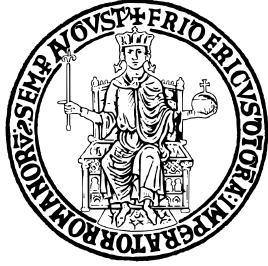


# UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II



# SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

# DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

# CORSO DI LAUREA TRIENNALE IN INFORMATICA

# DATA ACQUISITION AND CONTROL OF ROBOTIC ARM USING A VIRTUAL REALITY INTERFACES

## **Relatore**

Prof. Riccardo CACCIAVALE

## Candidato

Florindo ZECCONI

N86004544

Anno Accademico 2024–2025



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

DATA ACQUISITION AND CONTROL OF  
ROBOTIC ARM USING A VIRTUAL REALITY  
INTERFACES

**Relatore**

Prof. Riccardo CACCAVALE

**Candidato**

Florindo ZECCONI

N86004544

Anno Accademico 2024–2025



# **Abstract**

In this thesis work, the goal is to develop a simulation environment to enable the acquisition and control of a robotic arm through virtual reality interfaces. The document describes the technologies used in the project, including Isaac Sim and ROS2. Isaac Sim is essential for robotic simulation: it is an advanced simulator developed by NVIDIA, which allows real-time reproduction of the robot's behavior in a realistic and interactive virtual environment. ROS2 (Robot Operating System 2) is used to simplify communication between the various components of the system. In particular, it manages message exchange between the VR headset sensors and the robot, leveraging a distributed and modular structure based on the publisher/subscriber model. In addition, the OpenVR library is integrated to collect data from the VR headset sensors (such as head and hand position and orientation), and to feed them into the simulation pipeline. These data are then processed, converted, and stored in .npz and .csv formats, to be used for training machine learning algorithms or for other purposes. Finally, the document also describes the process of computing inverse kinematics, used to determine the joint positions required for the robot to reach a desired point in space.



# Sommario

In questo lavoro di tesi si vuole sviluppare un ambiente di simulazione per consentire l'acquisizione e il controllo di un braccio robotico tramite interfacce di realtà virtuale. Il documento descrive le tecnologie utilizzate nel progetto, tra cui *Isaac Sim* e *ROS2*. Isaac Sim è essenziale per la simulazione robotica: si tratta di un simulatore avanzato, sviluppato da NVIDIA, che consente di riprodurre in tempo reale il comportamento del robot in un ambiente virtuale realistico e interattivo. ROS2 (Robot Operating System 2) è utilizzato per semplificare la comunicazione tra i vari componenti del sistema. In particolare, gestisce lo scambio di messaggi tra i sensori del visore VR e il robot, sfruttando una struttura distribuita e modulare basata sul modello publisher/subscriber. Inoltre, viene integrata la libreria *OpenVR*, che consente di raccogliere i dati provenienti dai sensori del visore VR (come posizione e orientamento della testa e delle mani), per poi inviarli nella pipeline di simulazione. Tali dati vengono successivamente elaborati, convertiti e archiviati nei formati *.npz* e *.csv*, al fine di essere utilizzati per *l'addestramento* di algoritmi di machine learning o per altri scopi. Viene descritto anche il processo di calcolo della cinematica inversa, utilizzato per determinare la posizione dei giunti del robot necessari a raggiungere un punto desiderato nello spazio. Infine, viene spiegato com'è stato validato il sistema, quindi quali test sono stati effettuati e cosa è stato verificato.m'è stato validato il sistema , quindi quali test sono stati effettuati e cosa è stato testato



# Contents

<b>1</b>	<b>Tecnologie utilizzate</b>	<b>5</b>
1.1	Isaac Sim . . . . .	5
1.1.1	Isaac SDK . . . . .	5
1.1.2	Isaac Simulator . . . . .	6
1.1.3	Ragioni della scelta di Isaac . . . . .	6
1.1.4	Comunicazione tra Isaac Sim e ROS 2 . . . . .	6
1.1.5	Dominio ROS 2 e middleware DDS . . . . .	7
1.2	ROS 2 . . . . .	8
1.2.1	Visione d'insieme e architettura . . . . .	8
1.2.2	Librerie client e livello rmw: chiarimento terminologico . . . . .	9
1.2.3	Perché ROS 2 in questo lavoro . . . . .	11
1.2.4	Differenze principali tra ROS 1 e ROS 2 . . . . .	11
1.2.5	Cyclone DDS: scoperta e scambio dati . . . . .	12
1.2.6	Messaggi in ROS 2 e pipeline .msg → .idl . . . . .	12
1.3	ALVR . . . . .	14
1.3.1	Panoramica dell'architettura client-server . . . . .	14
1.3.2	Virtualizzazione del visore in SteamVR . . . . .	15
1.3.3	Flusso di funzionamento end-to-end . . . . .	15
1.3.4	Motivazioni della scelta di ALVR . . . . .	15
1.4	OpenVR e SteamVR . . . . .	17
1.4.1	Ruolo di OpenVR . . . . .	17
1.4.2	Ruolo del runtime SteamVR . . . . .	17
1.4.3	Gestione del tracking e degli input . . . . .	18
1.4.4	Ragioni della scelta di SteamVR e OpenVR . . . . .	18
<b>2</b>	<b>Implementazione</b>	<b>21</b>
2.1	Architettura . . . . .	21
2.2	Utilizzo della libreria OpenVR e SteamVR . . . . .	24
2.2.1	Codice d'esempio: Inizializzazione OpenVR e selezione del controller	25
2.2.2	Codice d'esempio: Raccolta dati . . . . .	27
2.3	Utilizzo di ALVR . . . . .	29
2.4	Calcolo del orientamento della mano . . . . .	30
2.4.1	Struttura della matrice . . . . .	30

---

2.4.2	Significato geometrico . . . . .	30
2.4.3	Esempio numerico . . . . .	31
2.4.4	Trasformazione delle rotazioni da VR a Robot . . . . .	32
2.5	Cinematica inversa per calcolo dei giunti . . . . .	33
2.5.1	Cinematica diretta (il problema facile) . . . . .	34
2.5.2	Cinematica inversa (il problema difficile) . . . . .	34
2.5.3	Algoritmo numerico iterativo per l'IK . . . . .	35
2.6	Il formato URDF . . . . .	36
2.6.1	Definizione e contesto . . . . .	36
2.6.2	Contenuti di un file URDF . . . . .	36
2.6.3	Funzione pratica dell'URDF . . . . .	37
<b>3</b>	<b>Test e Risultati</b>	<b>39</b>
3.1	Obiettivi del Test . . . . .	39
3.1.1	Approccio e Strumenti di Test . . . . .	39
3.2	Metodologia di Test . . . . .	40
3.3	Risultati e Analisi . . . . .	41
3.3.1	Qualità del tracking visivo . . . . .	42
3.3.2	Risoluzione e frequenza di aggiornamento . . . . .	42
3.3.3	Rumore e drift dei sensori . . . . .	42
3.4	Analisi visiva . . . . .	43
3.5	Test di <i>usabilità</i> . . . . .	46
<b>4</b>	<b>Conclusioni</b>	<b>49</b>
4.1	Discussione . . . . .	49
4.2	Sviluppi futuri . . . . .	51

# List of Figures

1.1	Schema di comunicazione tra Isaac Sim e ROS 2 tramite il <i>ROS 2 Bridge</i> . Ogni blocco rappresenta un nodo della pipeline, responsabile di uno specifico compito di elaborazione o trasferimento dati. . . . .	7
1.2	Schema a livelli di ROS 2. Dall'alto verso il basso: livello applicativo con i nodi dell'utente; librerie client <code>rclcpp/rclpy</code> ; interfaccia <code>rmw</code> come astrazione del middleware; implementazioni DDS; sistema operativo. Ogni elemento della figura è descritto nel testo. . . . .	10
1.3	Flusso end-to-end di un messaggio in ROS 2 tramite Cyclone DDS, rappresentato in forma verticale su due colonne. . . . .	13
2.1	Architettura del sistema: a sinistra il percorso <i>dispositivo</i> → <i>API</i> (tracking/input in andata, video/audio in ritorno), al centro l'estrazione e pubblicazione in ROS2, a destra l'applicazione dei comandi in Isaac Sim. Colori: azzurro (sistema/driver), arancione (nodi ROS2), verde (topic e dati). . . . .	24
2.2	Flusso tecnologie VR: SteamVR, OpenVR e ALVR . . . . .	24
2.3	Confronto Assi VR e Isaac Sim - Isaac . . . . .	32
3.1	Confronto errore in cm - Test singolo . . . . .	41
3.2	Confronto errore in cm su 5 test ripetuti . . . . .	41
3.3	Vista dall'alto dell'ambiente di test utilizzato per la validazione del sistema.	43
3.4	Confronto tra la posa reale (sopra) e la corrispondente rappresentazione in VR (sotto). . . . .	44
3.5	Sequenza di frame che mostra il movimento del braccio robotico verso la posizione indicata dall'utente in VR. . . . .	45
3.6	Tempi medi di completamento del compito di manipolazione (media su tre tentativi per ciascun utente). . . . .	47



# List of Tables

1.1 Principali differenze tra ROS 1 e ROS 2 . . . . .	11
---	----



# Introduction

La robotica rappresenta oggi uno dei settori più dinamici e innovativi dell'ingegneria, con applicazioni che spaziano dall'automazione industriale alla medicina, fino ai sistemi collaborativi in grado di interagire con l'essere umano. Un aspetto cruciale nello sviluppo di soluzioni robotiche è la possibilità di disporre di strumenti che consentano di testare e validare algoritmi e architetture prima di passare alla fase di implementazione su hardware reale. L'impiego della *simulazione* in questo contesto risulta particolarmente utile: essa permette di ridurre costi e tempi di sviluppo, evitare rischi legati all'utilizzo di prototipi fisici e offrire un ambiente controllato e ripetibile per l'analisi delle prestazioni.

In parallelo, la *realità virtuale* (VR) sta emergendo come una tecnologia capace di arricchire l'interazione uomo-macchina. L'integrazione tra VR e robotica apre nuove prospettive, consentendo di acquisire dati in modo naturale tramite i movimenti dell'utente e di trasferire tali informazioni al robot, rendendo l'interazione più intuitiva e immediata. Questo approccio non solo facilita il controllo dei sistemi robotici, ma consente anche di generare dataset preziosi per l'addestramento di algoritmi di *intelligenza artificiale*.

Lo scopo di questa tesi è sviluppare un *ambiente di simulazione* che consenta l'acquisizione e il controllo di un braccio robotico mediante interfacce di realtà virtuale. In particolare, è stato scelto di utilizzare il braccio robotico **KUKA LBR iiwa R800** come modello di riferimento e il visore **Oculus Quest 2** come dispositivo di input per la realtà virtuale. La scelta di operare in un contesto simulato consente di sperimentare in sicurezza le tecniche di controllo e valutarne l'efficacia prima di una futura applicazione su robot reali.

Lo studio è stato realizzato nell'ambito del progetto europeo **INVERSE** (*INteractive VR Environment for Robotic Simulation and Education*)<sup>1</sup>, che ha come obiettivo la creazione di soluzioni innovative per l'uso della VR nella robotica e nella formazione.

---

<sup>1</sup>[www.inverse-project.org](http://www.inverse-project.org)

---

La *metodologia sviluppata* si basa sull'integrazione di tre elementi fondamentali. In primo luogo, la realtà virtuale viene utilizzata per acquisire i dati di posizione e orientamento della testa e delle mani dell'utente tramite i sensori integrati nel visore **Oculus Quest 2**. Queste informazioni costituiscono l'input principale per il controllo del robot. Successivamente, i dati raccolti vengono elaborati all'interno dell'infrastruttura di comunicazione *ROS2*, dove vengono applicati gli algoritmi di *cinematica inversa* per calcolare i valori dei giunti corrispondenti del braccio robotico **KUKA LBR iiwa R800**. Infine, i valori di giunto ottenuti vengono inviati al simulatore *Isaac Sim*, che consente di riprodurre in modo fedele e realistico il comportamento del robot in un ambiente virtuale interattivo. In questo modo, i movimenti dell'utente nello spazio virtuale vengono tradotti in comandi eseguibili, permettendo al braccio robotico simulato di replicarli con coerenza e precisione. Inoltre, tali movimenti e i corrispondenti valori di giunto vengono registrati e salvati in appositi file, così da costituire una base di dati riutilizzabile per applicazioni future, come l'addestramento di algoritmi di apprendimento automatico o la riproduzione di traiettorie predefinite.

Un aspetto centrale in questo processo è rappresentato dalla *cinematica inversa*. L'utente che utilizza il visore e i controller VR ragiona naturalmente in termini di spazio cartesiano: spostare la mano in avanti, alzare il braccio, ruotare il polso. Il robot, invece, può muoversi solo modificando gli angoli dei suoi giunti. La cinematica inversa agisce dunque come un "ponte" tra questi due mondi: traduce i movimenti nello spazio virtuale (pose cartesiane) nei corrispondenti valori di giunto necessari al robot per riprodurli. Senza questo passaggio, non sarebbe possibile ottenere un controllo intuitivo del robot tramite la VR, poiché l'utente sarebbe costretto a ragionare in termini di configurazioni articolari, cosa complessa e poco naturale. L'integrazione tra VR e cinematica inversa è quindi fondamentale per consentire una mappatura diretta e coerente tra l'interazione dell'utente e il comportamento del robot, rendendo l'esperienza immersiva, precisa e adatta sia a scopi di ricerca sia a contesti formativi.

Gli strumenti utilizzati sono stati scelti per le loro caratteristiche di affidabilità, modularità e diffusione nella comunità scientifica:

- *Isaac Sim*, sviluppato da NVIDIA, come piattaforma di simulazione robotica ad alta fedeltà;
- *ROS2*, per la gestione della comunicazione e l'integrazione dei diversi moduli software;
- *OpenVR* e *ALVR*, per l'acquisizione dei dati dal visore e la gestione dell'interfaccia VR;
- algoritmi di *cinematica inversa* per la traduzione dei movimenti acquisiti in traiettorie robotiche.

Il *contributo principale* di questa tesi è la realizzazione di un'infrastruttura integrata che collega in modo coerente la realtà virtuale, la simulazione robotica e la cinematica inversa, dimostrando la fattibilità del controllo di un robot tramite VR in un ambiente simulato.

Tale infrastruttura rappresenta una base solida per futuri sviluppi, come l'integrazione con robot reali e l'utilizzo in ambito formativo o di ricerca.

La tesi è articolata come segue: il *Capitolo 1* introduce le tecnologie adottate, descrivendone il funzionamento e le motivazioni alla base della loro scelta; il *Capitolo 2* illustra l'implementazione della pipeline, includendo i processi di raccolta ed elaborazione dei dati, nonché il calcolo della cinematica inversa; il *Capitolo 3* presenta la fase di validazione e i test effettuati per valutare il sistema; infine, il *Capitolo 4* discute i risultati ottenuti, i limiti individuati e possibili prospettive di miglioramento.



