di tracciamento differenti l'unica fase che si differenzia è quella precedente all'immissione dei dati nell'Input Module. Il sistema dunque riesce effettivamente ad astrarre l'utilizzo dei dati di tracciamento dalla loro origine.

Capitolo 5

Funzionalità del Sistema

In questo capitolo verranno descritte le funzionalità offerte dal sistema Input Adapter: saranno illustrati nel dettaglio i due moduli del sistema, descrivendo le loro funzionalità, il modo in cui si interfacciano con il resto del sistema, e le soluzioni di design alla base del loro sviluppo.

5.1 Input Module

Il primo modulo del sistema ad essere stato progettato è il modulo di gestione dell'input, denominato **Input Module**. Questo si occupa di tutto ciò che riguarda la conversione e l'astrazione dei dati di tracciamento, acquisendo i dati dalle API esterne e fornendoli all'Input System di Unity.

5.1.1 Modello dei dati utilizzato

Il modello dei dati utilizzato per rappresentare le mani è stato basato sui modelli analoghi forniti sia da Unity che da Ultraleap. Il sistema prevede, per ciascuna mano, un modello osseo con un totale di **21 giunzioni articolari**, una per il polso e quattro per il singolo dito. Queste ultime rappresentano le quattro ossa delle dita umane, ovvero in ordine: Metacarpale, Prossimale, Distale e Intermedio. Sono state ignorate nel modello le punte delle dita, in quanto le loro informazioni rototraslazionali sono derivabili dai dati di cui sopra. Ogni giunzione è rappresentata dalla sua posizione (un vettore 3D) e dalla sua rotazione (un quaternione[1], seguendo lo standard di Unity); collegate tra loro in maniera gerarchica (il movimento di un punto viene applicato anche ai suoi "figli"), più precisamente, posizione e rotazione di una articolazione vengono specificate nel sistema di riferimento solidale con l'articolazione precedente nella gerarchia.

[1] estensione matematica dei numeri complessi, costituiti da quadruple in cui un numero è un valore reale, gli altri tre sono valori immaginari; per alcune loro proprietà, sono utili alla rappresentazione di rotazioni 3D

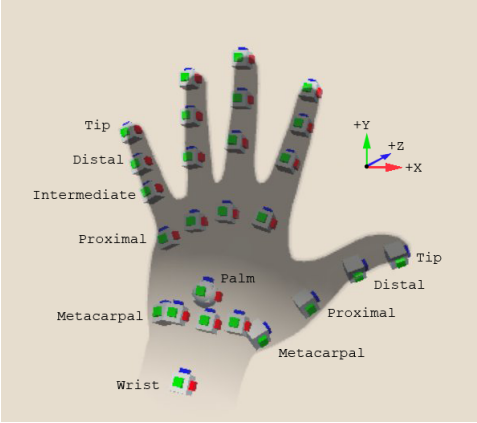


Figura 5.1: Modello di rappresentazione per i dati di tracciamento della mano

5.1.2 Conversione dei dati di Hand Tracking

La prima fondamentale operazione a carico dell'Input Module è l'acquisizione dei dati provenienti dalle API esterne, per poi renderli disponibili al resto del sistema tramite delle interfacce idonee. Il modulo deve però anche tenere conto della possibilità che diversi sistemi di tracciamento utilizzino diverse rappresentazioni dei dati, con la conseguente necessità di uniformarli ad un modello comune.

Acquisizione dei dati

I dati di tracciamento che si desidera astrarre vengono forniti all'input module tramite un'interfaccia apposita, che implementa il design pattern **Adapter**, più comunemente conosciuto come **Wrapper**. Quest'ultimo permette di convertire l'interfaccia di una classe in un'altra interfaccia, compatibile con quella attesa dall'utilizzatore della classe. Il pattern fornisce quindi un modo per conciliare interfacce che non potrebbero altrimenti essere utilizzate insieme.

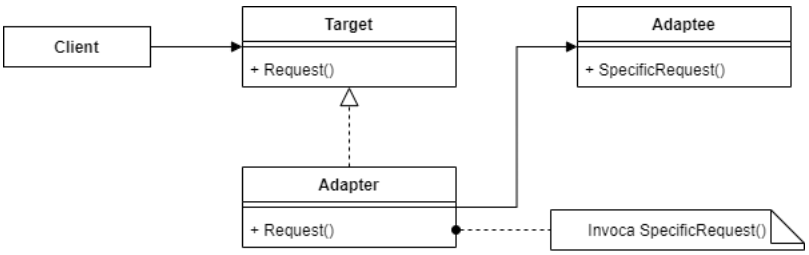


Figura 5.2: Class Diagram del pattern Adapter

Nell'Input Module il pattern Adapter viene implementato da un'interfaccia, denominata **IHandAdapter** (nel rispetto delle convenzioni di nomenclatura di C#). Questa interfaccia definisce tutti i dati che il provider del tracciamento deve fornire al sistema, nel formato sopra descritto, tramite i suoi metodi. Per

utilizzare i dati di un provider all'interno del sistema, è quindi sufficiente implementare tale interfaccia utilizzando le API del provider come componente *adaptee* del pattern: ogni metodo si occuperà quindi di acquisire dei dati specifici dall'API esterna, convertirli nel formato atteso e restituirli. Nel caso dell'Ultraleap Plugin, il ruolo dell'*adaptee* è interpretato dalla struct *Hand* dell'API esterna, contenente i dati di tracciamento relativi ad uno specifico frame.

IHandAdapter fornisce dati tramite strutture di tipo **Finger**, ciascuna contenente dati per una delle dita, che a sua volta rappresenta i dati ossei con struct di tipo **Bone**, fornite dall'API di Unity.

Unificazione del sistema di riferimento

In fase di acquisizione dei dati va notato poi che ogni dispositivo utilizzerà un proprio sistema di riferimento per definire posizioni e rotazioni. infatti raro che provider differenti utilizzino lo stesso sistema di riferimento. Per questo motivo è necessario applicare una trasformazione lineare ai dati in ingresso, in modo da unificare i loro sistemi di riferimento. Per fare ciò bisogna prima di tutto definire il sistema di riferimento comune da utilizzare, e in seguito identificare la trasformazione da applicare ai dati forniti dal provider per ottenerne i dati equivalenti in tale sistema.

5.1.3 Integrazione nello Unity Input System

Una volta acquisiti e trasformati adeguatamente i dati di tracciamento, questi andranno forniti all'Input System descritto nel Capitolo 2, tramite l'API che esso mette a disposizione.

Le classi dell'Input System

I Layout vengono definiti tramite delle apposite interfacce: per i Control va implementata l'interfaccia *InputControl*, per i Device *InputDevice*. Da notare che InputDevice è in realtà una specializzazione di InputControl: in questo modo la gerarchia può essere definita sulle interfacce InputControl sfruttando il pattern **Composite**, che permette proprio di definire una relazione ad albero tra elementi di una stessa classe.

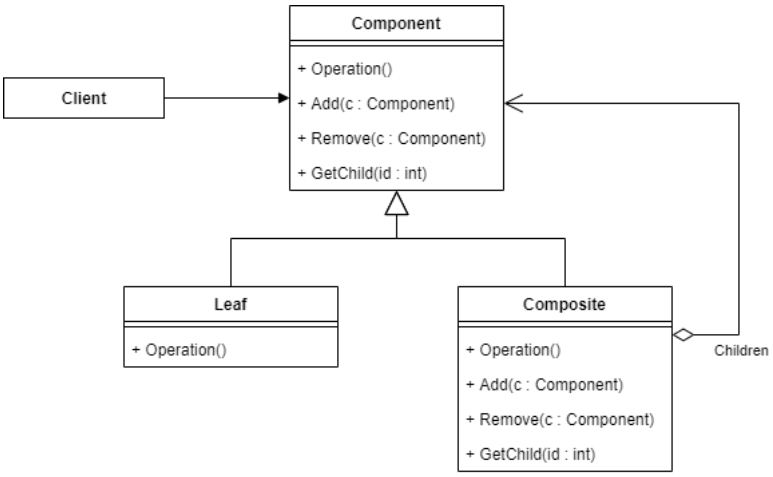


Figura 5.3: Class Diagram del pattern Composite

I Memory Layout vengono invece definiti con delle struct, che implementano l'interfaccia **IInputStateTypeInfo**. Ogni struct così definita ha come unico vincolo la definizione di un ID di 4 caratteri, che identifica univocamente il Memory Layout, gli altri campi della struct rappresentano invece le locazioni di memoria dedicate ai Control.

Creazione degli Input Device

La definizione degli InputDevice rappresenta il fulcro della struttura del modulo di input. È stata definita una classe che implementa *InputDevice*, chiamata **TrackedHandDevice**. Questa rappresenta il layout di un dispositivo che l'Input System definisce **sintetico**, ovvero che fornisce dati non effettivamente corrispondenti a misurazioni reali, ma generati a livello software. Questa è evidentemente la descrizione migliore per il servizio Ultraleap, che genera dati di tracciamento tramite computer vision.

Per rappresentare dati di tracciamento scheletrico, Unity mette a disposizione soltanto un Control da associare ai dati relativi ad un singolo osso, **BoneControl**. Per fornire una rappresentazione dei dati esaustiva è stato quindi definito un nuovo tipo di Control, **FingerControl**, per inserire nell'Input System i dati delle struct *Finger*.

TrackedHandDevice fornisce i Control fondamentali per la rappresentazione dei dati di hand tracking: posizione e rotazione delle 21 giunzioni del modello osseo, raggruppate quando possibile per dita. Nel complesso quindi, il Layout descritto da TrackedHandDevice (a cui viene associato il nome *Tracked Hand*) fornisce due Control per i dati del polso (Vector3Control e QuaternionControl) e cinque FingerControl.

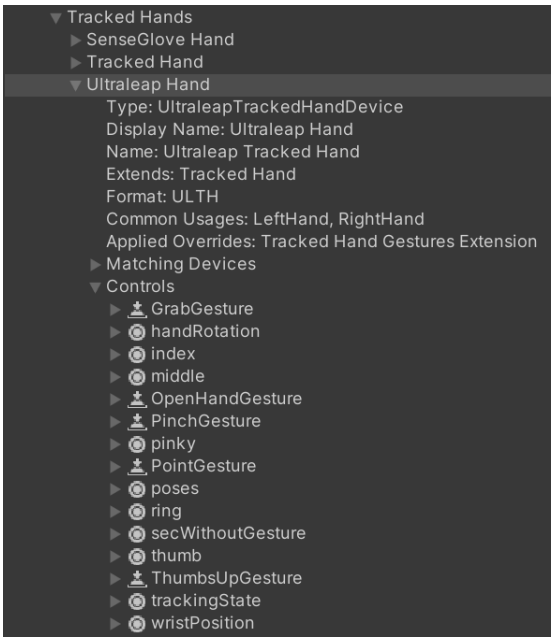


Figura 5.4: Schermata dello Unity Editor che mostra le caratteristiche e la struttura del layout Ultraleap Hand

L'estensibilità del sistema in questo contesto è garantita proprio dalla capacità espressiva delle interfacce dell'Input System: per definire un nuovo Device di hand tracking è infatti sufficiente estendere TrackedHandDevice, con un nuovo Device più specifico che rappresenti il sistema che si intende introdurre nell'Input System. Nel caso specifico di Ultraleap, è stato quindi definita la classe **UltraleapTrackedHandDevice**, che definisce il Layout che verrà associato a ciascuna mano virtuale riconosciuta dal sistema di hand tracking.

Input Feeding

Il processo di input feeding dei Control, modellato dal Sequence Diagram nella figura 5.6, viene gestito internamente a TrackedHandDevice e le sue eventuali specializzazioni. Oltre ad *InputDevice* la classe implementa infatti un'altra interfaccia, **IInputUpdateCallbackReceiver**. Una classe che implementa tale interfaccia può definire una funzione di callback **OnUpdate**, la quale verrà invocata in risposta a qualunque evento che richieda l'aggiornamento dei dati di input. Tramite questo metodo è quindi possibile aggiornare quando neccessario il contenuto dei Control, utilizzando interfacce fornite dall'API dell'Input System. Questa interfaccia è stata utilizzata in combinazione con il design pattern **Template Method**, tramite il quale si definisce lo scheletro di un algoritmo, localizzandone le parti invarianti e delegando alle sottoclassi la definizione degli step variabili.

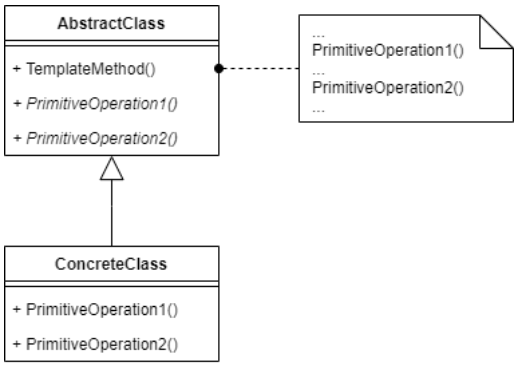


Figura 5.5: Class Diagram del pattern Template Method

Nel nostro caso il metodo da astrarre è proprio *OnUpdate*, il quale è composto da tre componenti:

1. Conversione dei dati di tracciamento correnti
2. Se i dati sono assenti, aggiornamento conseguente dello stato del Device e dei suoi Control
3. Altrimenti, aggiornamento dei Control con i dati ricevuti

Per questo motivo sono stati definiti dei metodi astratti per ciascuna di queste fasi del ciclo di input feeding, rispettivamente *AdaptCurrentTrackingData*, *NoInputReceived* e *UpdateState*. L'implementazione di tali metodi potrà quindi variare separatamente per ogni dispositivo collegato al modulo di input, e in particolare *AdaptCurrentTrackingData* dovrà essere necessariamente reimplementato a seconda del dispositivo che si desidera integrare in Unity.

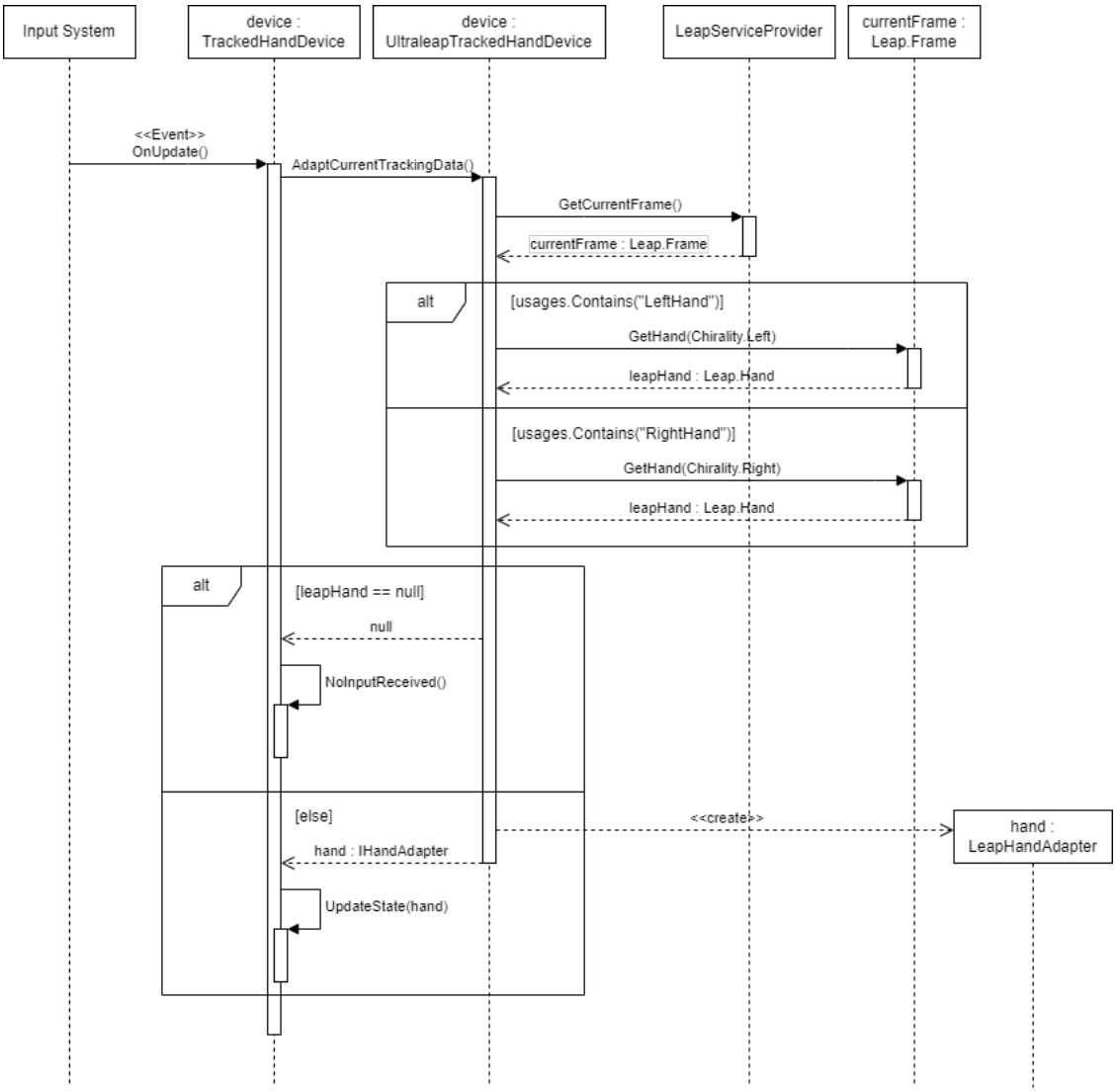


Figura 5.6: Sequence Diagram del processo di input feeding

5.1.4 Class Diagram di design dell'Input Module

La figura 5.7 mostra il design finale dell'Input Module, modellato con un class diagram UML. Le interfacce facenti parte di API esterne (Ultraleap Plugin, Unity Input System, API Unity base) sono indicate in notazione *lollipop*.

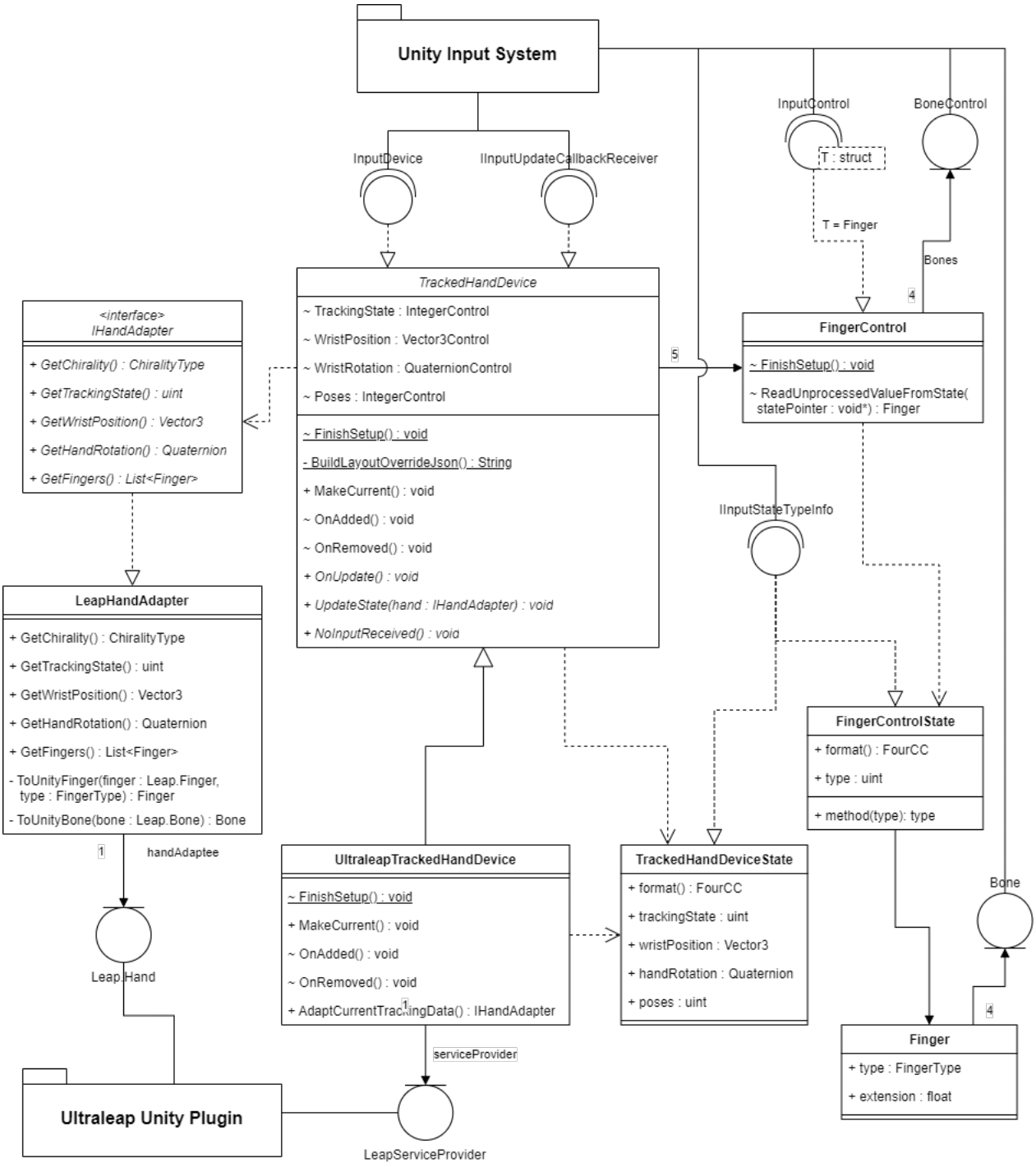


Figura 5.7: Class Diagram di Design dell'Input Module

5.2 Interaction Module

Dopo aver ultimato la progettazione del modulo di input, è iniziato il lavoro di design riguardante il secondo modulo del sistema. Tale modulo prende il nome di **Interaction Module**, e lavora sulla base dei dati forniti dall'Input Module permettendo di utilizzare i device integrati nel sistema per interagire con le applicazioni XR, mediante un algoritmo per il riconoscimento di pose statiche.

5.2.1 Algoritmo per il riconoscimento delle pose statiche

La funzionalità di gran lunga principale del modulo di interazione è il suo algoritmo per il riconoscimento delle pose statiche. È stata scelta una soluzione basata su una rappresentazione vettoriale delle pose della mano, per la cui piena comprensione saranno necessari alcuni rimandi a nozioni di algebra lineare.

Space Vector Quantization (SVQ)

Per il riconoscimento delle pose è stato utilizzato un algoritmo di partizionamento vettoriale, chiamato **Space Vector Quantization (SVQ)**.

Le pose sono state rappresentate in forma di vettori del tipo:

$$\underline{p} = \begin{pmatrix} a_1 & \cdots & a_5 \end{pmatrix} \in [0,1]^5$$

dove ciascun elemento $a_i \in [0,1]$ è un valore normalizzato che codifica l'**abduzione** di una delle dita, ovvero quanto questa è "flessa" verso il palmo. Questa codifica viene applicata sia alla posa attuale della mano che alle pose che si richiede di riconoscere. Così facendo si avrà un insieme finito $P \subset [0,1]^5$ di pose da riconoscere. Questo insieme può essere utilizzato per definire una **partizione di Voronoi** sullo spazio vettoriale su cui è definito.

Per fare chiarezza, sono necessarie prima alcune nozioni di geometria euclidea:

Definizione 1 (Distanza euclidea tra vettori)**.** Dato uno spazio euclideo $S \subseteq \mathbb{R}^n$, e dati due vettori:

$$\underline{a} = (a_1 \cdots a_n) \in S$$
$$\underline{b} = (b_1 \cdots b_n) \in S$$

la distanza euclidea tra $\underline{a}$ e $\underline{b}$ è definita come:

$$d(\underline{a}, \underline{b}) = \sqrt{(a_1 - b_1)^2 + \cdots + (a_n - b_n)^2} = \sqrt{\sum_{k=1}^{n} (a_k - b_k)^2}$$

Definizione 2 (Distanza euclidea vettore-insieme)**.** Dati uno spazio euclideo $S \subseteq \mathbb{R}^n$, un vettore $\underline{a} \in S$, e un sottoinsieme di vettori $B \subset S$, si definisce la distanza euclidea di $\underline{a}$ da B come:

$$d(\underline{a}, B) = \min_{\underline{b} \in B} d(\underline{a}, \underline{b})$$

Sulla base di queste definizioni possiamo definire correttamente una partizione di Voronoi:

Definizione 3 (Partizione di Voronoi)**.** Dati uno spazio euclideo S e un insieme finito $P \subset S$, si definisce partizione di Voronoi di S per P la partizione di S:

$$\mathcal{V}(S; P) = \{S(\underline{p}) \subseteq S \mid \underline{p} \in P\}$$

dove i sottoinsiemi $S(\underline{p})$ della partizione sono definiti come:

$$S(\underline{p}) = \{\underline{s} \in S \mid d(\underline{s}, \underline{p}) = d(\underline{s}, P)\}$$

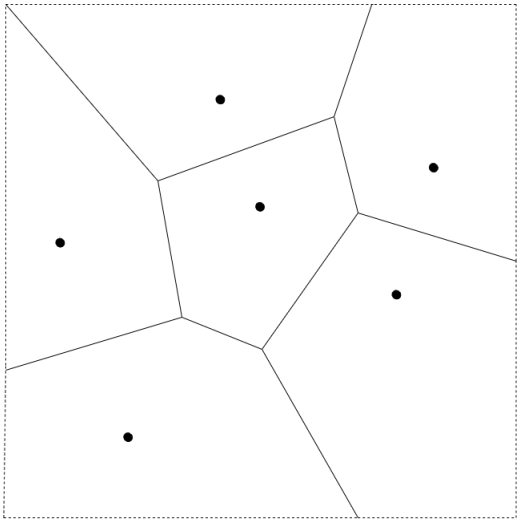


Figura 5.8: Esempio di una partizione di Voronoi per il piano

Una partizione di Voronoi associa quindi ad ogni vettore $\underline{p} \in P$, spesso chiamati **centroidi**, l'insieme dei punti dello spazio che sono distanti da $\underline{p}$ meno che da qualunque altro vettore di P. Il risultato è un partizionamento *parziale* dello spazio (parziale in quanto vi saranno dei punti equidistanti da due o più centroidi).