# **–1–**

## **Tecnologie utilizzate**

CONTENTS: **1.1 Isaac Sim.** 1.1.1 Isaac SDK – 1.1.2 Isaac Simulator – 1.1.3 Ragioni della scelta di Isaac – 1.1.4 Comunicazione tra Isaac Sim e ROS 2 – 1.1.5 Dominio ROS 2 e middleware DDS. **1.2 ROS 2.** 1.2.1 Visione d’insieme e architettura – 1.2.2 Librerie client e livello `rmw`: chiarimento terminologico – 1.2.3 Perché ROS 2 in questo lavoro – 1.2.4 Differenze principali tra ROS 1 e ROS 2 – 1.2.5 Cyclone DDS: scoperta e scambio dati – 1.2.6 Messaggi in ROS 2 e pipeline `.msg` → `.idl`. **1.3 ALVR.** 1.3.1 Panoramica dell’architettura client-server – 1.3.2 Virtualizzazione del visore in SteamVR – 1.3.3 Flusso di funzionamento end-to-end – 1.3.4 Motivazioni della scelta di ALVR. **1.4 OpenVR e SteamVR.** 1.4.1 Ruolo di OpenVR – 1.4.2 Ruolo del runtime SteamVR – 1.4.3 Gestione del tracking e degli input – 1.4.4 Ragioni della scelta di SteamVR e OpenVR.

### **1.1 Isaac Sim**

*NVIDIA Isaac* è una suite di strumenti software e hardware progettata per accelerare lo sviluppo, l’addestramento e il deployment di sistemi robotici. La piattaforma è concepita per ridurre drasticamente i tempi di progettazione e validazione dei robot, permettendo di passare rapidamente dalla fase di *simulazione* a quella di implementazione reale. Isaac si basa sull’impiego intensivo di *GPU* per garantire elevate prestazioni e sfrutta tecnologie NVIDIA come *CUDA*, *TensorRT* e *RTX ray tracing* per un’elaborazione dati ottimizzata.

#### **1.1.1 Isaac SDK**

L’*Isaac Software Development Kit (SDK)* è un insieme di librerie, modelli preaddestrati e moduli software che consentono di gestire le principali funzionalità di un robot autonomo. Le aree di applicazione comprendono la *percezione*, con algoritmi ottimizzati per GPU dedicati all’elaborazione delle immagini e al riconoscimento degli oggetti; la *pianificazione del movimento*, con moduli per la generazione di traiettorie ottimali in scenari complessi; e l’*integrazione con ROS/ROS 2*, resa possibile da interfacce e bridge che semplificano

---

l'inserimento del simulatore all'interno di un'architettura robotica standard. L'SDK è organizzato secondo un'*architettura modulare a grafo*, detta *Graph Execution Framework*, che permette di collegare tra loro nodi di elaborazione e comunicazione per costruire pipeline personalizzate e facilmente modificabili.

### 1.1.2 Isaac Simulator

L'*Isaac Simulator* rappresenta il cuore della piattaforma, ed è sviluppato sulla base della tecnologia *NVIDIA Omniverse*. È un *ambiente di simulazione ad alta fedeltà* che funge da vero e proprio motore grafico, capace di garantire *realismo* nella rappresentazione della realtà fisica. Questa caratteristica risulta particolarmente utile nella *validazione dei sistemi*, poiché permette di verificare il comportamento del robot in un contesto realistico e controllato, accelerando così il passaggio dalla sperimentazione virtuale all'implementazione su hardware reale.

### 1.1.3 Ragioni della scelta di Isaac

La scelta di utilizzare Isaac Sim in questo lavoro è stata motivata da diversi fattori. In primo luogo, Isaac è uno *strumento leader nel settore* ed è sostenuto sia dalla *community scientifica* sia da *NVIDIA*, che ne garantisce un aggiornamento costante e il supporto tramite plugin e documentazione. Inoltre, la sua *integrazione nativa con ROS 2* costituisce un aspetto fondamentale per il progetto: i sensori del visore VR devono comunicare in tempo reale con il braccio robotico e tale scambio è reso possibile dal *ROS 2 Bridge* fornito dal simulatore. Un ulteriore vantaggio è rappresentato dalla natura *modulare e scalabile* dell'architettura a grafo, che consente di sostituire o integrare componenti della pipeline senza doverla ricostruire da zero.

### 1.1.4 Comunicazione tra Isaac Sim e ROS 2

Uno degli elementi chiave della piattaforma è il *ROS 2 Bridge*, che abilita la comunicazione diretta tra la simulazione e nodi ROS 2 esterni. Questo modulo integra *nodi ROS 2 virtuali*, eseguiti all'interno del simulatore, in grado di pubblicare e sottoscrivere *topic* in tempo reale. Grazie a tale architettura, Isaac Sim è in grado sia di *ricevere comandi* provenienti da nodi ROS 2 (ad esempio valori di giunto calcolati tramite la *cinematica inversa*), sia di *inviare dati di simulazione* come stati dei giunti o output dei sensori virtuali.

In *Figura 1.1* è mostrato un esempio di pipeline realizzata con l'architettura a grafo di Isaac Sim per la comunicazione con ROS 2. La figura rappresenta lo *scambio di informazioni* tra diversi nodi: l'evento *On Playback Tick* gestisce il ciclo di esecuzione della simulazione e attiva gli altri moduli. Il blocco *ROS2 Subscribe Joint State* riceve dai nodi ROS 2 esterni lo *stato dei giunti* (posizione, velocità e sforzo), che viene poi passato all'*Articulation Controller*, il quale controlla il modello del robot simulato. Parallelamente, il nodo *Isaac Read Simulation Time* fornisce il *tempo di simulazione*, necessario per garantire la sincronizzazione. Infine, il modulo *ROS2 Publish Joint State* invia ai nodi ROS 2 esterni lo *stato*

dei giunti calcolato nella simulazione, permettendo così un continuo *scambio bidirezionale di informazioni*.

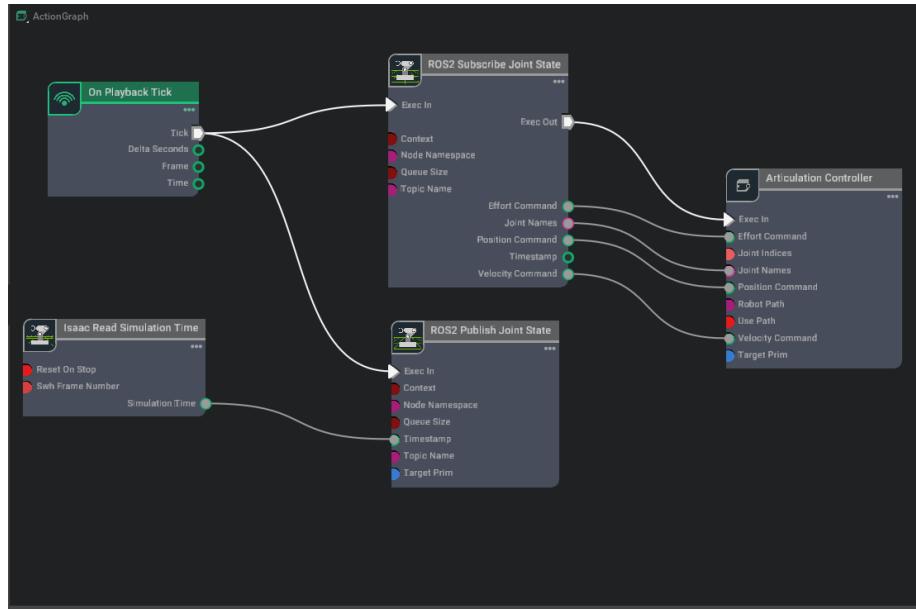


Figure 1.1: Schema di comunicazione tra Isaac Sim e ROS 2 tramite il *ROS 2 Bridge*. Ogni blocco rappresenta un nodo della pipeline, responsabile di uno specifico compito di elaborazione o trasferimento dati.

### 1.1.5 Dominio ROS 2 e middleware DDS

La comunicazione tra i diversi nodi in *ROS 2* si basa sul concetto di *dominio*, ovvero uno spazio di comunicazione isolato in cui più nodi possono scambiarsi messaggi senza interferire con altri processi ROS 2 attivi sulla stessa macchina o in rete. Il dominio è identificato dalla variabile d'ambiente `ROS_DOMAIN_ID`, che assume il ruolo di vero e proprio identificatore di rete: due nodi ROS 2 possono comunicare fra loro solo se condividono lo stesso valore di dominio. Questo meccanismo garantisce che più sistemi robotici possano coesistere nello stesso ambiente computazionale senza conflitti, fornendo una chiara separazione logica delle comunicazioni. All'interno di ciascun dominio, la trasmissione dei messaggi avviene tramite un middleware basato su *DDS* (*Data Distribution Service*). DDS è uno standard industriale ampiamente adottato per la comunicazione distribuita, progettato per offrire *scalabilità*, *bassa latenza* e *affidabilità* nello scambio di dati in tempo reale. In ROS 2, DDS funge da strato di astrazione che consente ai nodi di pubblicare e sottoscrivere messaggi senza preoccuparsi della gestione esplicita delle connessioni, in quanto la distribuzione dei dati è automatica e trasparente. Nel caso specifico di *ROS 2 Humble*, adottato in questo lavoro, il middleware predefinito è **Cyclone DDS**.

---

## 1.2 ROS 2

*ROS 2 (Robot Operating System 2)* è un framework open-source per lo sviluppo di applicazioni robotiche. Non è un sistema operativo in senso stretto: fornisce librerie, strumenti e convenzioni che permettono di comporre applicazioni complesse a partire da componenti più semplici. L’obiettivo è rendere più agevole la comunicazione tra sensori, attuatori e moduli di elaborazione, sia quando tali componenti sono eseguiti sullo stesso computer sia quando sono distribuiti su una rete.

### 1.2.1 Visione d’insieme e architettura

L’architettura di ROS 2 è una serie di strati(Figura 1.2) che si appoggiano l’uno all’altro. Nella parte più alta si trova il livello applicativo, cioè il codice sviluppato dall’utente: qui vivono i *nodi*, unità logiche che svolgono compiti specifici come acquisire dati da un sensore, elaborare un’immagine, o comandare un attuatore. Questi nodi comunicano tra loro in diversi modi. La modalità più comune è lo scambio di *messaggi* attraverso i *topic*, un meccanismo asincrono e uno-a-molti che consente a un publisher di inviare dati a più subscriber. Accanto ai topic esistono i *servizi*, che realizzano una comunicazione di tipo richiesta/risposta in modo sincrono, e le *azioni*, pensate per gestire operazioni più lunghe nel tempo, con la possibilità di fornire aggiornamenti intermedi e persino di annullare l’esecuzione. Per scrivere questi nodi, ROS 2 mette a disposizione delle **librerie client**, come `rclcpp` (per C++) e `rclpy` (per Python). Queste librerie rappresentano il punto di contatto principale per lo sviluppatore: offrono un insieme di API coerenti che permettono di creare nodi, pubblicare e sottoscrivere topic, definire servizi e azioni, gestire parametri e controllare il ciclo di vita delle applicazioni. Un aspetto importante è che queste librerie non dialogano direttamente con il middleware di comunicazione, ma si appoggiano a uno strato intermedio chiamato **rmw** (*ROS Middleware Interface*). Il livello rmw può essere visto come un traduttore: riceve le richieste provenienti dal codice dell’applicazione e le trasforma in chiamate comprensibili dal middleware sottostante. In questo modo l’applicazione rimane indipendente dall’implementazione concreta del trasporto. A seconda della configurazione scelta, infatti, il livello rmw può appoggiarsi a diversi middleware DDS, come Cyclone DDS, Fast DDS o RTI Connext. Ciò consente di adattare ROS 2 a scenari differenti senza dover modificare il codice dei nodi, semplicemente cambiando l’implementazione rmw utilizzata. Alla base di tutto si trova il **sistema operativo**, che fornisce le funzionalità di basso livello necessarie al funzionamento del middleware: rete, gestione dei thread, temporizzazione e pianificazione. Sebbene lo sviluppo principale di ROS 2 avvenga su Linux, il sistema è portabile anche su Windows e macOS, ampliando così le possibilità di adozione in contesti diversi.

**Chi crea i nodi?** I nodi non vengono avviati da un’entità centrale, ma sono creati direttamente dall’utente all’interno dei propri processi applicativi, sfruttando le API di `rclcpp` o `rclpy`. Questo significa che ROS 2 non ha un “server centrale” che gestisce o coordina i nodi, a differenza di quanto avveniva in ROS 1 con il `roscore`. Ogni nodo è

quindi autonomo e si scopre dinamicamente in rete, rendendo l'architettura più distribuita e flessibile.

### 1.2.2 Librerie client e livello rmw: chiarimento terminologico

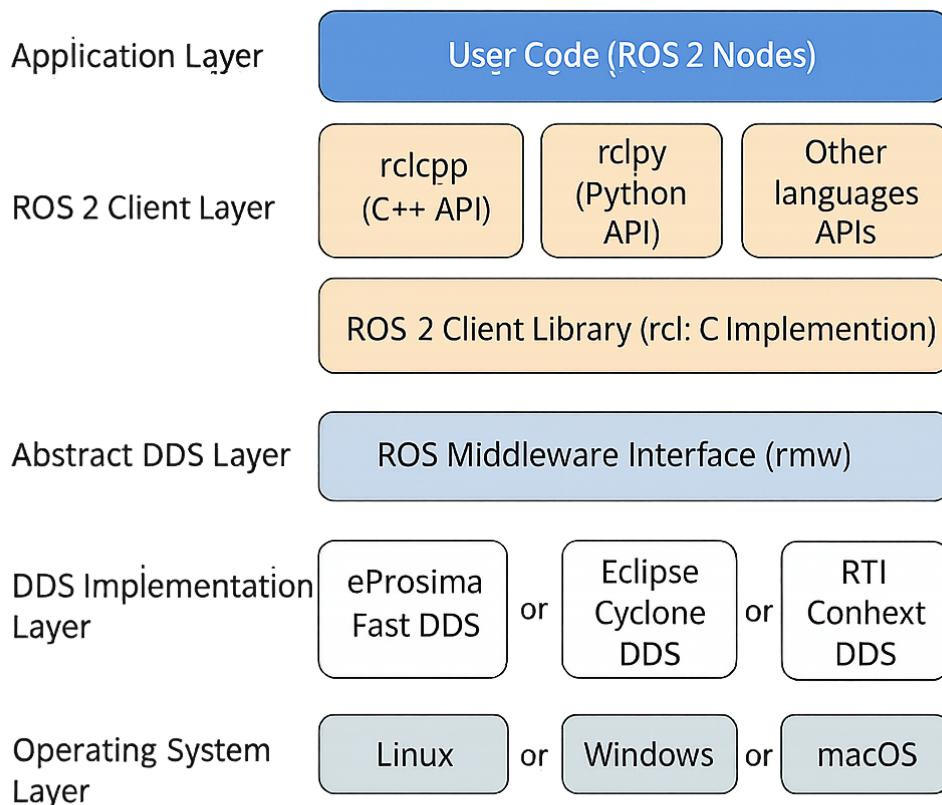
Nel contesto di ROS 2 capita spesso di trovare riferimenti a un “layer DDS” o a una generica astrazione del middleware. In realtà il termine corretto è `rmw` (ROS Middleware interface). Si tratta di uno strato intermedio che ha il compito di nascondere i dettagli del middleware di comunicazione e di offrire alle librerie client di ROS 2 (come `rclcpp` in C++ o `rclpy` in Python) un insieme uniforme di funzioni.