L'algoritmo SVQ applica le precedenti definizioni per **discretizzare uno spazio euclideo**: dato un sottoinsieme finito (e in genere ristretto) dei suoi vettori, che rappresentano i centroidi, utilizza la corrispondente partizione di Voronoi come **insieme quoziente**[2] del dominio, e associa un vettore in input al centroide associato secondo la relazione di equivalenza indotta sullo spazio dalla partizione.

È semplice a questo punto utilizzare SVQ per attuare un algoritmo di riconoscimento sulle pose statiche fornite:

- Il dominio delle istanze del problema è lo spazio delle codifiche vettoriali delle pose statiche, $[0, 1]^5$

[2]dati un insieme A e una relazione di equivalenza totale r sugli elementi di A, è l'insieme delle classi di equivalenza di A rispetto ad r

- L'istanza è descritta dall'insieme $P \subset [0,1]^5$ delle pose da riconoscere, che rappresenteranno i centroidi, e dalla posa $p \in [0,1]^5$ fornita in input
- Possiamo utilizzare la partizione di Voronoi $\mathcal{V}([0,1]^5; P)$ come insieme quoziente del dominio, e confrontare la posa in input con le classi di equivalenza così ottenute, ciascuna associata ad una delle pose da riconoscere.

La distanza tra la posa attuale e una delle pose del sistema rappresenterà una misura di **dissimilarità** tra le due: al diminuire della distanza, le due pose verranno considerate via via più similari, e la scelta finale ricadrà sulla posa con la similarità maggiore rispetto a quella in input.

Così facendo abbiamo un modo per associare in maniera quasi univoca una posa in input ad una di quelle che si desidera riconoscere: la posa attuale della mano verrà associata alla posa statica dalla minore distanza vettoriale tra quelle disponibili. Tuttavia la partizione di Voronoi da sola non è sufficiente a soddisfare la richiesta del problema, per due motivi:

1. Come già detto, vi saranno pose "di confine" tra due classi di equivalenza, che quindi il sistema non potrà associare univocamente ad un sottoinsieme
2. Il partizionamento genera una relazione di equivalenza totale sullo spazio delle pose, ma non necessariamente la posa attuale deve essere ricondotta ad una delle pose da riconoscere

Per risolvere entrambi questi problemi è sufficiente attuare una semplice soluzione: fornire all'algoritmo una **soglia di dissimilarità** oltre la quale la posa candidata non verrà riconosciuta. Al termine del confronto quindi, la distanza tra l'input e il centroide candidato verrà confrontata con una certa soglia, ritenuta sufficientemente alta, e nel caso in cui dovesse superarla (e le pose non fossero quindi sufficientemente simili) il centroide verrà scartato e l'input non verrà associato ad alcuna posa.

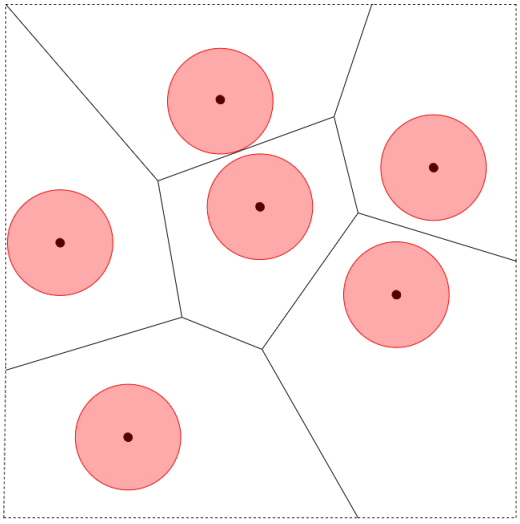


Figura 5.9: Esempio grafico di SVQ con l'introduzione del threshold

L'algoritmo proposto associa l'input alle pose statiche in maniera sufficientemente precisa per lo scopo del sistema. Tuttavia vi sono altri problemi che sorgono non dalla definizione dell'algoritmo stesso, ma dal suo utilizzo nel contesto dell'intero sistema, e che verranno discussi di seguito.

Problemi legati all'utilizzo real-time

Per prima cosa va fatto presente che l'algoritmo SVQ verrà utilizzato in un sistema real-time, con dati di input soggetti a frequenti fluttuazioni, e il suo output non solo avrà necessariamente le stesse caratteristiche ma sarà anche visibile direttamente all'utente finale del sistema, sotto forma di interazione con l'applicativo. Dal punto di vista pratico questo ha un effetto significativo sul sistema: nel caso in cui l'input fosse al limite tra più classi di equivalenza, l'output dell'algoritmo sarebbe soggetto a frequenti fluttuazioni, cosa che porterebbe l'applicazione ad avere un comportamento instabile. A titolo d'esempio, la posa tipicamente più utilizzata in sistemi di questo tipo è il *grab*, ovvero la chiusura del pugno, che in genere viene associata al gesto di afferrare un oggetto. Nel caso in cui l'input si trovasse al limite della classe di equivalenza del *grab*, il sistema passerebbe continuamente dall'afferrare un oggetto al rilasciarlo, con conseguenti difficoltà nell'utilizzo dell'applicazione.

Per attaccare il problema si è deciso di inserire un piccolo intervallo di tolleranza in fase di modifica della posa attuale (sezione dell'Interaction Module che verrà discussa più avanti). Quando la posa corrente viene modificata, il sistema invece di modificarla immediatamente attende un certo intervallo di tempo predefinito (nella versione finale del modulo 0.3s). Se durante l'intervallo di tempo la posa riconosciuta cambia ancora, il sistema rinuncia a modificare la posa attuale e, se la nuova posa è ancora differente da quella attuale, il timer ricomincia; se invece l'intervallo di tempo passa senza che la posa riconosciuta cambi, questa viene impostata come nuova posa corrente.

Problemi legati alla saturazione dello spazio di ricerca

Come già detto l'algoritmo cerca di mantenere delle zone dello spazio di ricerca non associate ad alcuna posa, introducendo una distanza massima dal centroide per ciascuna classe di equivalenza. Nonostante ciò il problema torna ad essere preponderante nel momento in cui lo spazio tra i centroidi si riduce, cosa che può avvenire per due motivi:

1. Numero eccessivo di centroidi (saturazione dello spazio di ricerca)
2. Concentrazione eccessiva di centroidi in una certa zona dello spazio (saturazione di una regione dello spazio di ricerca)

Il primo problema non è facilmente evitabile ed è legato più all'utilizzo che si fa dell'algoritmo: da esso si evince che SVQ non è adatto a contesti in cui il numero di classi tra cui distinguere è elevato rispetto all'estensione dello spazio di ricerca. Fortunatamente non è il caso dell'Interaction Module, dato che il numero medio

di pose che andrebbero riconosciute in una applicazione XR non supera le 10-15.

Ben più problematico è invece il secondo problema, che può verificarsi indipendentemente dal numero di pose che si vuole riconoscere. Se anche buona parte della sua prevenzione sta in una buona scelta delle pose da riconoscere, compito che spetterà agli utilizzatori del Tracking Adapter, tuttavia la prevenzione a priori diventa importante per un caso in particolare: quello della mano aperta.

Per gli utilizzatori di una applicazione XR di questo tipo, viene naturale associare la mancanza di interazione con l'apertura completa della mano, a segnalare al sistema che non si sta eseguendo alcuna azione in particolare. Tuttavia anche la posizione della mano aperta occupa un valore nello spazio delle pose che abbiamo definito. Di conseguenza, se tale valore dovesse venire inglobato dalla classe di equivalenza di una qualche posa l'interazione dell'utente con il sistema diventerebbe di colpo molto meno intuitiva.

La soluzione scelta per risolvere il problema può sembrare tanto banale quanto è in verità efficace: il sistema dedica **una classe di equivalenza** proprio alla posa **"mano aperta"**, assicurando così sempre che quella specifica porzione dello spazio non venga associata ad una posa definita dall'utente. Naturalmente il buon senso nella definizione delle pose da parte degli utilizzatori del sistema resta importante, ma questa soluzione fornisce nella maggior parte dei casi una maggiore stabilità all'algoritmo.

5.2.2 Design Object-Oriented dell'algoritmo

A questo punto l'algoritmo appena illustrato dovrà essere integrato nel sistema object-oriented che si sta progettando. Per farlo sono state definite delle classi apposite, cercando di ridurre al minimo il loro accoppiamento. Inoltre, per rendere l'Interaction Module semplice da utilizzare, è necessario fornire agli sviluppatori degli strumenti per definire facilmente e ad alto livello le pose da riconoscere, senza richiedergli di eseguire ulteriori operazioni per inserirle nell'algoritmo di riconoscimento.

Definizione delle pose statiche

Per la definizione delle pose da riconoscere è stata utilizzata una particolare classe dell'API di Unity, **ScriptableObject**. Uno ScriptableObject è un oggetto serializzabile e istanziabile dallo Unity Editor, il cui contenuto viene memorizzato in un asset con estensione *.scriptableobject*. L'engine mette a disposizione questa classe come supporto al Data Oriented Programming (DOP), ovvero la programmazione orientata all'ottimizzazione dell'utilizzo di memoria, in quanto permettono di mantenere una sola istanza dei dati utilizzabile da più classi, senza bisogno di duplicarli. Il contenuto di un'istanza di ScriptableObject è modificabile dall'editor, ma vi si può accedere dagli script tramite la sua posizione nel file system.

È stata definita una classe **StaticGesture** che estende ScriptableObject, con-

tenente tutti i dati necessari a definire una posa da riconoscere. Si è scelto per semplicità di definire una posa direttamente nella sua forma vettoriale (tramite l'abduzione delle dita), ma nulla toglie la possibilità di definirle in forma estesa con dati rototraslazionali, sulla quale torneremo nelle conclusioni.

Così facendo, gli sviluppatori che volessero definire nuove pose da integrare nel sistema dovranno semplicemente definire dall'editor un nuovo asset di tipo *StaticGesture*, e inserire i dati rappresentativi della posa.

Esecuzione dell'algoritmo

Incaricata dell'esecuzione dell'algoritmo di riconoscimento è la classe **GestureComparer**: il suo costruttore accetta una lista di pose, in forma di oggetti *StaticGesture*, che rappresenta le pose su cui si vuole eseguire SVQ. La classe mette poi a disposizione due metodi per richiedere l'esecuzione effettiva dell'algoritmo. Il primo, *CompareToGestures*, accetta un oggetto *IHandAdapter* contenente i dati di tracciamento, sul quale esegue l'algoritmo di riconoscimento. Il secondo metodo è invece *GetBestGesture*, il quale restituisce la posa riconosciuta dall'algoritmo (*null* se nessuna è valida). Questa separazione tra esecuzione e lettura del risultato è stata scelta per due motivi:

1. Separa l'algoritmo SVQ (CompareToGestures) dalle strategie di riconoscimento applicate a posteriori (GetBestGestures), aumentando la manutenibilità del codice
2. Permette di leggere più volte il risultato di un confronto senza bisogno di memorizzarlo altrove oppure di eseguire nuovamente l'algoritmo

Utilizzo nel ciclo di input feeding

Il riconoscimento dovrà quindi essere eseguito costruendo un oggetto di tipo GestureComparer con una lista contenente tutte le pose definite nel progetto. I metodi dell'oggetto andranno poi utilizzati per eseguire il confronto con la posa attuale della mano: ciò andrà fatto ad ogni ciclo di aggiornamento dell'Input System, per eseguire il confronto sui dati ricevuti dal frame corrente.

Per recuperare tutte le StaticGesture definite nel progetto, è stata definita una classe statica **GestureLoader**, che si occupa proprio di cercare tali asset nel file system e di raccoglierli in una lista, la quale potrà poi essere utilizzata come argomento del costruttore di GestureComparer.

Per quanto riguarda la costruzione di questi oggetti e il loro utilizzo, questi compiti spetteranno inevitabilmente a TrackedHandDevice e le sue eventuali specializzazioni, in quanto sono loro incaricati della gestione del ciclo di input feeding. TrackedHandDevice dovrà quindi possedere un riferimento ad un oggetto di tipo GestureComparer, costruito come sopra, il quale durante il ciclo di input feeding, definito tramite *OnUpdate*, verrà utilizzato per eseguire il confronto tra le pose definite e i dati di tracciamento correnti.

Passaggio alla game logic

Il risultato dell'algoritmo di riconoscimento dovrà essere notificato all'Input System, per renderlo utilizzabile nella logica di gioco dell'applicazione. Dato che il riconoscimento viene effettuato durante l'aggiornamento dell'Input Device, l'approccio migliore sarebbe fornire il risultato del riconoscimento tramite dei Control di tipo *Button*: quando una posa viene riconosciuta, il Control corrispondente si attiva, notificando di conseguenza tutte le Action a lui collegate. Questa soluzione crea però un ulteriore problema: diventa necessario definire dinamicamente i Control associati a TrackedHandDevice, poichè questi dipenderanno dalle pose statiche definite nel progetto.

Fortunatamente, l'Input System mette a disposizione delle API proprio per queste necessità. È possibile infatti definire a run time un **Layout Override**, ovvero una modifica ad un Layout preesistente. Un Layout Override permette ad esempio di aggiungere nuove proprietà ad un Layout, in modo da associarlo a dispositivi non previsti in origine. Ma ciò che torna utile nel contesto dell'Input Adapter è la possibilità di definire nuovi Control associati ad un Layout proprio tramite un suo Override. Va però fatto presente che un Override non permette di modificare il Memory Layout associato, quindi gli eventuali nuovi Control dovranno basarsi sui dati contenuti nel Memory Layout originale.

La soluzione finale è stata quindi aggiungere al Memory Layout di TrackedHandDevice una variabile intera dedicata alle pose statiche da riconoscere. Questa variabile agisce come una bitmap: ciascun bit viene associato ad una posa da riconoscere, e il suo valore corrisponde all'attivazione o meno della posa corrispondente. Per quanto riguarda la definizione dinamica dei Control, questa avviene in fase di inizializzazione del Layout, prima raccogliendo tutte le pose da riconoscere (tramite la classe GestureLoader descritta in precedenza), poi definendo un Layout Override che aggiunge un Control per ciascuna StaticGesture trovata, ciascuno associato ad una posizione nella bitmap dedicata.

5.2.3 Class Diagram di Design dell'Interaction Module

Il class diagram UML nella figura 5.10 illustra il design finale dell'Interaction Module.

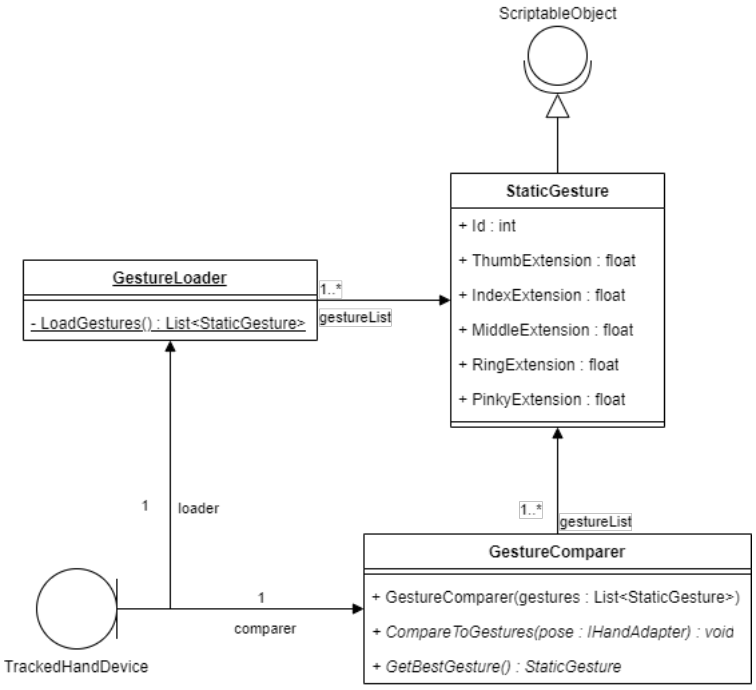


Figura 5.10: Class Diagram di Design dell'Interaction Module

Capitolo 6

Implementazione del Sistema

Una volta definito il software design dei due moduli, si è passati alla loro implementazione. In questo capitolo si entrerà quindi nel dettaglio degli strumenti utilizzati in questa fase dello sviluppo, prestando particolare attenzione al modo in cui queste hanno facilitato la traduzione in codice delle scelte di design precedentemente fatte.

Verrà poi presentato un esempio di utilizzo dell'Input Adapter, all'interno di un applicativo preesistente sviluppato in Unity, per verificarne e validarne le funzionalità.

6.1 Linguaggi di programmazione

Il linguaggio principale utilizzato per implementare il sistema è **C#**, proprietario di Microsoft e al centro del suo framework *.NET*. La scelta risulta quasi ovvia trattandosi di un sistema finalizzato all'utilizzo in applicazioni sviluppate per lo Unity Engine, che fa proprio di C# il suo linguaggio di scripting principe. Tuttavia è errato pensare che questa sia l'unico vantaggio nell'utilizzo di questo linguaggio: C# ha caratteristiche ottime per la natura del software sviluppato, e mette a disposizione una serie di strumenti che sono stati largamente utilizzati in fase di implementazione.

C# ha alla base un'idea ben precisa: unire la libertà fornita da C++ alla potenza espressiva object-oriented e la maggiore semplicità d'uso di Java. Da questa filosofia risulta un linguaggio molto più esplicitamente object-oriented di C++, ma meno rigido di Java nei principi da rispettare. Questo permette di implementare un design a oggetti in maniera semplice e intuitiva, anche se spesso non totalmente rappresentabile tramite diagrammi UML. A titolo di esempio: nell'implementazione C# dello Unity Input System si fa largo uso di oggetti **Attribute**, elementi simili ai *Tag* Java, i quali per loro natura solo complessi da rappresentare in un Class Diagram UML.

Altra caratteristica del linguaggio ampiamente utilizzata nel progetto sono le

cosiddette **proprietà** (*properties*), particolari membri di classe che forniscono un meccanismo flessibile per leggere, scrivere o calcolare campi privati. Formalmente le proprietà sono composte da metodi speciali, detti **metodi di accesso** (*accessors*). Una property può avere tre tipi di accessors (*init*, *get*, *set*), in qualunque combinazione e con qualunque modificatore di accesso. Ciò fornisce quindi una enorme flessibilità permettendo di definire con lo stesso meccanismo sia dei classici getter-setter, che dei campi read-only, campi calcolati a run-time e altro ancora.

In forma molto più limitata è stato utilizzato anche JSON, in particolare per rappresentare dati all'interno di file di configurazione (scopo a cui si presta perfettamente) e per eseguire modifiche a run-time ai layout degli Input Device. Per completezza di informazione, ricordiamo però che JSON non è considerabile un vero e proprio linguaggio (né di markup, come XML, né tantomeno di programmazione), ma rappresenta unicamente un formato di rappresentazione dei dati.

6.2 Ambiente di sviluppo

Il fulcro dell'ambiente di sviluppo utilizzato è rappresentato dallo **Unity Editor**. Questo fornisce tutti gli strumenti tipici di un moderno game engine general-purpose, come un editor di scena, la possibilità di definire oggetti di scena predefiniti, strumenti per il debugging e altro.

Come principale IDE è stato scelto **Visual Studio**, che oltre ad essere uno dei principali IDE utilizzati per soluzioni industry-grade, e in particolare quello consigliato per lo sviluppo in C#, supporta anche il debugging di progetti per Unity Engine, collegandosi allo Unity Editor e permettendo di utilizzarli in sinergia.

6.3 Analisi del codice implementativo

Verranno ora commentati alcuni frammenti di codice presi dall'implementazione finale del sistema. Sono state scelte le sezioni di codice che meglio rappresentano il sistema e l'insieme delle sue funzionalità. Gran parte di esse sono tuttavia troppo corpose per essere inserite integralmente in questa sezione della tesi, e sono quindi state aggiunte come documenti in Appendice.

6.3.1 Implementazione dell'Input Module

IHandAdapter

La classe astratta *IHandAdapter* fornisce l'interfaccia generica dei Wrapper per dati di tracciamento esterni. Oltre a dei metodi per accedere a questi ultimi, ne definisce anche altri per rappresentare la chiralità della mano (mano destra/mano sinistra) e lo stato di tracciamento: questo di base indica solo se la mano fornisce

attualmente dati o meno, ma può essere utilizzato per trasmettere informazioni aggiuntive, come il grado di accuratezza dei dati correnti.

```
public abstract class IHandAdapter
{

    public enum ChiralityType
    {
        Invalid = -1,
        Left = 0,
        Right = 1
    }

    public abstract ChiralityType Chirality { get; }

    public abstract uint TrackingState { get; }

    public abstract Vector3 WristPosition { get; }
    public abstract Quaternion HandRotation { get; }

    public abstract IReadOnlyList<Finger> Fingers { get; }
}
```

Listing 6.1: L'interfaccia IHandAdapter

Come si può vedere è stato fatto largo uso delle *properties* di C#, in quanto si prestano molto bene a definire getter-setter dei campi di una classe. In questo caso vanno a definire solo le firme dei getter dei vari dati richiesti dall'Input Module ai sistemi esterni, i quali andranno implementati per ciascuna possibile sorgente dati.

LeapHandAdapter

La classe *LeapHandAdapter* (appendice A.1) è l'implementazione dell'interfaccia descritta nella sezione precedente, specializzata per i dati di tracciamento di Ultraleap.

```
    public override Vector3 WristPosition
    {
        get
        {
            if (handAdaptee == null)
                return Vector3.zero;

            return handAdaptee.WristPosition;
        }
    }
```

Listing 6.2: Esempio di implementazione di un getter in LeapHandAdapter

Le implementazioni dei getter restituiscono dati a cui la classe attinge dal campo *handAdaptee*, il quale altro non è che un oggetto di tipo *Hand* dell'API di Ultraleap, fornito nel costruttore e contenente i dati di tracciamento di una mano riconosciuta al frame corrente.

```
    public LeapHandAdapter(Leap.Hand leapHand)
    {
        if (leapHand == null)
        {
            throw new ArgumentException();
        }

        this.handAdaptee = leapHand;
        SetFingers(leapHand);
    }
```

Listing 6.3: Costruttore di LeapHandAdapter

TrackedHandDeviceState

Il nucleo dell'intero modulo di input è di fatto rappresentato dalla classe che implementa il principale Input Device utilizzato come interfaccia con l'Input System, *TrackedHandDevice*. Prima di soffermarci su di essa, è tuttavia utile mostrare come viene implementato il memory layout ad essa associato.

La struct *TrackedHandDeviceState* (appendice A.2) implementa appunto il memory layout di *TrackedHandDevice*. Al suo interno, i campi costituiscono l'effettiva ripartizione del blocco di memoria dedicato allo stato attuale del dispositivo, eccetto il campo *format*, che indica un codice identificativo del layout.

```
  [Preserve, InputControl(name = "wristPosition", layout = "Vector3",
     displayName = "Wrist Position", noisy = true, dontReset = true)]
  public Vector3 wristPosition;
```

Listing 6.4: Esempio di un campo della struct TrackedHandDeviceState

A ciascun attributo di layout sono poi stati associati due oggetti Attribute: *Preserve* assicura che il processo di build di Unity non elimini quella specifica sezione di codice, cosa che l'engine fa con istruzioni inutilizzate per ridurre le dimensioni dell'applicazione finale; *InputControl* permette invece di riconoscere quello specifico campo come contenitore dello stato di un Input Control figlio del device descritto, specificandone il tipo, la codifica, l'id e altro.