Grazie a questa interfaccia, uno sviluppatore può creare publisher, subscriber, servizi o azioni e configurare le politiche di qualità del servizio senza doversi preoccupare di come queste operazioni verranno effettivamente realizzate dal middleware sottostante. In altre parole, il livello `rmw` funge da “traduttore”: riceve le chiamate provenienti dal codice dell'applicazione e le mappa sulle API concrete di un determinato middleware DDS.

Esistono infatti diverse implementazioni di `rmw`, ciascuna pensata per collegarsi a un particolare middleware. Ad esempio, `rmw_cyclonedds` usa Cyclone DDS, `rmw_fastrtps` si appoggia a Fast DDS, mentre `rmw_connextdds` è compatibile con RTI Connex. Il vantaggio di questa architettura è che l'applicazione ROS 2 non deve cambiare: basta scegliere quale implementazione `rmw` utilizzare, e i messaggi verranno gestiti dal relativo middleware in maniera trasparente.

È importante sottolineare che lo **standard DDS** non fa parte di ROS 2, ma è una specifica esterna definita dall'OMG (Object Management Group). ROS 2 si limita a sfruttarlo attraverso l'interfaccia `rmw`, che costituisce dunque il vero punto di collegamento tra il livello applicativo di ROS e i diversi middleware disponibili.

# ROS 2 Architecture Overview



DDS ≡ Data Distribution Service is a decentralized, publish subscribe communication protocol. rmw

= ROS Middleware interface hides the details of the DDS implementations.

Use rclcpp for efficiency and fast response times, use rclpy for prototyping and shorter development time.

Figure 1.2: Schema a livelli di ROS 2. Dall'alto verso il basso: livello applicativo con i nodi dell'utente; librerie client rclcpp/rclpy; interfaccia rmw come astrazione del middleware; implementazioni DDS; sistema operativo. Ogni elemento della figura è descritto nel testo.

### 1.2.3 Perché ROS 2 in questo lavoro

La scelta di ROS 2 è motivata da ragioni tecniche e pratiche. In primo luogo, la *modularità*: il modello a nodi permette di separare chiaramente acquisizione, percezione e controllo, semplificando sviluppo e test. In secondo luogo, l'integrazione con strumenti esterni come NVIDIA Isaac (e Isaac Sim), che offrono bridge ROS 2 e una rappresentazione a grafo coerente con la filosofia a nodi. Infine, la comunicazione decentralizzata su DDS è adatta sia alla comunicazione “intra-robot” (tra processi sulla stessa macchina) sia a quella “inter-robot” o tra dispositivi diversi in rete, caratteristica utile quando si devono mettere in relazione, ad esempio, i sensori di un visore VR e un braccio robotico.

### 1.2.4 Differenze principali tra ROS 1 e ROS 2

Rispetto a ROS 1, ROS 2 elimina la dipendenza da un nodo centrale (`roscore`) per registrazione e discovery, introducendo discovery distribuita basata su DDS. Questo abilita configurazioni multi-robot più semplici, migliora l'affidabilità (assenza di singoli punti di fallimento) e abilita la configurazione fine della QoS. Il supporto al tempo reale è stato migliorato grazie a scelte architettoniche (executor, callback group) e al middleware. Sul fronte sicurezza, ROS 2 integra meccanismi basati su DDS-Security (autenticazione, crittografia, controllo accessi). Pur esistendo progetti di retro-compatibilità, lo sviluppo attivo è incentrato su ROS 2, mentre ROS 1 è in manutenzione.

Table 1.1: Principali differenze tra ROS 1 e ROS 2

Aspetto	ROS 1	ROS 2
<b>Middleware</b>	<code>roscore</code> , TCPROS/UDPROS; discovery centralizzato.	DDS con discovery distribuita; nessun nodo centrale.
<b>Tempo reale</b>	Limitato.	Migliorato (QoS, executor, configurazioni RT).
<b>QoS</b>	Non configurabile.	Configurabile per topic/servizi/azioni.
<b>Sicurezza</b>	Assente nativamente.	DDS-Security (autenticazione, cifratura, ACL).
<b>Piattaforme</b>	Principalmente Linux.	Linux, Windows, macOS; integrazione con micro-ROS.
<b>Multi-robot</b>	Configurazioni ad hoc.	Nativo grazie a discovery DDS.
<b>Stato del progetto</b>	Manutenzione.	Sviluppo attivo.

---

### 1.2.5 Cyclone DDS: scoperta e scambio dati

Cyclone DDS è un’implementazione open-source dello standard DDS, utilizzata in ROS 2 come middleware di comunicazione. Le comunicazioni avvengono tipicamente sopra il protocollo di trasporto UDP, scelto perché leggero e adatto a scenari distribuiti. Il primo passo fondamentale è la **scoperta** (*discovery*), cioè il meccanismo tramite cui i diversi nodi che partecipano alla rete DDS imparano a “vedersi” e a riconoscersi a vicenda. Questo processo avviene in due fasi principali:

1. **Scoperta dei partecipanti** – ogni nodo che entra in un dominio DDS invia dei messaggi di annuncio (ad esempio tramite multicast). In questo modo gli altri nodi attivi nello stesso ROS\_DOMAIN\_ID vengono a conoscenza della sua esistenza.
2. **Scoperta degli endpoint** – una volta che i partecipanti si conoscono, si scambiano informazioni dettagliate sugli *endpoint* che gestiscono, cioè i publisher e i subscriber che ciascuno ha creato. In questa fase vengono condivisi anche i topic e i tipi di dati associati.

Quando due partecipanti individuano che esiste una corrispondenza — cioè un publisher e un subscriber che parlano dello stesso topic e utilizzano lo stesso tipo di messaggio — allora si stabilisce una connessione logica che abilita lo **scambio dei dati**. La trasmissione dei messaggi non avviene in formato arbitrario: i dati vengono **serializzati** secondo le regole dell’IDL (Interface Definition Language) e del formato CDR (Common Data Representation). Questa serializzazione garantisce che anche sistemi eterogenei (scritti in linguaggi diversi o su architetture differenti) possano interpretare correttamente i messaggi.

### 1.2.6 Messaggi in ROS 2 e pipeline .msg → .idl

In ROS 2 i messaggi non vengono definiti direttamente nel linguaggio di programmazione, ma tramite file dedicati con estensione `.msg` (e in modo analogo `.srv` e `.action`). Questi file permettono di descrivere in maniera semplice e dichiarativa la struttura del dato, specificando campi, tipi e costanti senza doversi preoccupare dei dettagli di implementazione. A partire da queste descrizioni, entra in gioco la toolchain `rosidl`. Il suo compito è duplice: da un lato genera automaticamente il codice nei diversi linguaggi supportati (C, C++, Python), in modo che gli sviluppatori possano utilizzare i messaggi come normali tipi di dato nelle proprie applicazioni; dall’altro produce una rappresentazione IDL (Interface Definition Language), che costituisce lo standard usato da DDS per garantire interoperabilità tra sistemi eterogenei. In questo modo, publisher e subscriber condividono una definizione rigorosa e comune della struttura dei dati, condizione necessaria per potersi scambiare messaggi in maniera sicura.

Una volta definito e generato il tipo di messaggio, il flusso di comunicazione vero e proprio riprende lo schema già discusso nella Figura ???. Il nodo applicativo crea il messaggio e lo passa allo strato `rmw`, che a sua volta lo inoltra al middleware DDS (ad esempio Cyclone DDS). Qui il dato viene serializzato secondo lo standard CDR, cioè trasformato in un

formato binario compatto e indipendente dalla piattaforma, pronto per essere spedito in rete via UDP.

Sul lato ricevente il processo avviene in maniera speculare: il middleware ricostruisce la struttura originaria deserializzando il flusso ricevuto, lo consegna allo strato rmw e infine il nodo subscriber riceve il messaggio nella propria callback.

In questo senso, la pipeline `.msg → .idl` non è soltanto un meccanismo tecnico di generazione del codice, ma rappresenta il ponte che consente a ROS 2 di offrire agli sviluppatori un’interfaccia semplice e astratta per la definizione dei dati, pur mantenendo la compatibilità con gli standard DDS che regolano serializzazione, trasporto e interoperabilità.

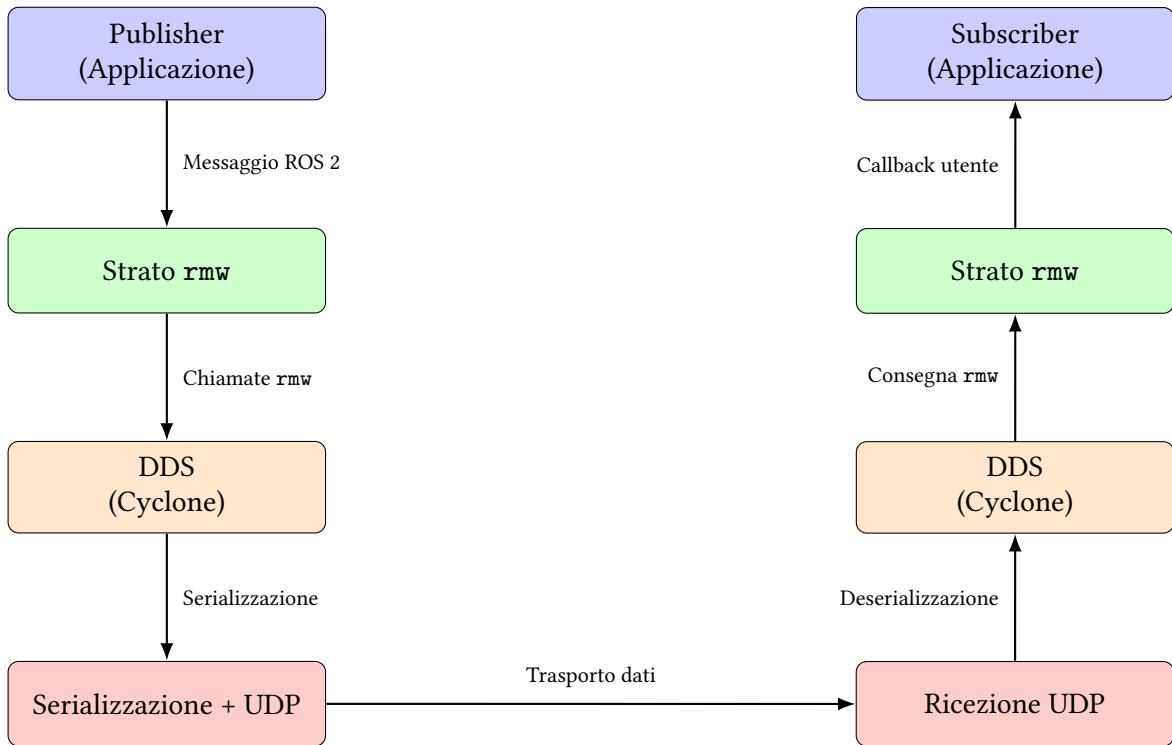


Figure 1.3: Flusso end-to-end di un messaggio in ROS 2 tramite Cyclone DDS, rappresentato in forma verticale su due colonne.

---

## 1.3 ALVR

ALVR (Air Light VR) è un progetto open-source che consente di utilizzare un visore VR *standalone* (ad esempio Meta Quest o Pico) come terminale di visualizzazione per esperienze VR generate da un PC. In pratica, il PC esegue l'applicazione VR e produce i fotogrammi della scena, mentre il visore riceve in streaming *video* e *audio* e, in senso inverso, invia in tempo reale al PC i *dati di tracking* della testa e gli input dei controller. L'obiettivo principale è duplice: da un lato mantenere la latenza complessiva quanto più bassa possibile, dall'altro preservare una qualità visiva elevata anche in modalità completamente wireless. In altre parole, ALVR permette di “staccare il cavo” tra PC e visore senza rinunciare alla reattività tipica delle esperienze VR collegate via cavo.

### 1.3.1 Panoramica dell'architettura client-server

ALVR adotta un'architettura **client-server**. Il **server** è il PC su cui girano i giochi o le applicazioni VR: qui risiedono SteamVR (o altre piattaforme compatibili con OpenXR) e il driver di ALVR che si occupa di virtualizzare un visore agli occhi del sistema. Il **client** è il visore fisico: qui un'app ALVR dedicata acquisisce continuamente la posa della testa e gli input dei controller, li invia al server e, contemporaneamente, riceve il flusso multimediale codificato dal PC per poi decodificarlo e mostrarlo sui display interni del visore. Il lettore può pensare ad ALVR come a un “ponte” bidirezionale: verso il visore viaggiano fotogrammi e audio, verso il PC viaggiano dati di movimento e comandi dell'utente.

#### Componente server (PC)

Sul PC, ALVR installa un driver che crea un *visore virtuale* all'interno di SteamVR. Questo visore virtuale si presenta a SteamVR come se fosse un dispositivo fisico reale: di conseguenza qualsiasi applicazione VR compatibile continua a funzionare senza modifiche, perché crede di parlare con un normale HMD. È un dettaglio importante: *non è necessario cambiare gli applicativi*, poiché l'integrazione avviene a livello di driver.

Ricevuti dal client i dati di tracking (posizione e orientamento della testa, pulsanti e movimenti dei controller), il server aggiorna lo stato del visore virtuale e avvia il rendering della scena tramite GPU, esattamente come farebbe con un visore collegato via cavo. I fotogrammi prodotti vengono poi compressi in tempo reale in H.264 o HEVC (codec selezionabili in base al supporto hardware della GPU) e sincronizzati con l'audio del gioco. Il flusso compresso viene quindi inviato al visore tramite rete locale, preferibilmente su Wi-Fi a 5 GHz o su Ethernet, per minimizzare congestione e latenza.

#### Componente client (visore fisico)

Sul visore, l'app ALVR campiona ad alta frequenza la posa della testa (dati forniti dai sensori e dal sistema di tracciamento integrato) e legge gli input dei controller. Queste

informazioni vengono impacchettate con marcatori temporali e inviate immediatamente al server, così che il PC possa aggiornare la scena con lo stato più recente dell'utente. In parallelo, il client riceve il flusso video/audio in arrivo dal PC, lo decodifica attraverso gli acceleratori hardware del visore e lo presenta ai display interni. In questo modo, ogni movimento della testa viene riflesso sul contenuto mostrato con un ritardo percepito minimo, che è la somma di acquisizione, rete, codifica/decodifica e presentazione.

### 1.3.2 Virtualizzazione del visore in SteamVR

Il cuore del sistema è la *virtualizzazione* operata dal driver scritto con OpenVR personalizzato di ALVR. Dal punto di vista di SteamVR, il dispositivo esposto da ALVR è indistinguibile da un visore fisico: espone pose head-tracked, controller, parametri ottici e superfici di rendering. Dal punto di vista del visore reale, invece, ALVR si comporta come un lettore/decoder di uno *stream* video a bassa latenza che, allo stesso tempo, funge da “telecomando” per il visore virtuale su PC tramite l’invio dei dati di tracking. Questo disaccoppiamento consente di sfruttare l’ecosistema esistente di SteamVR e OpenVR senza intervenire sulle applicazioni finali. In altre parole: le app continuano a parlare con SteamVR; ALVR si occupa di tradurre, inoltrare e sincronizzare tutto via rete.