Il memory layout così definito potrà poi essere associato all'effettiva classe descrivente il Layout come di seguito:

```
[Preserve, InputControlLayout(displayName = "Tracked Hand",
    isGenericTypeOfDevice = true,
  commonUsages = new[] { "LeftHand", "RightHand" },
  stateType = typeof(TrackedHandDeviceState))]
public abstract class TrackedHandDevice : InputDevice,
    IInputUpdateCallbackReceiver
```

Listing 6.5: Dichiarazione della classe TrackedHandDevice

TrackedHandDevice

L'interfaccia generica definita tra l'Input System ed i sistemi di hand tracking esterni, *TrackedHandDevice* (appendice A.3), rappresenta il fulcro del modulo di input, e di fatto è la classe più corposa dell'intero Input Adapter.

Nella sua firma, mostrata poco più sopra nel frammento di codice 6.5, si possono osservare varie caratteristiche che vengono associate alla sua definizione: ad essa viene associato un nuovo Attribute, *InputControlLayout*, che identifica la classe come descrizione del Layout di un Input Control (più nello specifico un Input Device). Tale Attribute accetta anche altri parametri aggiuntivi: il nome mostrato a schermo del Layout ("Tracked Hand"), se il device è da considerare "generico" (ovvero se ammette specializzazioni nella Control Hierarchy), utilizzi frequentemente associati e il memory layout che utilizza.

Gli Input Control figli del Device vengono poi rappresentati come attributi della classe, nello specifico riferimenti a oggetti che implementano l'interfaccia *InputControl*. Lo specifico tipo dell'attributo rappresenterà il tipo esatto di Control che si vorrà associare (Integer, Vector3, Quaternion, Bone ecc.).

```
1  public IntegerControl TrackingState { get; protected set; }
2  public Vector3Control WristPosition { get; protected set; }
3  public QuaternionControl HandRotation { get; protected set; }
4
5  public IReadOnlyList<FingerControl> Fingers => m_FingerList;
6  protected List<FingerControl> m_FingerList;
7
8  protected IntegerControl Poses { get; set; }
9
10 public IReadOnlyDictionary<StaticGesture, ButtonControl> Gestures =>
       m_Gestures;
11 protected Dictionary<StaticGesture, ButtonControl> m_Gestures;
```

Listing 6.6: Attributi di tipo Control di TrackedHandDevice

Gran parte del comportamento del Device è poi data dal metodo *OnUpdate*, che come già detto (sez. 5.1.3) viene invocato in risposta ad un evento dell'Input System che richieda l'aggiornamento dello stato del Device. La callback verificherà l'effettiva presenza di dati di input aggiornati tramite il metodo astratto *AdaptCurrentTrackingData*, che le sottoclassi implementeranno per definire la sorgente dati specifica di ciascun *TrackedHandDevice*.

```
1  public void OnUpdate()
2  {
3    IHandAdapter hand = AdaptCurrentTrackingData();
4
5    if (hand == null) NoInputReceived();
6    else UpdateState(hand);
7  }
```

Listing 6.7: Metodo OnUpdate di TrackedHandDevice

Se non dovessero esserci nuovi dati di input, allora lo stato del Device verrà posto ai valori di default con il metodo *NoInputReceived*. Altrimenti, verrà invocato *UpdateState*, che utilizzerà i dati forniti dalla sorgente (e incapsulati in un *IHandAdapter*) per aggiornare il contenuto degli Input Control del *TrackedHandDevice*. L'aggiornamento di ciascun control avviene a sua volta tramite un evento dell'Input System, che il Control gestirà in maniera analoga a quella appena descritta.

```
public virtual void UpdateState(IHandAdapter hand)
{
  InputSystem.QueueDeltaStateEvent<uint>(TrackingState, hand.
     TrackingState);

  //Update Wrist Position
  InputSystem.QueueDeltaStateEvent<Vector3>(
    WristPosition, hand.WristPosition);

  //Update Hand Rotation
  InputSystem.QueueDeltaStateEvent<Quaternion>(
    HandRotation, hand.HandRotation);

  //Update finger positions/rotations
  foreach (Finger finger in hand.Fingers)
  {
    InputSystem.QueueDeltaStateEvent<FingerControlState>(
      GetFingerControl(finger.type), ToFingerControlState(finger));
  }
```

Listing 6.8: Aggiornamento dei Control nel metodo UpdateState di TrackedHandDevice

UltraleapTrackedHandDevice

Il Layout generico è stato poi specializzato in uno che fungesse da interfaccia con il sistema Ultraleap: è stata quindi definita la classe *UltraleapTrackedHandDevice* (appendice A.4), che implementa *TrackedHandDevice* e più precisamente il metodo *AdaptCurrentTrackingData*. Questo come già detto verifica se vi sono dei nuovi dati di input con cui aggiornare lo stato del Device, ed eventualmente li restituisce, mettendoli a disposizione del metodo invocatore *OnUpdate*.

```
public override IHandAdapter AdaptCurrentTrackingData()
{
  serviceProvider ??= UnityEngine.Object.FindObjectOfType<Leap.Unity.
     LeapServiceProvider>();

  Leap.Hand leapHand = serviceProvider?.CurrentFrame?.GetHand(
    (usages.Contains(CommonUsages.LeftHand)) ? Chirality.Left :
       Chirality.Right);

  if (leapHand == null)
  {
    return null;
```

```
    }

    return new LeapHandAdapter(leapHand);
  }
}
```

Listing 6.9: Implementazione del metodo AdaptCurrentTrackingData di UltraleapTrackedHandDevice

Nel caso di Ultraleap, la sorgente dei dati di tracciamento è l'oggetto *LeapServiceProvider* del plugin del servizio. Il metodo implementato quindi, se necessario lo cerca prima tra gli oggetti attualmente in scena. Se lo trova (oppure ne aveva già il riferimento), prova ad ottenerne dei dati di tracciamento relativi alla mano corrispondente al Device a cui fa riferimento. Infine, in caso di successo incapsula questi dati in un *IHandAdapter* e li restituisce, in tutti gli altri casi restituirà un riferimento nullo.

6.3.2 Implementazione dell'Interaction Module

La rappresentazione vettoriale delle pose statiche da riconoscere è stata implementata nella classe *StaticGesture* (appendice B.1).

```
    [Header("Thumb")]
    [Range(0,1)]
    public float ThumbExtensionAvg = 0;
```

Listing 6.10: Esempio di un campo della struct TrackedHandDeviceState

Gli Attribute di C# associati ai vari campi sono utili allo Unity Editor: in particolare, *Range* fa in modo che il valore float relativo sia selezionabile solo nell'intervallo specificato (in questo caso [0,1]).

La parte fondamentale del modulo di interazione è però ovviamente l'algoritmo SVQ (sezione 5.2.1), che è stato implementato nella classe *GestureComparer* (appendice B.2). Il suo comportamento è stato suddiviso in tre metodi: il primo è *CompareToGestures*, il quale confronta una posa con tutte i gesti statici standard a disposizione della classe.

```
    public virtual void CompareToGestures(IHandAdapter hand)
    {
        foreach (StaticGesture gesture in m_GestureList)
        {
            m_GestureDistances[gesture] = Compare(hand, gesture);
        }
    }
```

Listing 6.11: Il metodo CompareToGestures di GestureComparer

Il confronto tra due pose viene poi effettuato dal secondo metodo, *Compare*, che restituisce un valore di similarità tra le due.

```
    private float Compare(IHandAdapter hand, StaticGesture gesture)
    {
```

```
        float result = 0;
        foreach (Finger f in hand.Fingers)
        {
            result += GetFingerDistance(f, gesture);
        }
        return Mathf.Sqrt(result);
    }
```

Listing 6.12: Il metodo Compare di GestureComparer

Infine, la terza e ultima parte dell'algoritmo è stata implementata nel metodo *GetBestGesture*: questo restituisce la posa più similare a quella fornita in input al metodo precedente, scartando quelle il cui valore di similarità non supera una certa soglia.

```
    public StaticGesture GetBestGesture()
    {
        float minDistance = float.MaxValue;
        StaticGesture bestGesture = null;

        foreach (StaticGesture gesture in m_GestureDistances.Keys)
        {
            float distance = Mathf.InverseLerp(0f, 2.24f/*~Sqrt(5)*/,
                m_GestureDistances.GetValueOrDefault(gesture));

            if (distance < minDistance)
            {
                minDistance = distance;
                bestGesture = gesture;
            }

            if (distance < 0.1f) { break; }
        }

        if (minDistance > 0.65f) { return null; }

        return bestGesture;
    }
```

Listing 6.13: Il metodo GetBestGesture di GestureComparer

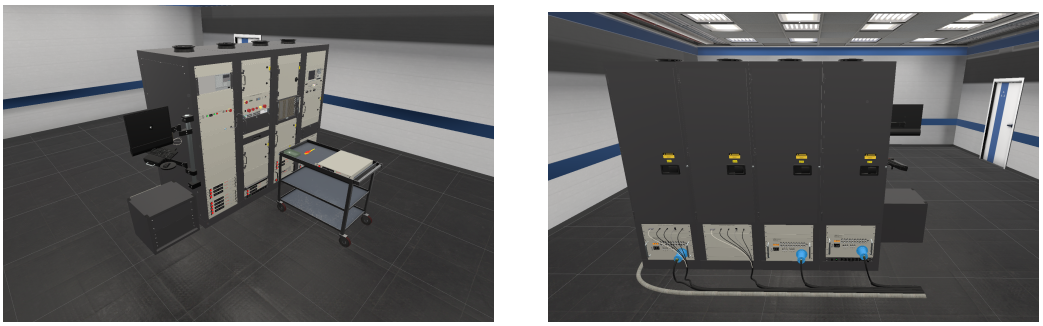
6.4 Scenario dimostrativo

In quest'ultima sezione dell'elaborato verrà descritto il lavoro di integrazione del sistema all'interno di un progetto preesistente, al fine di verificarne le funzionalità e infine validarlo. Il progetto descritto di seguito è stato fornito dall'azienda ospitante, ed è costituito da una semplice scena interattiva VR in Unity.

Va fatto presente che per integrare al meglio l'Input Adapter nella scena, è stato necessario modificarne in parte la logica applicativa. Ciò non costituisce tuttavia un problema rilevante, dato che tutte le funzionalità dell'applicativo sono state preservate.

6.4.1 Logica di gioco

Il progetto in questione è un *Proof of Concept*[1] (POC) sviluppato dall'azienda ospitante: l'applicazione fornisce un ambiente VR in cui viene simulata la manutenzione di un rack di alimentatori (*power supply*).



(a) (b)

Figura 6.1: Immagini dell'ambiente del POC

L'obiettivo dell'utente è scollegare dai cavi il power supply danneggiato, svitare le viti che lo fissano al rack e sostituirlo con uno funzionante. I vari step del procedimento vanno eseguiti in un ordine ben preciso, motivo per cui questi vengono illustrati all'utente mediante dei prompt visuali, come descrizioni testuali delle azioni da compiere, oppure animazioni e highlight che mostrano all'utente su quali componenti agire e come procedere.

La game logic suddivide questo procedimento in tre fasi:

1. Scollegamento del power supply danneggiato
2. Sostituzione del power supply
3. Collegamento del nuovo power supply

Ciascuna di queste fasi è gestita mediante una apposita *Coroutine* di Unity, ovvero un metodo la cui esecuzione viene gestita in maniera asincrona rispetto al game loop principale dell'applicazione.

6.4.2 Logica di interazione

La precedente logica di interazione del POC era stata sviluppata con l'intento di ultimare un prototipo che si prevedeva avrebbe avuto un ciclo di vita molto breve. Di conseguenza erano state utilizzate le API di interazione fisica fornite dal plugin Unity per *SteamVR*, già accennato nella sezione 3.1.2. Gli script risultanti vincolavano però all'utilizzo del suddetto plugin, per cui si è rivelato

[1] simulazione di utilizzo di un prototipo software, hardware o di un'altra soluzione tecnologica, in genere per verificarne le funzionalità previste e/o l'effettiva possibilità di svilupparlo

necessario rimpiazzare le API problematiche con altre compatibili con lo Unity Input System.

Per rimanere in continuità con le scelte architetturali del progetto, è stato fatto uso di API e plugins messi a disposizione direttamente e gratuitamente da Unity Technologies, fornendo quindi anche la massima compatibilità con l'Input System. Per i dispositivi di input XR classici sono state utilizzati gli script che l'API standard di Unity mette a disposizione: in particolare, il visore viene rappresentato nell'ambiente virtuale da uno script *XROrigin*.