### 1.3.3 Flusso di funzionamento end-to-end

Il ciclo operativo può essere descritto come segue. *Primo*, il visore misura la posa della testa e cattura gli input dei controller; queste informazioni, accompagnate da timestamp, vengono inviate immediatamente al PC. *Secondo*, il server aggiorna lo stato del visore virtuale in SteamVR con i dati appena ricevuti, così che il motore grafico renderizzi la scena per la vista corretta dell’utente. *Terzo*, la GPU del PC calcola i nuovi fotogrammi stereoscopici; subito dopo, un codificatore hardware (ad esempio H.264 o HEVC) comprime i frame per il trasporto in rete, mantenendo un equilibrio fra qualità e bitrate. *Quarto*, il flusso compresso viene spedito al visore attraverso la rete locale (Wi-Fi o Ethernet), con pacchetti ottimizzati per ridurre ritardi e jitter. *Quinto*, il client sul visore decodifica i fotogrammi appena arrivati e li presenta ai display interni, sincronizzandoli con l’audio. *Infine*, l’intero processo si ripete decine di volte al secondo: mentre il visore mostra un fotogramma, sta già inviando la posa successiva e ricevendo i dati per il fotogramma ancora seguente. Questa pipeline sovrapposta è ciò che permette di percepire un’esperienza fluida e reattiva.

### 1.3.4 Motivazioni della scelta di ALVR

La scelta di ALVR è stata guidata da quattro considerazioni principali. **Stabilità**. Nella pratica d’uso, ALVR si è dimostrato la soluzione più solida e matura per collegare un visore VR a sistemi Linux: il driver virtuale è affidabile, l’integrazione con SteamVR è comprovata e la pipeline di streaming si comporta in modo prevedibile. **Semplicità d’uso**. La configurazione è relativamente lineare: si installa il server sul PC, si avvia il client

---

sul visore e si procede all’abbinamento; non sono richiesti *workaround* complessi né personalizzazioni invasive delle applicazioni VR già esistenti. **Supporto nativo.** Il supporto diretto a Linux evita strati di compatibilità (come Wine/Proton) che potrebbero introdurre ulteriore latenza o problemi di interoperabilità; in altre parole, meno intermediari ci sono nella catena, più facile è garantire prestazioni costanti. **Comunità e aggiornamenti.** Il progetto è sostenuto da una comunità attiva che rilascia aggiornamenti con regolarità, corregge bug e introduce miglioramenti; questo si traduce in tempi di risposta più rapidi in caso di regressioni e in una documentazione ricca di casi d’uso reali. Ripetiamo il concetto chiave: *ALVR consente di usare oggi l’ecosistema SteamVR con visori standalone, su Linux, senza modificare i software esistenti.*

## 1.4 OpenVR e SteamVR

*OpenVR* e *SteamVR*, Questi due elementi lavorano insieme, ma hanno funzioni diverse e complementari: OpenVR è l’interfaccia di programmazione (SDK e API) utilizzata dagli sviluppatori, mentre SteamVR è il runtime che esegue e gestisce tutto il sistema VR sul PC.

### 1.4.1 Ruolo di OpenVR

*OpenVR* è un SDK (Software Development Kit) e un’API progettata da Valve con lo scopo di fornire un’interfaccia unica e standardizzata per lo sviluppo di applicazioni di realtà virtuale. Senza una libreria come OpenVR, ogni sviluppatore dovrebbe scrivere codice specifico per ciascun visore, con il rischio di dover mantenere versioni diverse dello stesso software. Grazie a OpenVR, invece, il programmatore scrive il codice una sola volta e può eseguire la propria applicazione su visori molto diversi fra loro, come HTC Vive, Valve Index, Oculus Rift (attraverso SteamVR) o Windows Mixed Reality.

Il compito principale di OpenVR è dunque quello di fare da “collante” tra il software e l’hardware. Da un lato, fornisce all’applicazione funzioni per leggere i dati di tracking, cioè posizione e orientamento della testa e dei controller, in maniera indipendente dal produttore del visore. Dall’altro lato, mette a disposizione funzioni per inviare i fotogrammi renderizzati al display del visore, curandosi di garantire che la latenza sia sufficientemente bassa per non compromettere l’esperienza immersiva. Un aspetto fondamentale è che tutto questo avviene in modo trasparente per lo sviluppatore, che non deve preoccuparsi di quale tecnologia di tracking (Lighthouse o inside-out) o quale sistema di input (joystick, touchpad, trigger) sia effettivamente utilizzato.

Un ulteriore compito di OpenVR è facilitare l’integrazione con i principali motori grafici, come Unity e Unreal Engine, così da permettere a chi sviluppa contenuti di concentrarsi sugli aspetti creativi e non sui dettagli tecnici della comunicazione con l’hardware. In sintesi, possiamo dire che OpenVR costituisce lo strato di astrazione software che uniforma l’accesso a funzionalità fondamentali come il tracking, gli input e la presentazione grafica.

### 1.4.2 Ruolo del runtime SteamVR

Se OpenVR è l’interfaccia che i programmati utilizzano per scrivere le loro applicazioni, SteamVR è il runtime, cioè il software che viene eseguito sul PC e che mette effettivamente in comunicazione l’applicazione con il visore. In altre parole, mentre lo sviluppatore chiama le funzioni di OpenVR per chiedere “dove si trova il controller?” o “invia questo fotogramma al visore”, è SteamVR che riceve quelle richieste e si occupa di tradurle in operazioni concrete sui dispositivi fisici.

SteamVR mantiene e gestisce i driver dei vari visori e dei loro accessori. Ogni produttore (HTC, Valve, Meta, Microsoft) fornisce un proprio driver, e SteamVR si assicura che questo driver venga caricato e che il dispositivo venga controllato correttamente. Questo meccanismo permette a SteamVR di fungere da “ponte universale”: indipendentemente

---

dal visore collegato, le applicazioni continuano a funzionare nello stesso modo perché parlano con SteamVR attraverso l'API OpenVR.

Un'altra funzione fondamentale di SteamVR è l'orchestrazione del rendering. Quando un'applicazione genera un fotogramma VR, non lo invia direttamente al visore, ma lo consegna a SteamVR. Sarà quest'ultimo a occuparsi di eventuali correzioni ottiche, della sincronizzazione fra i due display (uno per occhio) e della trasmissione finale al dispositivo. In questo modo si garantisce uniformità e stabilità nell'esperienza.

### 1.4.3 Gestione del tracking e degli input

Un punto cruciale del runtime SteamVR riguarda il tracciamento della posizione e l'elaborazione degli input. Senza un sistema accurato di tracking, l'utente non avrebbe la sensazione di immersione. SteamVR supporta diversi metodi di tracciamento a seconda del visore utilizzato.

Nel caso della tecnologia *Lighthouse*, utilizzata ad esempio da HTC Vive e Valve Index, il tracciamento si basa su stazioni base che emettono impulsi laser sincronizzati. Questi impulsi scansionano rapidamente la stanza e colpiscono i fotodiodi installati sul visore e sui controller. Analizzando l'istante preciso e l'angolo con cui ciascun sensore riceve il segnale, SteamVR è in grado di ricostruire con estrema precisione la posizione e l'orientamento del dispositivo nello spazio reale.

Nel caso del tracciamento *inside-out*, adottato da visori più recenti e standalone, la logica è completamente diversa. Qui il visore utilizza telecamere o sensori di profondità montati direttamente sul dispositivo per osservare l'ambiente circostante. Algoritmi di visione artificiale, tipicamente basati su tecniche di *Simultaneous Localization and Mapping* (SLAM), rilevano punti di riferimento stabili nell'ambiente e li utilizzano per calcolare i movimenti dell'utente. Anche in questo scenario, SteamVR raccoglie i dati elaborati e li mette a disposizione dell'applicazione attraverso OpenVR.

Oltre al tracking, SteamVR gestisce gli input dai controller, come pulsanti, trigger, joystick o superfici tattili, e fornisce funzioni per il feedback aptico (ad esempio vibrazioni). Infine, include sistemi di sicurezza come il cosiddetto *Chaperone System*, che mostra all'utente i confini dell'area di gioco per evitare collisioni con oggetti o pareti. Tutti questi aspetti, che per l'utente risultano quasi invisibili, sono in realtà fondamentali per garantire un'esperienza VR stabile, immersiva e sicura.

### 1.4.4 Ragioni della scelta di SteamVR e OpenVR

La scelta di basarsi su SteamVR e OpenVR deriva da considerazioni sia pratiche sia strategiche. In primo luogo, la coppia OpenVR-SteamVR offre una compatibilità estesa con una vasta gamma di visori, dai più diffusi come HTC Vive e Valve Index, fino a dispositivi come Oculus Rift o Windows Mixed Reality. Questo garantisce flessibilità e longevità al progetto: lo stesso software può funzionare con hardware diverso senza modifiche sostanziali.

In secondo luogo, SteamVR costituisce un runtime consolidato e stabile, in grado di gestire in modo affidabile tracciamento, input, rendering e sicurezza. Affidarsi a un runtime ufficiale riduce i rischi di incompatibilità e assicura che le applicazioni funzionino in un ambiente ben ottimizzato.

Terzo aspetto: OpenVR fornisce un livello di astrazione hardware che semplifica notevolmente il lavoro degli sviluppatori. Scrivere il codice una sola volta e vederlo funzionare su diversi visori è un vantaggio concreto in termini di tempo, manutenzione e riduzione degli errori.

Infine, la scelta è stata favorita dalla presenza di un ecosistema maturo e supportato da una comunità ampia. SteamVR non è soltanto un runtime tecnico: offre anche strumenti accessori come la dashboard in VR, ambienti personalizzabili e un'integrazione nativa con l'infrastruttura multiplayer di Steam. Tutto questo rende la piattaforma non solo una soluzione tecnica, ma anche un ambiente completo per sviluppare, distribuire e utilizzare esperienze VR.

In conclusione, OpenVR e SteamVR rappresentano oggi lo standard di fatto per l'integrazione di visori VR su PC. OpenVR semplifica la vita agli sviluppatori fornendo un'interfaccia unificata, mentre SteamVR assicura che quella interfaccia corrisponda a un'esperienza utente stabile e performante. Questa combinazione giustifica pienamente la scelta adottata in questo lavoro.



# **-2-**

# **Implementazione**

CONTENTS: **2.1 Architettura.** **2.2 Utilizzo della libreria OpenVR e SteamVR.** 2.2.1 Codice d'esempio: Inizializzazione OpenVR e selezione del controller – 2.2.2 Codice d'esempio: Raccolta dati. **2.3 Utilizzo di ALVR.** **2.4 Calcolo del orientamento della mano.** 2.4.1 Struttura della matrice – 2.4.2 Significato geometrico – 2.4.3 Esempio numerico – 2.4.4 Trasformazione delle rotazioni da VR a Robot. **2.5 Cinematica inversa per calcolo dei giunti.** 2.5.1 Cinematica diretta (il problema facile) – 2.5.2 Cinematica inversa (il problema difficile) – 2.5.3 Algoritmo numerico iterativo per l'IK. **2.6 Il formato URDF.** 2.6.1 Definizione e contesto – 2.6.2 Contenuti di un file URDF – 2.6.3 Funzione pratica dell'URDF.

## **2.1 Architettura**

In Figura 2.1 è riportato lo schema architetturale completo adottato per acquisire i dati dal visore di realtà virtuale, convogliare tali dati in ROS2 e controllare in tempo reale un robot simulato in *Isaac Sim* oppure salvare i dati al interno di un dataset. A sinistra è rappresentata la catena di dispositivi e software che mette in comunicazione il visore fisico con il PC (visore → ALVR → SteamVR → OpenVR); al centro sono mostrati i nodi ROS2 responsabili della pubblicazione ed elaborazione dei dati; a destra è illustrata l'integrazione con il simulatore, che riceve i comandi articolari e li applica al modello del robot. Nella figura, i **blocchi azzurri** indicano componenti o librerie di sistema (visore, ALVR, SteamVR, API OpenVR), i **blocchi arancioni** rappresentano processi applicativi ROS2 (nodi *Publisher/Subscriber*), mentre i **blocchi verdi** denotano flussi o contenitori di dati (topic ROS2 e file CSV). Le frecce orizzontali in alto da sinistra verso destra indicano il *flusso di andata* (tracking e input che partono dal visore e arrivano fino a OpenVR), mentre le frecce orizzontali in alto da destra verso sinistra rappresentano il *flusso di ritorno* (frame video/audio prodotti sul PC e inviati al visore), così da rendere evidente la natura bidirezionale del sistema.

---

## Dal visore a OpenVR: acquisizione e normalizzazione dei dati

Il punto di partenza è il **visore VR**, che misura in continuo la posa dei controller dell'operatore (posizione e orientamento) e gli input dei controller (grilletto, grip, pulsanti e joystick). Questi segnali, nativamente dipendenti dal produttore del dispositivo, vengono inoltrati ad **ALVR**, il quale agisce da *driver bridge*: crea su PC un visore virtuale “fantoccio” compatibile con SteamVR e trasporta in rete i dati del dispositivo reale. **SteamVR** riceve da ALVR i pacchetti di tracking e input, li sincronizza e li espone in modo uniforme alle **API OpenVR**. A questo punto, dal punto di vista del software, i dati sono stati “normalizzati”: un'applicazione che interroga OpenVR ottiene pose e input in un formato standard, indipendente dall'hardware fisico realmente in uso. Questa catena è importante perché separa la specificità del visore (gestita da ALVR/SteamVR) dalla logica applicativa che, in quanto tale, può rimanere invariata.

## Pubblicazione in ROS2: dalla libreria all'ecosistema di messaggistica

Un **nodo ROS2 Publisher** interroga in tempo reale le API OpenVR e pubblica i dati grezzi su due topic distinti, mantenendo la separazione semantica tra stato di movimento e comandi utente. In particolare, il topic `vr/pose` veicola la posa del controller (o, più in generale, dell'*effector* controllato dall'operatore), mentre il topic `vr/trigger` riporta i valori analogici dei grilletti/pulsanti. La disponibilità di questi topic consente a qualsiasi altro nodo ROS2 di sottoscriversi e consumare le stesse informazioni con un contratto chiaro e stabile, favorendo la componibilità della soluzione.

## Cinematica inversa: dove, come e perché viene calcolata

La trasformazione dai movimenti dell'operatore ai comandi per il robot richiede la **cinematica inversa** (IK). Il calcolo avviene in un **nodo ROS2 con doppio ruolo Subscriber/Publisher**, che si sottoscrive ai topic `vr/pose` e `vr/trigger` e, sulla base della posa desiderata dell'organo terminale (derivata dal controller del visore), determina i **valori articolari** del manipolatore.