Sulla base dello Unity Input System, gli sviluppatori dell'engine hanno sviluppato anche un package per semplificare la definizione delle interazioni nei prodotti XR, lo **Unity XR Interaction Toolkit**. Proprio questo plugin è stato utilizzato per ridefinire la logica di interazione del POC, e sarà quindi utile una piccola digressione sulla sua struttura generale.

Struttura dello Unity XR Interaction System

Lo Unity XR Interaction Toolkit ha alla base una struttura molto semplice. Tutti gli oggetti riconosciuti dal motore di interazione vengono suddivisi in due categorie:

- **Interactables** (oggetti interagibili): sono tutti gli elementi dell'ambiente virtuale che dovranno reagire in seguito a determinate azioni; sono *interactables* ad esempio un cacciavite che può essere afferrato, oppure un pulsante da premere
- **Interactors** (agenti): sono gli enti che agiscono sull'ambiente virtuale, modificando lo stato degli *interactables* con cui è concesso loro interagire; gli *interactors* sono generalmente gli strumenti con cui l'utente interagisce, ma potrebbero anche non essere legati all'input (ad esempio anche un robot simulato virtualmente può essere un *interactor*)

L'interazione tra questi oggetti nella scena viene gestita da un intermediario, l'**Interaction Manager**, responsabile di attuare i cambiamenti di stato tra gli Interactable e gli Interactor registrati presso di lui. Un setup tipico è quello di avere in scena un singolo Interaction Manager che gestisce tutti gli Interactable e tutti gli Interactor presenti, permettendo quindi a tutti di interagire con tutto. Definire più Interaction Manager, ognuno con il suo insieme di Interactable e Interactor, può essere utile per disattivare facilmente interi gruppi di interazioni (disattivando il rispettivo Interaction Manager), oppure per definire un Interaction Manager modificato per specifici tipi di interazioni.

I principali stati di interazione previsti dal plugin sono tre: *hover* ("sfiorare"), quando un Interactable entra nel raggio d'azione di un Interactor; *select* ("selezionare"), quando un Interactor agisce attivamente su un Interactable, ad esempio per afferrarlo; infine, *activate* ("attivare"), effettuabile su un oggetto già selezionato per ottenere un effetto aggiuntivo, ad esempio per accendere una torcia già afferrata. Oltre a questi è possibile definire nuovi stati estendendo le definizioni

di Interactable, Interactor e Interaction Manager fornite dal plugin, ma ciò non risulta necessario nella stragrande maggioranza dei casi ed è quindi sconsigliato.

L'XR Interaction Toolkit fornisce anche altre funzionalità utili: mette a disposizione ad esempio un'astrazione per i dati di input provenienti da controller XR classici, chiamata **XR Controller** appunto, che si appoggia allo Unity Input System e quindi valido da utilizzare nel progetto qui interessato. Sono presenti anche interfacce utili a definire interfacce grafiche: gli sviluppatori possono definire pulsanti, slider, schermate mobili, tutti interagibili tramite gli strumenti dell'Interaction Toolkit.

API del plugin

L'API fornisce quindi due script, *XRInteractable* ed *XRInteractor*, che associati ad un oggetto di scena permettono di identificarlo in una delle due categorie. Le interfacce dei due script permettono inoltre di definire il comportamento degli oggetti in risposta ad una certa interazione e al loro conseguente cambio di stato. Un esempio importante è il metodo *ProcessInteractable* di XRInteractable, nel quale è possibile definire il comportamento di un oggetto quando entra nello stato *select* a causa di un Interactor. Vice versa, la classe XRInteractor contiene vari metodi che specificano come comportarsi quando l'Interactor stesso causa un cambio di stato in un Interactable, come ad esempio il metodo *OnSelectEnter*, che definisce il comportamento dell'Interactor non appena seleziona un oggetto interagibile. Gli Interactor devono inoltre specificare il loro raggio d'azione, cioè la porzione di spazio su cui gli è possibile agire (nei termini del framework, quand'è che effettua *hover* su un Interactable).

Tutte queste interfacce vanno implementate per definire degli Interactable e Interactor che rappresentino al meglio il comportamento atteso dalle interazioni in scena. Lo stesso plugin mette a disposizione delle implementazioni già pronte per i comportamenti più comuni, su tutti **XRGrabInteractable**, che rappresenta un oggetto che si aggancia all'Interactor che lo seleziona (*selezionare* l'oggetto corrisponde ad *afferrarlo*).

Utilizzo dell'XR Interaction Toolkit

Per ridefinire le interazioni del POC utilizzando l'XR Interaction Toolkit, si è prima di tutto associato uno script *XRDirectInteractor* ai modelli delle mani dell'utente. Questi script definiscono un semplice Interactor che seleziona l'Interactable più vicino in risposta all'attivazione di una Input Action specifica, che in questo caso corrispondeva all'azione di *grab* riconosciuta dall'Input Adapter.

Fatto ciò è bastato associare uno script *XRInteractable* di qualche tipo agli oggetti con cui era necessario interagire: in gran parte sei casi è stato utilizzato *XRGrabInteractable*, per gli oggetti che era necessario afferrare normalmente. Questo comportamento è stato associato al cacciavite necessario a svitare le viti

di supporto dei power supply, e alla maniglia del portello posteriore del rack.

Nel caso dei power supply invece era necessario definire un comportamento più complesso: un alimentatore andava afferrato contemporaneamente con entrambe le mani, e la logica di *XRGrabInteractable* non prevede un comportamento di questo tipo. È stato quindi creato un nuovo script, **XRDoubleGrabInteractable**, che soddisfacesse le necessità di cui sopra. Questo script, estensione di *XRGrabInteractable*, ridefinisce la logica con cui l'oggetto entra/esce dallo stato di *select*, e il modo in cui la sua posizione/rotazione viene calcolata mentre viene selezionato:

- L'oggetto deve essere necessariamente selezionato da due Interactor contemporaneamente, altrimenti rimarrà inerte
- Mentre è in stato *select*, la sua posizione viene aggiornata seguendo il punto medio dell'asse tra i due Interactor che l'hanno selezionato
- La sua rotazione invece viene aggiornata seguendo la rotazione del suddetto asse

Per assicurarsi che le mani dell'utente siano gli unici Interactor a poter interagire con questi oggetti di scena, è sufficiente che i modelli delle mani siano gli unici Interactor registrati presso l'Interaction Manager che si occupa di tutti gli oggetti interessati.

Conclusioni

L'obiettivo del progetto esposto in questo elaborato era quello di sviluppare un sistema che potesse astrarre qualunque sottosistema di hand tracking dall'utilizzo che se ne faceva nelle applicazioni prodotte con Unity, con lo scopo in particolare di interfacciarsi in maniera efficace con il servizio Ultraleap per il tracciamento delle mani nude. Inoltre il sistema doveva fornire delle funzionalità di gesture recognition, per gestire le interazioni tramite le suddette tecnologie di hand tracking in maniera unificata. I requisiti fondamentali richiesti al sistema erano la sua estensibilità e la semplicità di utilizzo per gli sviluppatori che ne avrebbero usufruito, sia tramite interfacce intuitive che possibilmente con tool appositi.

Per poter valutare correttamente il prodotto finale, quest'ultimo è stato inserito in un progetto VR in Unity già sviluppato, sostituendolo al precedente sistema di input/interazioni e integrando così il tracciamento Ultraleap nell'applicazione. Questa fase del lavoro ha messo in evidenza molti problemi e criticità del sistema tramite il suo utilizzo concreto, permettendo quindi di correggerne molti aumentando in definitiva la qualità del prodotto finale.

Il risultato finale è nel complesso corretto e non mostra errori sostanziali, tuttavia alcuni degli obiettivi proposti all'inizio del progetto non sono stati portati a termine. In particolare non è stato fornire all'Interaction Module funzionalità dedicate al riconoscimento di gesti dinamici (e.g. rotazione del polso, apertura della mano), il che avrebbe permesso di gestire interazioni ancora più complesse all'interno dello stesso Input Adapter, mentre attualmente interazioni di questo tipo vengono gestite in maniera *ad hoc* con l'XR Interaction Toolkit e/o altra logica di gioco dedicata. Tuttavia questa problematica è così complessa che necessiterebbe di un intero progetto dedicato, e in definitiva esulava dallo scopo principale del lavoro qui discusso.

Inoltre vi sono diversi miglioramenti che sarebbe possibile apportare al sistema nel suo stato attuale, soprattutto all'Interaction Module, che dei due moduli sviluppati risulta in conclusione essere quello su cui vi sono più possibilità di evoluzione. Nello specifico, oltre al problema dei gesti dinamici sopra discusso, sarebbe interessante applicare al riconoscimento delle pose statiche delle tecniche di machine learning, che ben si prestano al riconoscimento di pattern all'interno di dati strutturati complessi. Si potrebbero inoltre definire dei tool per gli sviluppatori, integrati nello Unity Editor e utili a definire le pose da riconoscere in maniera più intuitiva, piuttosto che tramite il loro modello dei dati.

In conclusione, gli obiettivi posti all'inizio del progetto possono considerarsi raggiunti a pieno titolo: il sistema sviluppato permette un'ottima integrazione del tracciamento Ultraleap, è una valida interfaccia di astrazione per altri sistemi di hand tracking, e il suo funzionamento in un'ottica di integrazione con prodotti software preesistenti è stata correttamente verificata. Le possibili migliorie su cui lavorare al di fuori dei requisiti di base sono molteplici, ma grazie alla sua struttura modulare e all'utilizzo delle soluzioni tipiche dell'ingegneria del software, esso si presta perfettamente a successivi miglioramenti e perfezionamenti, e anzi future modifiche al sistema sono da incentivare.

Appendice

A Codice dell'Input Module

A.1 LeapHandAdapter

```
/// <summary>
/// Implementation of <see cref="IHandAdapter"/> that adapts data
/// sent by the Ultraleap hand tracking service through a
/// <see cref="LeapServiceProvider"/> object.
/// </summary>
public class LeapHandAdapter : IHandAdapter
{
    private Leap.Hand handAdaptee;

    public override ChiralityType Chirality
    {
        get
        {
            if (handAdaptee == null)
                return ChiralityType.Invalid;

            if (handAdaptee.IsLeft)
                return ChiralityType.Left;
            else
                return ChiralityType.Right;
        }
    }

    public override uint TrackingState
    {
        get
        {
            if (handAdaptee == null)
                return 0;

            return 255;
        }
    }

    public override Vector3 WristPosition
    {
        get
```

```
        {
            if (handAdaptee == null)
                return Vector3.zero;

            return handAdaptee.WristPosition;
        }
    }

    public override Quaternion HandRotation
    {
        get
        {
            if (handAdaptee == null)
                return Quaternion.identity;

            return handAdaptee.Rotation;
        }
    }

    protected Finger[] m_Fingers = new Finger[5];
    public override IReadOnlyList<Finger> Fingers => m_Fingers;

    public LeapHandAdapter(Leap.Hand leapHand)
    {
        if (leapHand == null)
        {
            throw new ArgumentException();
        }

        this.handAdaptee = leapHand;
        SetFingers(leapHand);
    }

    private void SetFingers(Leap.Hand leapHand)
    {
        foreach (Leap.Finger leapFinger in leapHand.Fingers)
        {
            Finger.FingerType type = Finger.FingerType.Invalid;
            switch (leapFinger.Type)
            {
                case Leap.Finger.FingerType.TYPE_THUMB:
                    type = Finger.FingerType.Thumb;
                    break;
                case Leap.Finger.FingerType.TYPE_INDEX:
                    type = Finger.FingerType.Index;
                    break;
                case Leap.Finger.FingerType.TYPE_MIDDLE:
                    type = Finger.FingerType.Middle;
                    break;
                case Leap.Finger.FingerType.TYPE_RING:
                    type = Finger.FingerType.Ring;
                    break;
                case Leap.Finger.FingerType.TYPE_PINKY:
                    type = Finger.FingerType.Pinky;
```

```
92                      break;
93              }
94              if (type != Finger.FingerType.Invalid)
95              {
96                  m_Fingers[(int)type] = ToUnityFinger(leapFinger, type);
97              }
98          }
99      }
100
101     private Finger ToUnityFinger(Leap.Finger leapFinger, Finger.
            FingerType fingerType)
102     {
103         return new Finger
104         {
105             type = fingerType,
106             metacarpal = ToUnityBone(leapFinger.Bone(Leap.Bone.BoneType.
                TYPE_METACARPAL)),
107             proximal = ToUnityBone(leapFinger.Bone(Leap.Bone.BoneType.
                TYPE_PROXIMAL)),
108             intermediate = ToUnityBone(leapFinger.Bone(Leap.Bone.
                BoneType.TYPE_INTERMEDIATE)),
109             distal = ToUnityBone(leapFinger.Bone(Leap.Bone.BoneType.
                TYPE_DISTAL)),
110             extension = 1.0f - handAdaptee.GetFingerStrength((int)
                fingerType)
111         };
112     }
113
114     private Bone ToUnityBone(Leap.Bone leapBone)
115     {
116         return new Bone
117         {
118             position = leapBone.Center,
119             rotation = leapBone.Rotation
120         };
121     }
122 }
```