*Dove:* l'IK è eseguita localmente sul PC che esegue ROS2, all'interno del nodo di elaborazione dedicato, per minimizzare la latenza rispetto allo stream dei dati VR. *Come:* il nodo applica un risolutore numerico (ad esempio metodi basati sul Jacobiano come *damped least squares* o pseudoinversa con vincoli), tenendo conto dei limiti articolari, di eventuali priorità sui giunti e di criteri di regolarità del movimento. L'input del algoritmo di IK è la posa target dell'*end-effector* (posizione e orientamento), mentre l'output è il vettore di angoli/posizioni dei giunti compatibile con la cinematica diretta del robot. *Perché:* l'IK è necessaria per passare dal mondo “utente” (pose in spazio cartesiano, intuitive da impartire con un controller) al mondo “robotico” (comandi in spazio dei giunti, univocamente interpretabili da un controllore). Senza IK, la stessa posa cartesiana non sarebbe immediatamente realizzabile dal robot in termini di comandi articolari.

Oltre al calcolo in tempo reale, il nodo salva le sequenze di giunti e gli orientamenti della mano su un **file CSV** per analisi offline (verifica di stabilità, riproducibilità degli esperimenti, confronto tra diverse strategie di controllo). In parallelo, pubblica i comandi articolari risultanti sul topic `/joint_command`, rendendoli disponibili al livello di simulazione.

## Elaborazione dei dati

Un aspetto fondamentale che il nodo incaricato della cinematica inversa deve gestire la differenza tra i sistemi di riferimento del visore VR e quelli utilizzati da *Isaac Sim*. Infatti, i due ambienti non condividono la stessa convenzione per gli assi cartesiani: il visore utilizza un sistema orientato in base al punto di vista dell'utente, mentre il simulatore adotta un sistema coerente con la robotica classica e con l'ambiente 3D della simulazione. Se i dati venissero utilizzati direttamente senza alcuna trasformazione, i movimenti dell'operatore non corrisponderebbero a quelli del robot, generando rotazioni e traslazioni incoerenti. Per risolvere questo problema, il nodo che elabora le pose dei controller applica una trasformazione di riferimento, riallineando gli assi del visore con quelli di *Isaac Sim*. In questo modo, quando l'utente muove la mano in avanti, il robot in simulazione replica il movimento nella stessa direzione, garantendo una corrispondenza intuitiva e naturale tra mondo virtuale e modello robotico.

## Verso la simulazione: applicazione dei comandi in Isaac Sim

Il **bridge ROS2 di Isaac Sim** si sottoscrive al topic `/joint_command` e applica i comandi ai giunti del modello robotico in simulazione. Questo consente un controllo *end-to-end* in tempo reale: i movimenti impartiti dall'operatore nel visore si traducono, dopo la trasformazione via IK, in movimenti coerenti del robot virtuale. La scelta di separare chiaramente la pubblicazione dei dati grezzi (pose e trigger) dall'elaborazione (IK) e dall'applicazione finale (*Isaac Sim*) favorisce sia la leggibilità dell'architettura sia la possibilità di sostituire componenti senza effetti collaterali (ad esempio cambiando il solver IK o il simulatore).

**Come leggere la figura (colori, frecce e righe)** In Figura 2.1, la riga superiore descrive la pipeline *dispositivo → API*: il visore invia *tracking & input* ad ALVR, ALVR inoltra tali dati a SteamVR e SteamVR li espone tramite OpenVR. Le frecce di ritorno lungo la stessa riga rappresentano lo *stream video/audio* prodotto sul PC e inviato al visore: è questo canale che consente all'operatore di vedere la scena aggiornata. La colonna centrale illustra l'estrazione e la pubblicazione dei dati in ROS2, mentre la colonna di destra racchiude la simulazione, alimentata dal topic `/joint_command`. I colori sono coerenti con il ruolo dei blocchi: **azzurro** per infrastruttura/driver, **arancione** per processi ROS2, **verde** per dati e topic. Questa legenda è riportata anche direttamente nella figura per facilitarne la lettura.

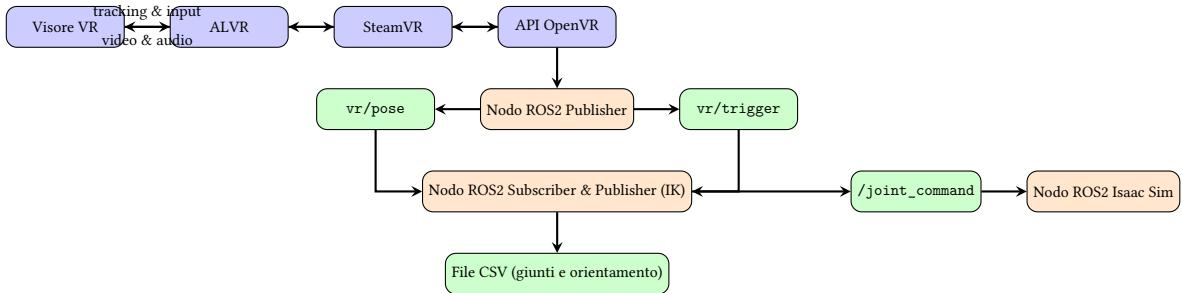


Figure 2.1: Architettura del sistema: a sinistra il percorso *dispositivo* → *API* (tracking/input in andata, video/audio in ritorno), al centro l'estrazione e pubblicazione in ROS2, a destra l'applicazione dei comandi in Isaac Sim. Colori: azzurro (sistema/driver), arancione (nodi ROS2), verde (topic e dati).

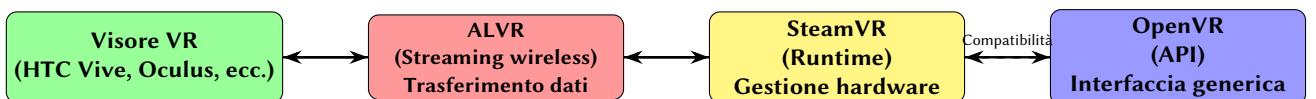
## 2.2 Utilizzo della libreria OpenVR e SteamVR

OpenVR e SteamVR vengono utilizzati per acquisire le informazioni provenienti dal visore. SteamVR rappresenta il *runtime* di riferimento: gestisce la comunicazione diretta con l'hardware del visore e con eventuali controller, fornendo un livello di astrazione che permette di raccogliere tutti i dati necessari (posizione, orientamento, input, stato dei sensori, ecc.) in maniera uniforme.

OpenVR, invece, è l'insieme di API sviluppate da Valve che ho utilizzato nel progetto. Queste API sono pensate per essere indipendenti dal produttore del visore: il codice scritto tramite OpenVR può funzionare con diversi dispositivi compatibili, mentre sarà SteamVR a tradurre le chiamate API in comandi specifici per ciascun modello di visore.

In questo modo, l'applicazione può rimanere generica e portabile, mentre il runtime si occupa della gestione delle peculiarità hardware.

Figure 2.2: Flusso tecnologie VR: SteamVR, OpenVR e ALVR



## 2.2.1 Codice d'esempio: Inizializzazione OpenVR e selezione del controller

Il frammento di codice riportato qui sotto ha il compito di *agganciare* il runtime SteamVR attraverso la libreria `openvr`, inizializzare la sessione e identificare in modo univoco il controller da interrogare. Questo blocco vive all'interno del nodo **ROS2 Publisher** descritto nella **Sezione 2.1**: è il primo passo della catena *OpenVR → ROS2*, perché prepara l'accesso ai dispositivi tracciati (HMD, controller, tracker) e, in particolare, seleziona il *controller destro* come sorgente primaria di pose e input. In altre parole, senza questa inizializzazione e senza la scelta del dispositivo corretto, il nodo non potrebbe leggere in modo affidabile né la posa del controller né i valori dei grilletti da pubblicare sui topic `vr/pose` e `vr/trigger`. Ribadiamo quindi l'idea chiave: questo blocco è il *punto di contatto* fra il mondo SteamVR/OpenVR e il middleware ROS2 che useremo per distribuire i dati al resto del sistema.

```
1 import openvr # Libreria OpenVR per interfacciarsi con SteamVR
2 # Initialize OpenVR
3 openvr.init(openvr.VRApplication_Other)
4 self.vrsystem = openvr.VRSystem()
5
6 # Choose left or right controller; here we use right hand
7 self.controller_index =
8     self._find_controller_index(openvr.TrackedControllerRole_RightHand)
9
10
11 # Trovare l'indice per il controller
12 def _find_controller_index(self, role):
13     # Iterate through tracked devices to find the controller
14     for i in range(openvr.k_unMaxTrackedDeviceCount):
15         if self.vrsystem.getTrackedDeviceClass(i) ==
16             openvr.TrackedDeviceClass_Controller:
17             if self.vrsystem.getControllerRoleForTrackedDeviceIndex(i) ==
18                 role:
19                 return i
20
21     self.get_logger().warn('Controller not found; defaulting to index 0')
22
23 return 0
```

Il codice inizia con l'import della libreria `openvr` (**riga 1**). Questa libreria fornisce l'API necessaria per comunicare con SteamVR, cioè per interrogare il sistema sui dispositivi tracciati (headset e controller), sui loro stati e sulle loro trasformazioni. Senza questo import l'applicazione non avrebbe accesso alle chiamate verso il runtime.

Subito dopo, la sessione OpenVR viene inizializzata (**righe 3-4**).

La chiamata `openvr.init(openvr.VRApplication_Other)` crea un contesto applicativo “generico”: non stiamo avviando un gioco o un’app grafica, ma un processo che vuole leggere dati dal sistema VR. Questa scelta è coerente con il ruolo del nostro nodo

---

ROS2, che deve *consumare* tracking e input piuttosto che presentare frame grafici al visore. L’istruzione seguente `self.vrSystem = openvr.VRSystem()` costruisce l’oggetto attraverso il quale effettueremo tutte le interrogazioni successive: `VRSystem` è l’interfaccia principale per elencare i dispositivi, leggere le loro classi (controller, HMD, tracker) e ottenerne lo stato. Una volta creato l’accesso al sistema, selezioniamo esplicitamente quale controller utilizzare (**riga 7**). L’assegnazione a `self.controller_index` invoca il metodo privato `_find_controller_index` passando il ruolo desiderato, in questo caso `TrackedControllerRole_RightHand`. È un passaggio importante e volutamente esplicito: nelle sezioni successive useremo questo indice per richiedere pose e input; fissare il ruolo a “mano destra” garantisce coerenza con il modello di interazione definito nell’architettura (controller destro come *end-effector* dell’operatore). Se si volesse utilizzare la mano sinistra, basterebbe passare `TrackedControllerRole_LeftHand` mantenendo invariata la logica. Il metodo `_find_controller_index` viene definito a partire dalla **riga 11** e incapsula la logica di ricerca del dispositivo corretto in base al ruolo. Il ciclo sulle `k_unMaxTrackedDeviceCount` (**riga 13**) copre l’intero spazio degli indici potenzialmente assegnabili da SteamVR; per ciascun indice, verifichiamo innanzitutto la classe del dispositivo con `getTrackedDeviceClass` (**riga 14**). In questo modo filtriamo tutto ciò che non è un controller (ad esempio l’HMD o eventuali tracker addizionali). Solo se la classe corrisponde a `TrackedDeviceClass_Controller` procediamo a verificare il ruolo con `getControllerRoleForTrackedDeviceIndex` (**riga 15**). Quando troviamo un controller il cui ruolo combacia con quello richiesto, restituiamo immediatamente l’indice `i` (**riga 16**). Questa uscita anticipata è voluta: ci interessa il primo match valido per minimizzare latenza e complessità. Se il ciclo termina senza trovare un match, viene emesso un avviso sul logger (**righe 17–18**) per rendere evidente l’anomalia a chi esegue l’esperimento. Infine, si restituisce 0 come valore di ripiego (**riga 19**). Questa scelta “fail-soft” permette al nodo di proseguire comunque, ma va intesa come comportamento conservativo: l’indice 0 potrebbe non corrispondere a un controller valido nel setup specifico. Per questo motivo, nel capitolo dedicato ai test indichiamo chiaramente che, in mancanza del dispositivo atteso, la pubblicazione dei topic `vr/pose` e `vr/trigger` deve essere considerata non affidabile fino al ripristino del collegamento.

## 2.2.2 Codice d'esempio: Raccolta dati

```

1 #Usando anche il codice precedente
2
3 def getPosition():
4     # Get pose
5     poses = self.vrsystem.getDeviceToAbsoluteTrackingPose(
6         openvr.TrackingUniverseSeated,
7         0,
8         openvr.k_unMaxTrackedDeviceCount)
9
10    pose = poses[self.controller_index]
11
12    result, state = self.vrsystem.getControllerState(self.controller_index)
13
14    if result:
15        # di solito l'asse 1 è il grilletto e 2 il grip
16        trigger_value = state.rAxis[2].x
17        front_trigger_value = state.rAxis[1].x
18    else:
19        #self.get_logger().warn(f'{state} qui {result}')
20        trigger_value = 0.0
21        front_trigger_value = 0.0
22
23    return pose, trigger_value, front_trigger_value

```

Il frammento definisce la funzione `getPosition`, utilizzata dal **nodo ROS2 Publisher** per interrogare in tempo reale OpenVR e recuperare (i) la *pose* del controller selezionato e (ii) i valori analogici dei grilletti. Questo passaggio è il primo anello della catena *OpenVR → ROS2*: i dati grezzi letti qui vengono poi pubblicati sui topic `vr/pose` e `vr/trigger` e, successivamente, consumati dal nodo di elaborazione che calcola la cinematica inversa. Senza questa funzione, il nodo non avrebbe accesso aggiornato allo stato del controller e non potrebbe alimentare il resto della pipeline (IK e comando verso Isaac Sim). Il codice parte alla **riga 3** con la definizione della funzione `getPosition`, che ha l'obiettivo di restituire una tripla composta da pose e valori dei due grilletti. Subito dopo, alle **righe 5–8**, la funzione invoca `getDeviceToAbsoluteTrackingPose`: chiediamo a OpenVR l'insieme delle pose correnti per tutti i dispositivi tracciati, specificando `TrackingUniverseSeated` come riferimento (coordinate relative alla postura “seduta”) e 0 come tempo di predizione, così da ottenere lo stato “al momento” della chiamata. Il risultato è una lista di pose indicizzate per dispositivo. Alla **riga 10** estraiamo la pose del *nostro* controller selezionato in fase di inizializzazione (tramite `self.controller_index`, vedi la sezione precedente). Questa pose è ancora nel sistema di riferimento di OpenVR e contiene posizione e orientamento in forma di matrice  $3 \times 4$ . Alla **riga 12** chiediamo a `VRSystem` lo *stato* del controller con `getControllerState`: la variabile `result` indica se la lettura è andata a buon fine; la struttura `state` contiene gli input analogici/digitali (assi, pulsanti, ecc.). Se `result` è vero

---

(righe 14–19), estraiamo due valori dagli assi analogici rAxis: per convenzione spesso l’asse 1 corrisponde al *grilletto anteriore* e l’asse 2 al *grip*; di conseguenza, nel nostro esempio salviamo rAxis[2].x in trigger\_value e rAxis[1].x in front\_trigger\_value. Questa mappatura è coerente con il setup di laboratorio e viene ribadita nel commento: sottolineiamo che la numerazione può variare tra modelli di controller e profili SteamVR, perciò è buona norma verificare le associazioni sugli specifici dispositivi in uso. Se la lettura fallisce, impostiamo entrambi a 0.0 come fallback conservativo. Infine, alla riga 20 la funzione restituisce la tupla (pose, trigger\_value, front\_trigger\_value) che alimenterà i topic ROS2.

**Allineamento dei sistemi di riferimento (visore → Isaac).** La pose restituita da OpenVR è espressa nel frame del *tracking space* di SteamVR, che **non** coincide necessariamente con il frame mondo di *Isaac Sim*. Questo significa che, se i dati venissero utilizzati direttamente, un movimento dell’utente nel visore potrebbe tradursi in una direzione completamente diversa nel simulatore (ad esempio un “avanti” percepito dall’utente potrebbe diventare un “su” per il robot). Per questo motivo, è il **nodo subscriber** che riceve i dati pubblicati dal nodo VR a farsi carico della trasformazione: subito dopo aver sottoscritto i topic vr/pose e vr/trigger, il nodo converte le posizioni e gli orientamenti tramite una matrice di cambio base fissa (rotazioni e, se necessario, traslazioni) che riallinea il frame di SteamVR a quello di Isaac. Solo a valle di questo passaggio i dati trasformati vengono passati al solver di cinematica inversa. In altre parole, la sequenza è sempre: *il nodo subscriber acquisisce i dati grezzi dai topic, li riallinea al frame del simulatore, li fornisce all’IK solver e infine pubblica i comandi risultanti sul topic /joint\_command*. Questo garantisce che ogni movimento dell’utente venga interpretato correttamente nello spazio del robot simulato.

## 2.3 Utilizzo di ALVR

ALVR (Air Light VR) è un software open-source che agisce come un driver per SteamVR, consentendo di creare un visore virtuale (o *fantoccio*) che viene pilotato da un visore fisico reale, anche se quest'ultimo non è direttamente collegato al PC tramite cavo. Questo approccio permette di sfruttare la connessione di rete (Wi-Fi o Ethernet) per trasmettere dati di tracciamento e flussi video tra il visore e il sistema host.

Nel contesto di questo progetto, ALVR è stato impiegato per simulare un visore compatibile con SteamVR utilizzando un dispositivo reale non nativamente supportato, garantendo così la possibilità di accedere alle API di OpenVR. In questo modo è stato possibile:

- intercettare le pose (posizione e orientamento) del visore e dei controller in tempo reale;
- acquisire lo stato dei pulsanti e dei trigger dei controller;
- garantire la compatibilità con l'infrastruttura di raccolta dati sviluppata in Python.

Durante le sessioni di acquisizione, il visore reale inviava continuamente ad ALVR i dati di tracciamento, che venivano poi inoltrati a SteamVR come se provenissero da un visore collegato fisicamente. Questo processo ha permesso di testare e registrare i dati anche in situazioni dove non era possibile utilizzare direttamente un visore supportato da SteamVR, mantenendo però un'elevata fedeltà nel tracciamento e minimizzando la latenza.

L'utilizzo di ALVR si è quindi rivelato fondamentale per garantire flessibilità nell'hardware utilizzato e per consentire la raccolta di dati sperimentali indipendentemente dalle limitazioni di compatibilità del sistema.<sup>1</sup>

---

<sup>1</sup>Nel caso di SteamVR su Windows, il visore utilizzato in questo progetto è pienamente supportato e non richiede ALVR. L'uso di ALVR è stato necessario solo su SteamVR per Linux, dove il visore non è nativamente compatibile.

---

## 2.4 Calcolo del orientamento della mano

OpenVR, per ogni dispositivo tracciato (controller, visore, tracker, ecc.), fornisce la posa del device sotto forma di una matrice  $3 \times 4$  detta `mDeviceToAbsoluteTracking`. Questa matrice descrive contemporaneamente la posizione e l'orientamento del dispositivo rispetto al sistema di riferimento assoluto di OpenVR.

### 2.4.1 Struttura della matrice

La matrice ha la forma:

$$M = \begin{bmatrix} R_{00} & R_{01} & R_{02} & t_x \\ R_{10} & R_{11} & R_{12} & t_y \\ R_{20} & R_{21} & R_{22} & t_z \end{bmatrix}$$

dove:

- $R_{3 \times 3}$  è la matrice di rotazione che rappresenta l'orientamento del device;
- $t = (t_x, t_y, t_z)^T$  è il vettore traslazione che rappresenta la posizione del device nello spazio VR.

Se la estendiamo alla forma omogenea classica ( $4 \times 4$ ):

$$T = \begin{bmatrix} R_{3 \times 3} & t_{3 \times 1} \\ 0 & 1 \end{bmatrix}$$

### 2.4.2 Significato geometrico

Sono coinvolti due sistemi di riferimento:

- **Frame VR (mondo VR)**: è il sistema assoluto della stanza, definito da SteamVR. L'origine si trova in un punto scelto dal setup, e gli assi seguono la convenzione OpenVR. TrackingUniverseSeated ad esempio mette come punto assoluto il centro dell'area di gioco
- **Frame Controller**: è un sistema solidale al controller. L'origine è posizionata all'interno del device e gli assi  $X, Y, Z$  ruotano e traslano insieme al controller.

La matrice permette di trasformare un punto espresso nel sistema locale del controller in coordinate del mondo VR. La relazione è:

$$p_{VR} = R \cdot p_{controller} + t$$

dove:

- $p_{controller}$  è un punto espresso nelle coordinate locali del controller (es. la punta del grilletto rispetto al centro);

- $R$  Come sono ruotati gli assi del controller rispetto a quelli assoluti di OpenVR
- $t$  trasla il punto nella posizione assoluta nello spazio VR;
- $p_{VR}$  è la posizione assoluta del punto nel mondo VR.

### 2.4.3 Esempio numerico

Si consideri la seguente matrice di OpenVR:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0.5 \\ 0 & 0 & -1 & 1.0 \\ 0 & 1 & 0 & 1.2 \end{bmatrix}$$

La rotazione corrispondente è:

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$$

che significa:

- l'asse  $X$  del controller rimane  $X$  nel VR,
- l'asse  $Y$  del controller diventa l'asse  $Z$  del VR,
- l'asse  $Z$  del controller diventa  $-Y$  del VR.

La traslazione è:

$$t = \begin{bmatrix} 0.5 \\ 1.0 \\ 1.2 \end{bmatrix}$$

Supponiamo ora di voler trasformare la punta del controller, descritta nel suo sistema locale come:

$$p_{controller} = \begin{bmatrix} 0 \\ 0 \\ 0.1 \end{bmatrix}$$

Applicando la trasformazione:

$$p_{VR} = R \cdot p_{controller} + t$$

$$p_{VR} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 0.1 \end{bmatrix} + \begin{bmatrix} 0.5 \\ 1.0 \\ 1.2 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.9 \\ 1.2 \end{bmatrix}$$

Dunque la punta del controller si trova alle coordinate (0.5, 0.9, 1.2) nel mondo VR.

#### 2.4.4 Trasformazione delle rotazioni da VR a Robot

Il problema principale è che il sistema di riferimento di OpenVR (visore/controller) e quello del robot Isaac non coincidono. Per questo serve una trasformazione per convertire le rotazioni.

**Step 1: Conversione tra basi (VR → Robot)** Si applica un cambio di base tramite la matrice:

$$R_{\text{convert}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad R_{\text{robot}} = R_{\text{convert}} \cdot R_{VR}$$

Interpretazione:

- [1, 0, 0]:  $X_{VR} \mapsto X_{\text{robot}}$
- [0, 0, 1]:  $Z_{VR} \mapsto Y_{\text{robot}}$
- [0, 1, 0]:  $Y_{VR} \mapsto Z_{\text{robot}}$

Quindi gli assi vengono rimappati come:

$$(X_{VR}, Y_{VR}, Z_{VR}) \mapsto (X_{\text{robot}}, Y_{\text{robot}}, Z_{\text{robot}}) = (X_{VR}, Z_{VR}, Y_{VR})$$

**Step 2: Conversione finale in quaternione** Si converte la matrice finale nel sistema del robot in quaternione:

$$q_{\text{robot}} = R.\text{from\_matrix}(R_{\text{final}}).\text{as\_quat}()$$

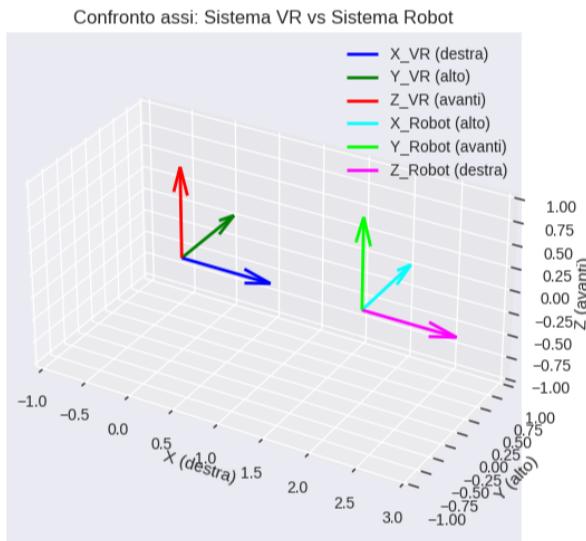


Figure 2.3: Confronto Assi VR e Isaac Sim - Isaac

## 2.5 Cinematica inversa per calcolo dei giunti

Per comprendere il problema della *cinematica inversa* (Inverse Kinematics, IK) partiamo da un esempio concreto. Consideriamo un robot antropomorfo come il *KUKA LBR iiwa*, che possiede 7 giunti rotazionali disposti in maniera analoga a quelli di un braccio umano (spalla, gomito, polso). Ciascun giunto è in grado di ruotare secondo un certo asse, e la combinazione coordinata di queste rotazioni determina la posizione e l'orientamento della mano del robot (l'*end-effector*) nello spazio. In altre parole, muovendo i giunti si sposta l'*end-effector*, ma il rapporto non è immediato: un piccolo cambiamento in un giunto può avere effetti complessi e non intuitivi sulla posizione finale della mano. Ed è proprio qui che entra in gioco la cinematica inversa. Mentre la cinematica diretta risponde alla domanda: “*dato un certo insieme di valori dei giunti, dove si trova l'end-effector?*”, la cinematica inversa affronta il problema opposto: “*dato un punto (o una posa) nello spazio cartesiano che voglio raggiungere, quali valori devono assumere i giunti del robot?*”. Questo problema è fondamentale quando si vuole controllare il robot in modo intuitivo, ragionando in termini di spazio cartesiano (posizioni e orientamenti della mano) invece che in termini di valori dei singoli giunti. Il collegamento con la realtà virtuale è diretto: l'utente che indossa il visore e muove i controller VR non ragiona in termini di giunti del robot, ma in termini di posizioni e orientamenti nello spazio tridimensionale. L'utente sposta la propria mano in avanti, oppure la ruota, e si aspetta che l'*end-effector* del robot compia un movimento analogo, senza preoccuparsi di come devono muoversi le “spalle” o i “gomiti” del robot. Per ottenere questa corrispondenza naturale è necessario un passaggio intermedio: i dati spaziali provenienti dal visore (posizioni e rotazioni dei controller) vengono interpretati come pose desiderate dell'*end-effector*, e successivamente la cinematica inversa calcola i valori di giunto che permettono al robot di assumere esattamente quella configurazione nello spazio. In sintesi, possiamo dire che la cinematica inversa ha il ruolo di “traduttore” tra due mondi: da un lato il mondo della VR, in cui i movimenti sono definiti in coordinate cartesiane, e dall'altro il mondo del robot, in cui i comandi devono essere espressi in termini di angoli di giunto. Grazie a questo disaccoppiamento, l'utente controlla il robot in maniera intuitiva muovendo la propria mano nello spazio virtuale, mentre il sistema si occupa di risolvere il problema matematico di convertire quelle pose cartesiane nei corrispondenti comandi articolari. Questo approccio garantisce che vi sia una mappatura diretta e naturale tra il movimento percepito dall'utente nel visore e il comportamento dell'*end-effector* nel simulatore robotico.

---

### 2.5.1 Cinematica diretta (il problema facile)

Indichiamo con

$$q = [q_1 \ q_2 \ \dots \ q_7]^\top$$

il vettore degli angoli dei giunti del robot.

La *cinematica diretta* (Forward Kinematics, FK) è la funzione che associa a questi valori:

- la **posizione** della mano nello spazio,

$$p(q) = \begin{bmatrix} x(q) \\ y(q) \\ z(q) \end{bmatrix} \in \mathbb{R}^3,$$

- l'**orientamento** della mano, espresso tramite una matrice di rotazione

$$R(q) \in SO(3).$$

In parole semplici: *se conosco quanto ho piegato spalla, gomito e polso ( $q$ ), posso calcolare dove si trova la mano ( $p(q)$ ) e come è orientata ( $R(q)$ ).*

### 2.5.2 Cinematica inversa (il problema difficile)

Il problema inverso è più complesso: data una posa desiderata dell'end-effector,

$$T_d = \begin{bmatrix} R_d & p_d \\ 0 & 1 \end{bmatrix},$$

dove  $p_d = [x_d, y_d, z_d]^\top$  è la posizione desiderata e  $R_d$  l'orientamento desiderato, vogliamo trovare i valori dei giunti  $q$  tali che

$$p(q) \approx p_d, \quad R(q) \approx R_d.$$

Questo è il problema della **cinematica inversa** (IK). Esso è difficile perché:

- possono esistere **più soluzioni** (ci sono vari modi di piegare braccio e gomito per raggiungere lo stesso punto),
- può non esserci **alcuna soluzione** (se il punto richiesto è fuori dallo spazio di lavoro del robot),
- raramente esiste una **formula chiusa** semplice; per questo si usano algoritmi numerici iterativi.

### 2.5.3 Algoritmo numerico iterativo per l'IK

La strategia più diffusa consiste nell'approssimare la soluzione in modo iterativo, migliorandola passo dopo passo:

1. **Configurazione iniziale:** si parte da una configurazione dei giunti  $q_0$  (ad esempio, tutti gli angoli nulli).
2. **Cinematica diretta:** si calcola la posa attuale  $T(q)$  dell'end-effector.
3. **Errore rispetto al target:**

- Errore di posizione:

$$e_p = p_d - p(q) \in \mathbb{R}^3,$$

- Errore di orientamento, **Metodo con logaritmo di matrice:**

$$R_{err} = R_d R(q)^\top, \quad e_o = \text{Log}(R_{err}) \in \mathbb{R}^3.$$

- Errore totale:

$$e = \begin{bmatrix} e_p \\ e_o \end{bmatrix} \in \mathbb{R}^6.$$

4. **Jacobiano**  $J$ : è una matrice che descrive la relazione locale tra variazioni dei giunti  $\Delta q$  e variazioni della posa  $\Delta x$ :

$$\Delta x \approx J(q) \Delta q, \quad J(q) \in \mathbb{R}^{6 \times n}.$$

In altre parole, il Jacobiano risponde alla domanda: *se muovo un po' il giunto i, in che direzione e con quale intensità si sposta o ruota la mano del robot?*

5. **Calcolo della correzione** (metodo di Levenberg–Marquardt / Damped Least Squares): la formula chiusa è:

$$\Delta q = -(J^\top J + \lambda^2 I)^{-1} J^\top e,$$

dove:

- $\Delta q \in \mathbb{R}^n$  è la correzione dei giunti,
- $e \in \mathbb{R}^6$  è l'errore attuale,
- $\lambda^2 I$  è un termine di smorzamento che evita instabilità numeriche e rende la soluzione più robusta vicino alle **singolarità**.