A.2 TrackedHandDeviceState

```
/// <summary>
/// Struct that defines the layout of the memory state block
/// for a <see cref="TrackedHandDevice"/>. For more informations
/// about state structs, see
/// <see href="https://docs.unity3d.com/Packages/com.unity.inputsystem@1.4/manual/Devices.html#device-state:~:text=device%20interprets%20it.-,Device%20state,-Like%20any%20other">
/// the Unity Input System docs</see>.
/// </summary>
public struct TrackedHandDeviceState : IInputStateTypeInfo
{
  public FourCC format => new FourCC('T', 'R', 'H', 'A');

  [Preserve, InputControl(name = "trackingState", layout = "Integer", displayName = "Tracking State")]
  public uint trackingState;

  [Preserve, InputControl(name = "wristPosition", layout = "Vector3", displayName = "Wrist Position", noisy = true, dontReset = true)]
  public Vector3 wristPosition;

  [Preserve, InputControl(name = "handRotation", layout = "Quaternion", displayName = "Hand Rotation", noisy = true, dontReset = true)]
  public Quaternion handRotation;

  [Preserve, InputControl(name = "thumb", layout = "Finger", displayName = "Thumb", noisy = true, dontReset = true)]
  public Finger thumb;

  [Preserve, InputControl(name = "index", layout = "Finger", displayName = "Index", noisy = true, dontReset = true)]
  public Finger index;

  [Preserve, InputControl(name = "middle", layout = "Finger", displayName = "Middle", noisy = true, dontReset = true)]
  public Finger middle;

  [Preserve, InputControl(name = "ring", layout = "Finger", displayName = "Ring", noisy = true, dontReset = true)]
  public Finger ring;

  [Preserve, InputControl(name = "pinky", layout = "Finger", displayName = "Pinky", noisy = true, dontReset = true)]
  public Finger pinky;

  [Preserve, InputControl(name = "poses", layout = "Integer", dontReset = true)]
  public uint poses;
}
```

A.3 TrackedHandDevice

```
/// <summary>
```

```
/// Class that defines the layout of an abstract Input Device,
/// whose state is updated by passing in <see cref="IHandAdapter"/>
    objects.
/// This layout may be extended when the hand tracking system to be
    adapted
/// has additional input types other than classic hand tracking data:
/// for more information on extending layouts, read
/// <see href="https://docs.unity3d.com/Packages/com.unity.inputsystem@1
    .4/manual/Devices.html#creating-custom-devices:~:text=native%20
    representation.-,Creating%20custom%20Devices,-Note%3A%20This%20
    example">
/// the Unity Input System docs</see>.
/// </summary>
#if UNITY_EDITOR
[InitializeOnLoad]
#endif
[Preserve, InputControlLayout(displayName = "Tracked Hand",
    isGenericTypeOfDevice = true,
  commonUsages = new[] { "LeftHand", "RightHand" },
  stateType = typeof(TrackedHandDeviceState))]
public abstract class TrackedHandDevice : InputDevice,
    IInputUpdateCallbackReceiver
{
  #region Child Control References
  public IntegerControl TrackingState { get; protected set; }
  public Vector3Control WristPosition { get; protected set; }
  public QuaternionControl HandRotation { get; protected set; }

  public IReadOnlyList<FingerControl> Fingers => m_FingerList;
  protected List<FingerControl> m_FingerList;

  protected IntegerControl Poses { get; set; }

  public IReadOnlyDictionary<StaticGesture, ButtonControl> Gestures =>
      m_Gestures;
  protected Dictionary<StaticGesture, ButtonControl> m_Gestures;

  protected static Dictionary<StaticGesture, int>
      m_GestureControlIndexes = new Dictionary<StaticGesture, int>();

  #endregion

  #region Device Instances
  public TrackedHandDevice Current { get; protected set; }
  public static IReadOnlyList<TrackedHandDevice> All => s_AllDevices;
  private static List<TrackedHandDevice> s_AllDevices = new List<
      TrackedHandDevice>();
  #endregion

  #region Initialization Methods
#if UNITY_EDITOR
  static TrackedHandDevice()
  {
    Initialize();
```

```
46    }
47  #endif
48
49    /// <summary>
50    /// Static initializer that registers the layout
51    /// both in Edit Mode and Play Mode.
52    /// </summary>
53    /// <seealso cref="BuildLayoutOverrideJson"/>
54    [Preserve, RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.
          BeforeSceneLoad)]
55    private static void Initialize()
56    {
57      InputSystem.RegisterLayout<TrackedHandDevice>(
58        name: "Tracked Hand",
59        matches: new InputDeviceMatcher()
60          .WithInterface("Tracked Hand"));
61
62      InputSystem.RegisterLayoutOverride(BuildLayoutOverrideJson(),
63        name: "Tracked Hand Gestures Extension");
64    }
65
66    /// <inheritdoc/>
67    protected override void FinishSetup()
68    {
69      base.FinishSetup();
70
71      TrackingState = GetChildControl<IntegerControl>("trackingState");
72      WristPosition = GetChildControl<Vector3Control>("wristPosition");
73      HandRotation = GetChildControl<QuaternionControl>("handRotation");
74
75      m_FingerList = new List<FingerControl>
76      {
77        GetChildControl<FingerControl>("thumb"),
78        GetChildControl<FingerControl>("index"),
79        GetChildControl<FingerControl>("middle"),
80        GetChildControl<FingerControl>("ring"),
81        GetChildControl<FingerControl>("pinky")
82      };
83
84      Poses = GetChildControl<IntegerControl>("poses");
85
86      m_Gestures = new Dictionary<StaticGesture, ButtonControl>();
87
88      foreach (StaticGesture gesture in GestureLoader.GestureList)
89      {
90        m_Gestures.Add(gesture, GetChildControl<ButtonControl>(gesture.
              name));
91      }
92    }
93
94    /// <summary>
95    /// This method builds a JSON string, representing a runtime layout
96    /// extension that adds to this layout a <see cref="ButtonControl"/>
97    /// for each <see cref="StaticGesture"/> found by the
```

```
98  /// <see cref="GestureLoader"/>.
99  /// </summary>
100 /// <returns>The JSON string representing the override to be applied
        to the layout.</returns>
101 private static string BuildLayoutOverrideJson()
102 {
103   //Calcola l'offset a cui le gesture sono posizionate nella State
        Struct
104   int byteOffset = Marshal.OffsetOf<TrackedHandDeviceState>("poses").
        ToInt32();
105   int bitOffset = 0;
106
107   string jsonControls = @"""controls"" : [";
108   foreach (StaticGesture gesture in GestureLoader.GestureList)
109   {
110     if (bitOffset > 31)
111     {
112       Debug.LogWarning("WARNING: More than 32 gestures found by the
            GestureLoader, " +
113         "exceeding gestures will be ignored.");
114       break;
115     }
116
117     jsonControls += @"{ ""name"" : """ + gesture.name + @""",
118               ""layout"" : ""Button"",
119               ""offset"" : " + byteOffset + @",
120               ""bit"" : """ + bitOffset + @"""
121             },";
122
123     m_GestureControlIndexes.Add(gesture, bitOffset);
124
125     bitOffset++;
126   }
127
128   jsonControls = jsonControls.TrimEnd(',');
129
130   return @"
131     {
132       ""name"" : ""Tracked Hand Gestures Extension"",
133       ""extend"" : ""Tracked Hand"",
134     " + jsonControls + @"
135       ]
136     }";
137 }
138 #endregion
139
140 #region Input Device methods
141 /// <inheritdoc/>
142 public override void MakeCurrent()
143 {
144   base.MakeCurrent();
145   Current = this;
146 }
147
```

```
148   /// <inheritdoc/>
149   protected override void OnAdded()
150   {
151     base.OnAdded();
152     s_AllDevices.Add(this);
153   }
154
155   /// <inheritdoc/>
156   protected override void OnRemoved()
157   {
158     base.OnRemoved();
159     s_AllDevices.Remove(this);
160     if (Current == this)
161     {
162       Current = null;
163     }
164   }
165   #endregion
166
167   #region State Update Methods
168
169   /// <summary>
170   /// Called automatically during <see cref="InputSystem.onBeforeUpdate
          "/>.
171   /// May be used to provide input events to update the current device
          state.
172   /// </summary>
173   public void OnUpdate()
174   {
175     IHandAdapter hand = AdaptCurrentTrackingData();
176
177     if (hand == null) NoInputReceived();
178     else UpdateState(hand);
179   }
180
181   /// <summary>
182   /// Encapsulates hand tracking data for this input update loop
183   /// </summary>
184   /// <returns>The encapsulated hand tracking data,
185   /// or null if none is found</returns>
186   public abstract IHandAdapter AdaptCurrentTrackingData();
187
188   private double secWithoutGesture = 0d;
189   private double secBetweenGestures = 0d;
190   private StaticGesture lastFramePose = null;
191   protected GestureComparer gestureComparer = new GestureComparer(
          GestureLoader.GestureList);
192   /// <summary>
193   /// Updates the state of the device bound to this layout based
194   /// on the hand tracking data given by the <see cref="IHandAdapter"/>.
195   /// </summary>
196   /// <param name="hand">Adapter interface that provides all data needed
          to update the device state.</param>
197   public virtual void UpdateState(IHandAdapter hand)
```

```
198   {
199     InputSystem.QueueDeltaStateEvent<uint>(TrackingState, hand.
          TrackingState);
200
201     //Update Wrist Position
202     InputSystem.QueueDeltaStateEvent<Vector3>(
203       WristPosition, hand.WristPosition);
204
205     //Update Hand Rotation
206     InputSystem.QueueDeltaStateEvent<Quaternion>(
207       HandRotation, hand.HandRotation);
208
209     //Update finger positions/rotations
210     foreach (Finger finger in hand.Fingers)
211     {
212       InputSystem.QueueDeltaStateEvent<FingerControlState>(
213         GetFingerControl(finger.type), ToFingerControlState(finger));
214     }
215
216     //Update gesture distances for current pose
217     gestureComparer.CompareToGestures(hand);
218
219     //Update current gesture
220     StaticGesture newPose = gestureComparer.GetBestGesture();
221     StaticGesture currentPose = GestureLoader.FindGesture(x =>
          GetStaticGestureControlBitmask(x) == (uint)Poses.ReadValue());
222     if (newPose != null && (currentPose == null || !currentPose.
          dropOnNoGesture))
223     {
224       secWithoutGesture = 0d;
225       InputSystem.QueueDeltaStateEvent<uint>(
226         Poses, GetStaticGestureControlBitmask(newPose));
227     }
228     else if (newPose == null && currentPose != null)
229     {
230       secWithoutGesture += Time.deltaTime;
231       InputSystem.QueueDeltaStateEvent<uint>(Poses,
            GetStaticGestureControlBitmask(currentPose));
232
233       if (secWithoutGesture > 0.15)
234       {
235         secWithoutGesture = 0d;
236         InputSystem.QueueDeltaStateEvent<uint>(Poses, 0);
237       }
238     }
239     else if(currentPose != null && currentPose.dropOnNoGesture)
240     {
241       secWithoutGesture = 0d;
242       InputSystem.QueueDeltaStateEvent<uint>(
243         Poses, GetStaticGestureControlBitmask(currentPose));
244     }
245
246     lastFramePose = newPose;
247   }
```

```
248
249   public virtual void NoInputReceived()
250   {
251     InputSystem.QueueDeltaStateEvent<uint>(TrackingState, 0);
252   }
253
254   /// <summary>
255   /// Utility method that converts a <see cref="Finger"/> struct
256   /// to an equivalent <see cref="FingerControlState"/> struct
257   /// containing the same positional data.
258   /// </summary>
259   /// <param name="finger">The struct to convert.</param>
260   /// <returns>A state struct containing the same input data.</returns>
261   protected FingerControlState ToFingerControlState(Finger finger)
262   {
263     return new FingerControlState()
264     {
265       type = (uint)finger.type,
266       boneMetacarpal = finger.metacarpal,
267       boneProximal = finger.proximal,
268       boneIntermediate = finger.intermediate,
269       boneDistal = finger.distal
270     };
271   }
272
273   /// <summary>
274   /// Utility method used to quickly access the references to this
        layout's
275   /// <see cref="FingerControl"/> children.
276   /// </summary>
277   /// <param name="type">The desired finger type.</param>
278   /// <returns>The child <see cref="FingerControl"/> corresponding to
        the given finger type.</returns>
279   protected FingerControl GetFingerControl(Finger.FingerType type)
280   {
281     if (type == Finger.FingerType.Invalid)
282     {
283       throw new ArgumentException();
284     }
285
286     return Fingers[(int)type];
287   }
288
289   /// <summary>
290   /// Utility method used internally to set
291   /// the value of the <see cref="StaticGesture"/>-related Input
        Controls.
292   /// </summary>
293   private uint GetStaticGestureControlBitmask(StaticGesture gesture)
294   {
295     return (uint)(1 << m_GestureControlIndexes[gesture]);
296   }
297   #endregion
298 }
```