6. **Aggiornamento dei giunti:**

$$q \leftarrow q + \Delta q.$$

7. **Iterazione:** si ripete il procedimento finché l'errore  $\|e\|$  è al di sotto di una soglia prefissata (ad esempio, millimetri per la posizione e gradi per l'orientamento).

---

## 2.6 Il formato URDF

### 2.6.1 Definizione e contesto

Quando si lavora con un robot, sia esso reale o simulato, è indispensabile avere una descrizione precisa della sua struttura meccanica. Non basta sapere che il robot ha un braccio o una pinza: il software deve conoscere esattamente come è fatto, quali parti lo compongono, come queste parti sono collegate tra loro e quali movimenti sono consentiti. Senza questa rappresentazione digitale, né un simulatore né un controllore sarebbero in grado di gestirlo correttamente. Per questo motivo, nei sistemi basati su ROS si utilizza un formato standard chiamato *URDF*, acronimo di *Unified Robot Description Format*. L'URDF non è altro che un file testuale, scritto in XML, che raccoglie in maniera ordinata tutte le informazioni necessarie a descrivere il robot. In questa tesi si assume che il robot di riferimento – il *KUKA LBR iiwa R800* – sia definito proprio tramite un file URDF, che costituisce quindi la base comune su cui costruire tutti gli altri livelli del sistema. Possiamo immaginare l'URDF come una sorta di carta d'identità digitale del robot: senza di esso, i vari strumenti software (ROS2, Isaac Sim, algoritmi di controllo, librerie di cinematica) non avrebbero un linguaggio comune per interpretare la struttura del robot. Con l'URDF, invece, si ottiene una descrizione condivisa, leggibile da diversi programmi, che assicura coerenza e compatibilità tra gli strumenti utilizzati.

### 2.6.2 Contenuti di un file URDF

Un file URDF descrive il robot in tutte le sue parti fondamentali. I principali elementi che lo compongono sono:

- **Link:** rappresentano i corpi rigidi, cioè i pezzi del robot che non si deformano. Possono essere la base fissa, i segmenti del braccio, il polso o l'end-effector.
- **Joint:** sono le giunzioni che collegano i link e che definiscono come questi possono muoversi l'uno rispetto all'altro. Esistono giunti di tipo rotazionale (che permettono una rotazione attorno a un asse), prismatico (che permettono uno scorrimento lineare) oppure fisso (che non consente alcun movimento relativo).
- **Proprietà geometriche e fisiche:** ogni link può essere descritto in termini di forma, dimensioni, massa e parametri di inerzia. Questi dati servono per i calcoli fisici e per simulare in maniera realistica i movimenti.
- **Proprietà visive:** oltre alla fisica, viene descritta anche l'aspetto grafico del robot, tramite primitive geometriche semplici o modelli CAD importati.
- **Limiti dei giunti:** infine, sono specificati i vincoli di movimento, come gli intervalli massimi di rotazione o traslazione, la velocità consentita e la coppia massima applicabile.

In altre parole, l'URDF non dice solo *com'è fatto* il robot, ma anche *come può muoversi e entro quali limiti*.

### 2.6.3 Funzione pratica dell'URDF

La presenza di un file URDF è quindi cruciale per rendere il robot utilizzabile in un ambiente simulato o in un sistema di controllo. A partire dall'URDF, i diversi software sanno:

- quali elementi compongono il robot e come sono collegati;
- dove si trovano i giunti e in che direzione possono muoversi;
- quali vincoli meccanici devono essere rispettati durante i movimenti.

Senza un formato standard come l'URDF, ogni strumento software richiederebbe una propria descrizione personalizzata del robot, con il rischio di dover ripetere più volte lo stesso lavoro e di incorrere in errori o incoerenze. Con l'URDF, invece, si ha un unico file che diventa la fonte di verità per l'intero sistema: la simulazione, i calcoli di cinematica, la pianificazione del movimento e il controllo del robot fanno tutti riferimento alla stessa descrizione. Per questo motivo, l'URDF può essere visto non solo come un file tecnico, ma come un vero e proprio *punto di partenza* imprescindibile per ogni applicazione che coinvolga robotica e simulazione. È grazie a questa rappresentazione che, in questa tesi, è stato possibile integrare in modo coerente la realtà virtuale, gli algoritmi di cinematica inversa e il simulatore Isaac Sim, garantendo che tutti parlassero lo stesso linguaggio: quello del robot descritto in URDF.



# **—3—**

## **Test e Risultati**

CONTENTS: **3.1 Obiettivi del Test.** 3.1.1 Approccio e Strumenti di Test. **3.2 Metodologia di Test.**

**3.3 Risultati e Analisi.** 3.3.1 Qualità del tracking visivo – 3.3.2 Risoluzione e frequenza di aggiornamento – 3.3.3 Rumore e drift dei sensori. **3.4 Analisi visiva.** **3.5 Test di usabilità.**

### **3.1 Obiettivi del Test**

L’obiettivo principale dei test è stato quello di verificare la correttezza sia della *cinematica inversa* implementata, sia dei *dati acquisiti dal visore VR*. In particolare, si è voluto accertare che i *joint* calcolati tramite gli algoritmi di cinematica inversa coincidessero con quelli previsti teoricamente e che i dati di posizione e orientamento provenienti dal visore fossero affidabili, consistenti e privi di perdite significative.

#### **3.1.1 Approccio e Strumenti di Test**

Per la validazione del sistema sono stati adottati due approcci differenti:

##### **1. Approccio statico**

In questa fase i punti sono stati forniti manualmente al nodo responsabile del calcolo della cinematica inversa. L’obiettivo era verificare che l’algoritmo fosse in grado di restituire soluzioni corrette e coerenti con l’input. In questo modo è stato possibile isolare il problema e concentrarsi esclusivamente sulla parte algoritmica, senza introdurre variabili esterne legate al VR.

##### **2. Approccio dinamico**

In questa seconda fase è stato impiegato il visore VR per generare i punti nello spazio in tempo reale. L’algoritmo ha quindi ricevuto i dati acquisiti direttamente dal dispositivo e i risultati sono stati confrontati con le posizioni effettive. In questo

---

modo si è potuto valutare sia l'accuratezza del sistema nel seguire i movimenti, sia l'affidabilità della trasmissione dei dati dal visore al sistema.

#### strumenti utilizzati:

- **Uno script di supporto:** impiegato principalmente nella fase statica, consentiva di fornire in ingresso al sistema una serie di punti predefiniti, permettendo così un controllo puntuale dell'algoritmo.
- **Il visore VR stesso:** fondamentale nella fase dinamica, ha permesso di valutare il comportamento del sistema in condizioni realistiche di utilizzo, simulando movimenti e interazioni naturali dell'utente.

## 3.2 Metodologia di Test

La metodologia adottata ha seguito un approccio *comparativo* tra due modalità differenti, ma basate sullo stesso scenario sperimentale.

Il test consisteva nel posizionare tre sfere in posizioni prefissate all'interno dello spazio di lavoro del braccio robotico. L'obiettivo era verificare che l'end-effector fosse in grado di raggiungere con precisione ciascuna di queste posizioni, utilizzando due modalità di input differenti:

- **Fase statica:** le coordinate delle tre sfere venivano fornite direttamente al sistema tramite uno script dedicato. In questo modo il nodo di calcolo della cinematica inversa riceveva i punti come input e il braccio eseguiva i movimenti, raggiungendo le posizioni prestabilite. Questa modalità ha permesso di validare l'implementazione algoritmica in condizioni controllate.
- **Fase VR:** lo stesso scenario è stato riprodotto in un ambiente virtuale immersivo tramite il visore VR e la piattaforma Isaac. In questo caso i tre punti venivano selezionati dall'utente con il controller VR, e il braccio riceveva in tempo reale le coordinate da raggiungere. Questo ha consentito di valutare il comportamento del sistema in condizioni più realistiche e interattive.

In entrambe le modalità, i risultati ottenuti sono stati *confrontati con le posizioni teoriche* delle sfere, così da verificare la precisione del movimento del braccio e l'affidabilità del flusso di dati provenienti dallo script o dal visore VR.

### 3.3 Risultati e Analisi

Nella figura 3.1 possiamo osservare come gli errori rilevati durante l'utilizzo del visore VR non siano né costanti né nulli. Infatti, mentre nel caso statico (senza VR) l'errore era nullo in tutti i test, con l'impiego del visore si evidenziano discrepanze nell'ordine dei millimetri/centesimi di centimetro.

Questo fenomeno risulta ancora più evidente se ripetiamo più volte lo stesso test. Come riportato nella figura 3.2, gli errori non seguono un andamento regolare ma risultano variabili e apparentemente casuali, rendendo impossibile definire un valore di errore costante o facilmente quantificabile.

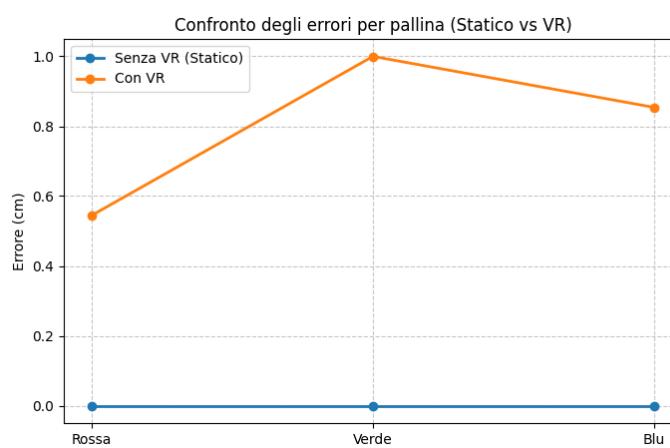


Figure 3.1: Confronto errore in cm - Test singolo

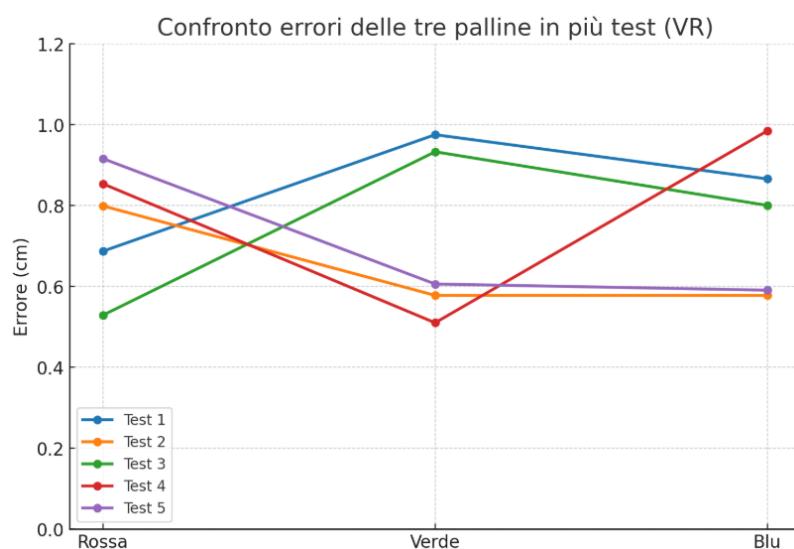


Figure 3.2: Confronto errore in cm su 5 test ripetuti

---

Questa variabilità è imputabile alle caratteristiche del visore *Oculus Quest 2*. Il dispositivo, infatti, utilizza un sistema di *inside-out tracking* basato su telecamere integrate e algoritmi di visione artificiale per stimare posizione e orientamento dei controller nello spazio. A differenza di un sistema di motion capture con marker esterni, la posizione ricostruita dipende fortemente da:

### 3.3.1 Qualità del tracking visivo

Le telecamere del visore osservano l'ambiente circostante per determinare la posizione nello spazio. La precisione del tracciamento dipende fortemente dalle condizioni ambientali:

- **Illuminazione:** condizioni troppo scure o troppo luminose riducono la capacità delle telecamere di rilevare dettagli e riferimenti spaziali, causando errori nella stima della posizione.
- **Texture e dettagli dell'ambiente:** superfici uniformi o specchianti, come pareti bianche o pavimenti lucidi, forniscono pochi punti di riferimento visivi, aumentando l'incertezza nella ricostruzione della posizione.
- **Occlusioni:** oggetti o parti del corpo che bloccano la visuale dei controller possono compromettere il tracciamento visivo.

### 3.3.2 Risoluzione e frequenza di aggiornamento

Il Quest 2 non è progettato per applicazioni metrologiche di alta precisione, ma per la realtà virtuale immersiva. Pertanto, la risoluzione delle telecamere e la frequenza di aggiornamento dei dati limitano la precisione del tracciamento:

- **Risoluzione:** i sensori hanno una capacità limitata nel catturare dettagli infinitesimali della scena.
- **Frequenza di aggiornamento:** durante movimenti rapidi, il sistema deve stimare la posizione dei controller tra un frame e l'altro, generando possibili discrepanze rispetto alla posizione reale.

### 3.3.3 Rumore e drift dei sensori

Il visore combina i dati delle telecamere con quelli di una *IMU* (Inertial Measurement Unit), soggetta a rumore e deriva (*drift*) nel tempo. Gli algoritmi di fusione sensoriale stimano posizione e orientamento, ma possono introdurre errori:

- **Drift:** in assenza di riferimenti visivi chiari, la posizione stimata può discostarsi lentamente dalla reale.

- **Rumore:** piccole oscillazioni dei dati possono manifestarsi come tremolii o vibrazioni dei controlleri virtuali.

In conclusione, gli errori riscontrati non rappresentano malfunzionamenti del sistema, ma sono una conseguenza naturale dei limiti tecnologici del visore VR. Essi rientrano comunque in un ordine di grandezza accettabile (tra 0.5 cm e 1 cm) per applicazioni di simulazione e interazione virtuale, ma vanno tenuti in considerazione se si intende utilizzare il sistema in contesti che richiedono elevata precisione.