A.4 UltraleapTrackedHandDevice

```
public struct UltraleapTrackedHandDeviceState : IInputStateTypeInfo
{
  public FourCC format => new FourCC('U', 'L', 'T', 'H');

  [Preserve, InputControl(name = "trackingState", layout = "Integer",
     displayName = "Tracking State")]
  public uint trackingState;

  [Preserve, InputControl(name = "wristPosition", layout = "Vector3",
     displayName = "Wrist Position", noisy = true, dontReset = true)]
  public Vector3 wristPosition;

  [Preserve, InputControl(name = "handRotation", layout = "Quaternion",
     displayName = "Hand Rotation", noisy = true, dontReset = true)]
  public Quaternion handRotation;

  [Preserve, InputControl(name = "thumb", layout = "Finger", displayName
      = "Thumb", noisy = true, dontReset = true)]
  public Finger thumb;

  [Preserve, InputControl(name = "index", layout = "Finger", displayName
      = "Index", noisy = true, dontReset = true)]
  public Finger index;

  [Preserve, InputControl(name = "middle", layout = "Finger",
     displayName = "Middle", noisy = true, dontReset = true)]
  public Finger middle;

  [Preserve, InputControl(name = "ring", layout = "Finger", displayName
     = "Ring", noisy = true, dontReset = true)]
  public Finger ring;

  [Preserve, InputControl(name = "pinky", layout = "Finger", displayName
      = "Pinky", noisy = true, dontReset = true)]
  public Finger pinky;

  [Preserve, InputControl(name = "poses", layout = "Integer", dontReset
     = true)]
  public uint poses;

  [Preserve, InputControl(name = "secWithoutGesture", dontReset = true)]
  public double secWithoutGesture;
}

#if UNITY_EDITOR
[InitializeOnLoad]
#endif
[InputControlLayout(displayName = "Ultraleap Hand",
  commonUsages = new[] { "LeftHand", "RightHand" },
  stateType = typeof(UltraleapTrackedHandDeviceState))]
public class UltraleapTrackedHandDevice : TrackedHandDevice
{
  public static new UltraleapTrackedHandDevice Current { get; private
```

```
      set; }
  public static new IReadOnlyList<UltraleapTrackedHandDevice> All =>
      s_AllDevices;
  private static List<UltraleapTrackedHandDevice> s_AllDevices =
    new List<UltraleapTrackedHandDevice>();

  protected LeapServiceProvider serviceProvider;

#if UNITY_EDITOR
  static UltraleapTrackedHandDevice()
  {
    Initialize();
  }
#endif

  [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.
      BeforeSceneLoad)]
  private static void Initialize()
  {
    InputSystem.RegisterLayout<UltraleapTrackedHandDevice>(
      name: "Ultraleap Tracked Hand",
      matches: new InputDeviceMatcher()
        .WithInterface("Tracked Hand")
        .WithDeviceClass("Ultraleap"));
  }

  protected override void FinishSetup()
  {
    base.FinishSetup();

    serviceProvider = UnityEngine.Object.FindObjectOfType<Leap.Unity.
        LeapServiceProvider>();
  }

  public override void MakeCurrent()
  {
    base.MakeCurrent();
    Current = this;
  }

  protected override void OnAdded()
  {
    base.OnAdded();
    s_AllDevices.Add(this);
  }

  protected override void OnRemoved()
  {
    base.OnRemoved();
    s_AllDevices.Remove(this);
    if (Current == this)
    {
      Current = null;
    }
```

```
95    }
96
97    /// <inheritdoc/>
98    public override IHandAdapter AdaptCurrentTrackingData()
99    {
100     serviceProvider ??= UnityEngine.Object.FindObjectOfType<Leap.Unity.
          LeapServiceProvider>();
101
102     Leap.Hand leapHand = serviceProvider?.CurrentFrame?.GetHand(
103       (usages.Contains(CommonUsages.LeftHand)) ? Chirality.Left :
            Chirality.Right);
104
105     if (leapHand == null)
106     {
107       return null;
108     }
109
110     return new LeapHandAdapter(leapHand);
111   }
112 }
```

B Codice dell'Interaction Module

B.1 StaticGesture

```
/// <summary>
/// A <see cref="ScriptableObject"/> representing a static hand pose
/// to be recognised by the Tracking Adapter's Interaction System
/// </summary>
[CreateAssetMenu(fileName = "Gesture", menuName = "ScriptableObjects/
    Interaction System/ScriptableGesture", order = 1)]
public class StaticGesture : ScriptableObject
{
    public int Id => name.GetHashCode();

    [Header("Finger Extensions")]
    [Header("Thumb")]
    [Range(0,1)]
    public float ThumbExtensionAvg = 0;

    [Header("Index")]
    [Range(0, 1)]
    public float IndexExtensionAvg = 0;

    [Header("Middle")]
    [Range(0, 1)]
    public float MiddleExtensionAvg = 0;

    [Header("Ring")]
    [Range(0, 1)]
    public float RingExtensionAvg = 0;

    [Header("Pinky")]
    [Range(0, 1)]
    public float PinkyExtensionAvg = 0;

    [Header("Other Params")]
    [Tooltip("Specify when this gesture should end:\n\n" +
        "- False = End gesture as usual.\n" +
        "- True = End gesture only when switching to a \"no gesture\"
            state.")]
    public bool dropOnNoGesture = false;

    public float GetFingerExtension(Finger.FingerType fingerType)
    {
        switch(fingerType)
        {
            case Finger.FingerType.Thumb:
                return ThumbExtensionAvg;
            case Finger.FingerType.Index:
                return IndexExtensionAvg;
            case Finger.FingerType.Middle:
                return MiddleExtensionAvg;
            case Finger.FingerType.Ring:
                return RingExtensionAvg;
```

```
            case Finger.FingerType.Pinky:
                return PinkyExtensionAvg;
            default:
                return 0f;
        }
    }
}
```

B.2 GestureComparer

```
/// <summary>
/// Given a list of StaticGesture's, it compares them to any
/// hand pose, represented by an IHandAdapter object
/// </summary>
public class GestureComparer
{
    private IReadOnlyList<StaticGesture> m_GestureList;

    private Dictionary<StaticGesture, float> m_GestureDistances = new
        Dictionary<StaticGesture, float>();
    public IReadOnlyDictionary<StaticGesture, float> GestureDistances =>
        m_GestureDistances;

    public GestureComparer(IReadOnlyList<StaticGesture> gestures)
    {
        m_GestureList = gestures;
        foreach (StaticGesture g in gestures)
        {
            m_GestureDistances.Add(g, float.MaxValue);
        }
    }

    /// <summary>
    /// Updates the similarity values of each gesture based
    /// on the input tracking data
    /// </summary>
    /// <param name="hand"></param>
    public virtual void CompareToGestures(IHandAdapter hand)
    {
        foreach (StaticGesture gesture in m_GestureList)
        {
            m_GestureDistances[gesture] = Compare(hand, gesture);
        }
    }

    /// <summary>
    /// Compares a hand pose to a static gesture
    /// </summary>
    /// <param name="hand">The hand pose to evaluate</param>
    /// <param name="gesture">The static gesture to compare to</param>
    /// <returns>A float representing the distance of the hand pose from
        the given gesture</returns>
    private float Compare(IHandAdapter hand, StaticGesture gesture)
    {
```

```
42          float result = 0;
43          foreach (Finger f in hand.Fingers)
44          {
45              result += GetFingerDistance(f, gesture);
46          }
47          return Mathf.Sqrt(result);
48      }
49
50      /// <summary>
51      /// Compares a specific finger pose to a static gesture
52      /// </summary>
53      /// <param name="finger">The finger pose to evaluate</param>
54      /// <param name="gesture">The gesture to compare to</param>
55      /// <returns>
56      /// A float representing the distance of the
57      /// finger pose from the given gesture's corresponding finger
58      /// </returns>
59      private float GetFingerDistance(Finger finger, StaticGesture gesture
            )
60      {
61          return Mathf.Pow(finger.extension - gesture.GetFingerExtension(
                finger.type), 2);
62      }
63
64      /// <summary>
65      /// Evaluates the gesture with the current highest similarity
66      /// (lowest distance) among the ones in the dictionary
67      /// </summary>
68      /// <returns>
69      /// The gesture with the highest similarity, or
70      /// null if the highest similarity is below a certain threshold
71      /// </returns>
72      public StaticGesture GetBestGesture()
73      {
74          float minDistance = float.MaxValue;
75          StaticGesture bestGesture = null;
76
77          foreach (StaticGesture gesture in m_GestureDistances.Keys)
78          {
79              float distance = Mathf.InverseLerp(0f, 2.24f/*~Sqrt(5)*/,
                    m_GestureDistances.GetValueOrDefault(gesture));
80
81              if (distance < minDistance)
82              {
83                  minDistance = distance;
84                  bestGesture = gesture;
85              }
86
87              if (distance < 0.1f) { break; }
88          }
89
90          if (minDistance > 0.65f) { return null; }
91
92          return bestGesture;
```

```
93          }
94  }
```

Bibliografia

[1] Thomas DeFanti e Daniel J. Sandin. *Sayre Glove Final Project Report, US NEA R60-34-163 Final Project Report*. Rapp. tecn. Nov. 1977.

[2] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1ª ed. Addison-Wesley Professional, 1994. ISBN: 0201633612. URL: http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1.

[3] Joseph J. LaViola. *A Survey of Hand Posture and Gesture Recognition Techniques and Technology*. Rapp. tecn. USA, 1999.

[4] Dharani Mazumdar, Anjan Kumar Talukdar e Kandarpa Kumar Sarma. «Gloved and free hand tracking based hand gesture recognition». In: (2013), pp. 197–202. DOI: 10.1109/ICETACS.2013.6691422.

[5] Paul Milgram et al. «Augmented reality: A class of displays on the reality-virtuality continuum». In: *Telemanipulator and Telepresence Technologies* 2351 (gen. 1994). DOI: 10.1117/12.197321.

[6] Leap Motion. *Ultraleap Unity Plugin documentation*. https://docs.ultraleap.com/unity-api/?_gl=1*1isas66*_ga*MTkwMjk4MDgxLjE2ODY5MTMzNDA.*_ga_5G8B19JLWG*MTY4NjkxMzM0MC4xLjEuMTY4NjkxMzM1OS40MS4wLjA.. Accessed: 2023-06-16.

[7] Munir Oudah, Ali Al-Naji e Javaan Chahl. «Hand Gesture Recognition Based on Computer Vision: A Review of Techniques». In: *Journal of Imaging* 6.8 (2020). ISSN: 2313-433X. DOI: 10.3390/jimaging6080073. URL: https://www.mdpi.com/2313-433X/6/8/73.

[8] Jesus Suarez e Robin R. Murphy. «Hand gesture recognition with depth images: A review». In: *2012 IEEE RO-MAN: The 21st IEEE International Symposium on Robot and Human Interactive Communication*. 2012, pp. 411–417. DOI: 10.1109/ROMAN.2012.6343787.

[9] Unity Technologies. *Unity Input System documentation*. https://docs.unity3d.com/Packages/com.unity.inputsystem@1.6/manual/index.html. Accessed: 2023-06-16.

[10] Unity Technologies. *Unity XR Interaction Toolkit documentation*. https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.3/manual/index.html. Accessed: 2023-06-16.

[11] Varjo. *Varjo Unity XR SDK documentation*. https://developer.varjo.com/docs/unity-xr-sdk/unity-xr-sdk. Accessed: 2023-06-16.