### 3.4 Analisi visiva

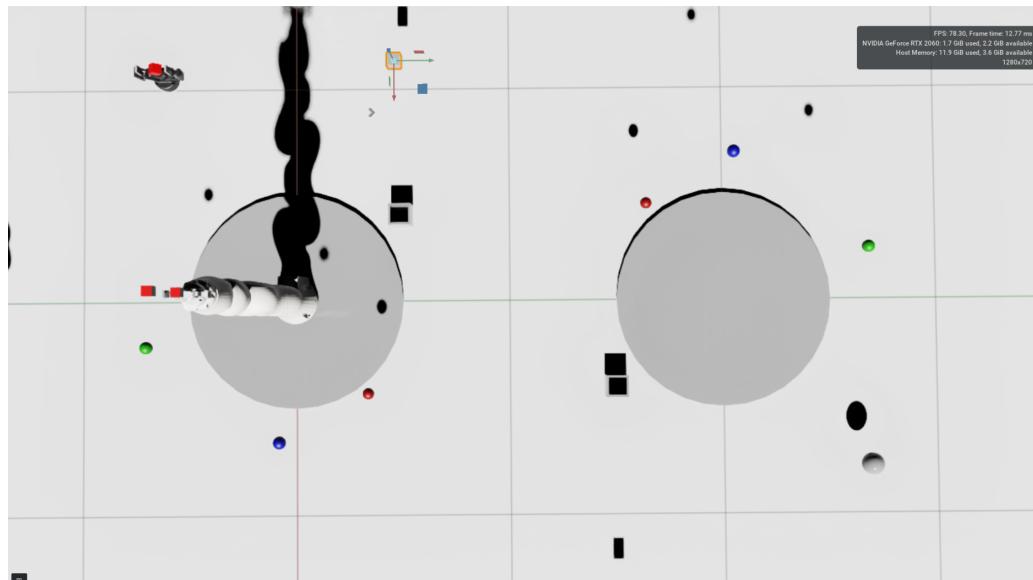


Figure 3.3: Vista dall’alto dell’ambiente di test utilizzato per la validazione del sistema.

La figura 3.3 mostra l’ambiente di test predisposto per la validazione del sistema di interazione tra visore VR e robot. L’area di lavoro è stata progettata in modo da simulare un contesto semplice ma funzionale, capace di mettere alla prova la pipeline sviluppata. Al centro della scena sono presenti due pedane circolari: su una di esse viene collocato l’utente virtuale, che attraverso il visore può muoversi e impartire comandi, mentre sull’altra si trova il braccio robotico incaricato di eseguire le azioni. Attorno alle pedane sono stati distribuiti diversi oggetti con cui interagire, scelti per la loro semplicità geometrica e per la chiarezza nella rappresentazione dei task. In particolare, si distinguono delle sfere colorate, che fungono da punti obiettivo da raggiungere o manipolare, e una scatola che può essere utilizzata come contenitore in cui depositare le palline. La disposizione degli elementi non è casuale: le sfere sono collocate a diverse altezze e distanze, in modo da richiedere movimenti differenti al robot e verificare la capacità del sistema di adattarsi a situazioni variabili. Questa configurazione consente di ricreare scenari tipici di interazione

---

uomo–robot, in cui l’utente indica un obiettivo nello spazio virtuale e il braccio robotico si muove di conseguenza, traducendo il comando in una sequenza di movimenti articolari. L’ambiente di test, pur essendo essenziale, si dimostra quindi efficace per valutare sia la precisione della cinematica inversa sia la naturalezza dell’interazione mediata dal visore VR. La presenza di oggetti semplici ma funzionali rende chiaro il comportamento del sistema e facilita l’analisi qualitativa dei risultati.



Figure 3.4: Confronto tra la posa reale (sopra) e la corrispondente rappresentazione in VR (sotto).

La figura 3.4 mostra un confronto diretto tra il movimento reale dell’utente, ripreso mentre indossa il visore VR, e la corrispondente rappresentazione all’interno dell’ambiente simulato. Nella parte superiore si osserva la posa effettivamente assunta dall’utente, che estende il braccio in avanti mantenendo il controller nella mano destra. Nella parte inferiore, invece, è riportata la scena virtuale generata in Isaac Sim, in cui la stessa postura viene riprodotta dal sistema di tracking. Questo confronto è particolarmente significativo perché evidenzia la coerenza tra il movimento reale e la sua proiezione nello spazio virtuale.

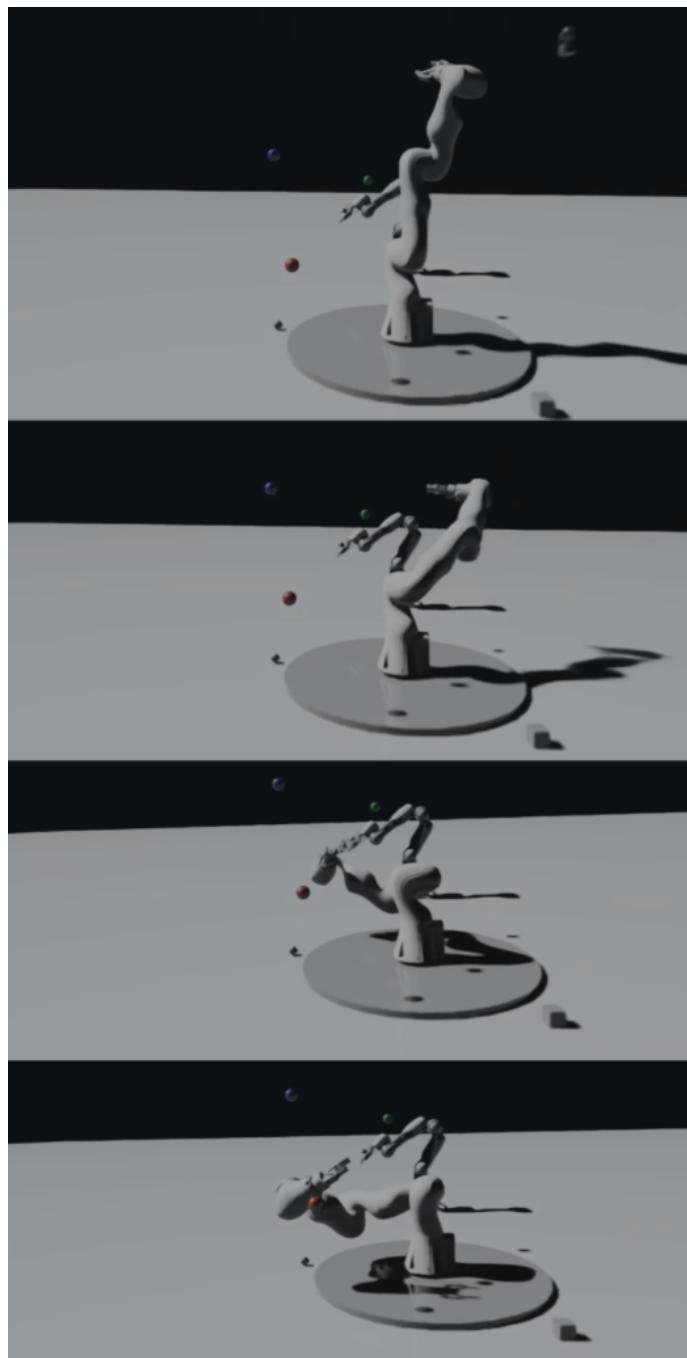


Figure 3.5: Sequenza di frame che mostra il movimento del braccio robotico verso la posizione indicata dall'utente in VR.

La figura 3.5 illustra una sequenza di fotogrammi che rappresenta il movimento del braccio robotico durante l'esecuzione di un comando generato dall'utente attraverso il visore VR. Nella parte iniziale della sequenza si osserva il robot in una configurazione di partenza, con l'end-effector distante dal punto da raggiungere. Man mano che i frame avanzano, il

---

braccio modifica progressivamente la sua configurazione articolare, seguendo le traiettorie calcolate dall'algoritmo di cinematica inversa e avvicinandosi gradualmente all'obiettivo. Il punto di arrivo, evidenziato nella scena dalla sfera colorata, rappresenta la posizione nello spazio selezionata dall'utente all'interno dell'ambiente virtuale. Il movimento del robot testimonia come il sistema sia in grado di tradurre in tempo reale l'input proveniente dal visore in una sequenza di comandi articolari coerenti e fluidi, rispettando i vincoli cinematici e i limiti meccanici del modello. Questa dimostrazione visiva permette di apprezzare non solo l'accuratezza della cinematica inversa, ma anche la naturalezza con cui l'interazione VR viene trasferita alla simulazione robotica. La corrispondenza tra il gesto dell'utente e il successivo spostamento del robot rafforza l'efficacia dell'approccio integrato, confermando la possibilità di controllare un sistema complesso come un braccio antropomorfo attraverso movimenti intuitivi e immediati.

### 3.5 Test di usabilità

Per completare la fase di validazione non ci si è limitati alla sola correttezza dei calcoli e alla precisione dei dati, ma si è deciso di valutare anche l'*usabilità* del sistema dal punto di vista dell'utente. L'obiettivo di questa parte del test era misurare quanto tempo fosse necessario a diversi soggetti per portare a termine un compito semplice ma significativo: prendere una pallina e riporla all'interno di una scatola utilizzando il braccio robotico controllato attraverso il *visore VR*. Il gruppo di prova era composto da cinque persone con livelli di esperienza differenti. Il primo soggetto era lo sviluppatore del sistema, quindi già pienamente familiare con il funzionamento sia del *visore* che dell'architettura robotica. Altri due soggetti erano utenti che avevano già utilizzato in passato dispositivi di realtà virtuale, ma che non conoscevano nel dettaglio il funzionamento del mio sistema. Infine, gli ultimi due soggetti non avevano mai utilizzato un *visore VR* né avevano conoscenze di robotica; per loro è stato quindi necessario un breve *training* iniziale in cui veniva spiegato come indossare il visore, come muovere i controller e come interagire con l'ambiente virtuale. Il compito da svolgere era uguale per tutti: afferrare con il braccio robotico una pallina posizionata all'interno dell'ambiente di test e depositarla all'interno di una scatola. L'unico parametro misurato era il *tempo* necessario a completare l'operazione, dall'istante in cui l'utente prendeva il controllo fino al momento in cui la pallina veniva effettivamente rilasciata nella scatola. Lo sviluppatore ha impiegato un *tempo di completamento* di circa 40 secondi, risultato prevedibilmente inferiore rispetto agli altri partecipanti, in quanto già abituato a utilizzare sia il *visore* che l'interfaccia di controllo del robot. I due utenti che avevano familiarità con i *visori VR* ma non conoscevano il sistema hanno impiegato un tempo leggermente superiore, dimostrando comunque una buona capacità di adattamento: pur non conoscendo la logica interna della simulazione, la loro esperienza con la realtà virtuale ha consentito loro di orientarsi rapidamente e di portare a termine il compito senza grosse difficoltà. Diversa è stata la situazione per i due *principianti*, che non avevano mai utilizzato un *visore VR*. Dopo la fase di *training* sono riusciti a completare il *task*, ma con tempi sensibilmente più alti. Nelle prime prove hanno mostrato incertezza soprattutto nella

coordinazione dei movimenti, dovendo abituarsi alla sensazione di controllare un oggetto virtuale attraverso i controller. Tuttavia, una volta superata la fase iniziale di adattamento, sono stati in grado di portare a termine il compito in modo corretto, seppur con tempi più lunghi rispetto agli altri soggetti. La figura 3.6 riassume in modo chiaro questi risultati, mostrando i *tempi medi* di completamento del compito per ciascun utente. I valori riportati non derivano da una singola esecuzione, ma rappresentano la *media calcolata su tre tentativi*, in modo da ridurre l'influenza di eventuali errori isolati o condizioni particolari. Dal grafico si nota immediatamente come l'utente sviluppatore abbia tempi significativamente inferiori, attestandosi sui 40 secondi, mentre gli utenti già abituati al *visore VR* si posizionano in una fascia intermedia, con tempi medi intorno al minuto. I *principianti*, invece, richiedono tempi decisamente più lunghi, vicini o superiori al minuto e mezzo, a testimonianza della maggiore difficoltà iniziale nel prendere confidenza con la modalità di interazione proposta. Nel complesso, i risultati mostrano una chiara distinzione tra i diversi livelli di esperienza. L'utente sviluppatore ha completato l'attività in minor tempo, i due utenti con esperienza di *VR* si sono posizionati in una fascia intermedia, mentre i *principianti* hanno richiesto più tempo ma sono comunque riusciti a concludere il compito. Questo evidenzia che il sistema è utilizzabile anche da chi non ha alcuna esperienza, purché venga fornito un breve *addestramento* iniziale, e che la curva di apprendimento non è particolarmente ripida. In altre parole, l'interazione proposta risulta *naturale* e *intuitiva*, anche se l'efficienza cresce sensibilmente con la familiarità maturata nell'uso del *visore* e con la pratica.

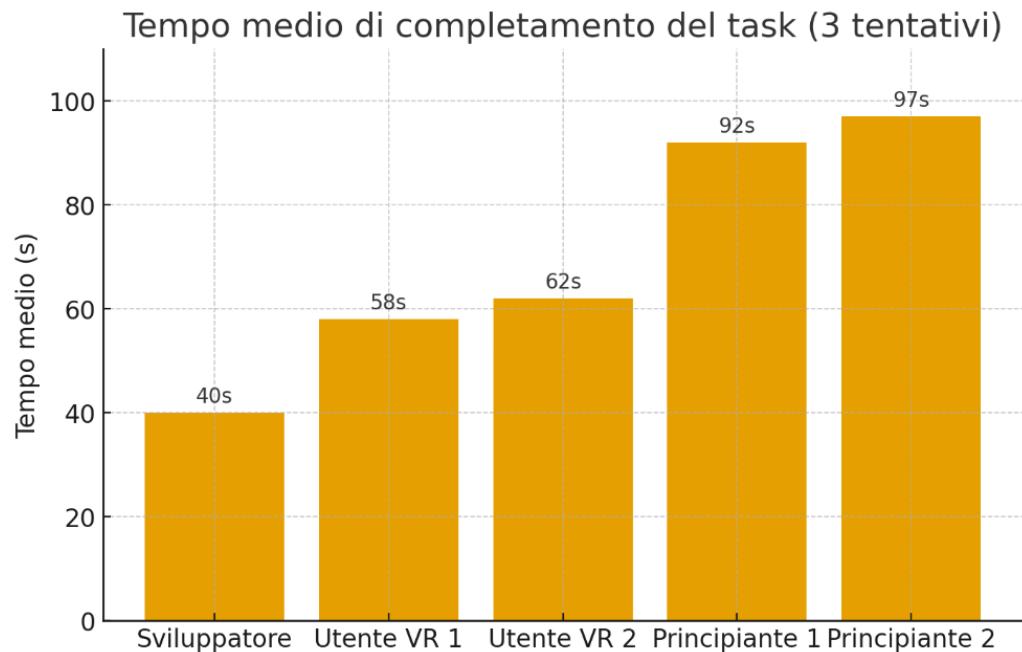


Figure 3.6: Tempi medi di completamento del compito di manipolazione (media su tre tentativi per ciascun utente).



# **-4-**

## **Conclusioni**

CONTENTS: **4.1 Discussione**. **4.2 Sviluppi futuri**.

### **4.1 Discussione**

Il lavoro di tesi ha dimostrato la fattibilità dell'integrazione tra un visore di realtà virtuale e un braccio robotico tramite un'infrastruttura software basata su **ROS2**, **Isaac Sim** e librerie dedicate come **OpenVR** e **ALVR**. L'obiettivo principale consisteva nell'acquisizione dei dati di posizione e orientamento dal visore VR e nella loro conversione in comandi di controllo per il robot. Tale obiettivo è stato raggiunto, ottenendo un sistema modulare e scalabile, capace di collegare il movimento dell'utente in realtà virtuale con il movimento del robot nello spazio simulato. I test sperimentali hanno evidenziato un livello di affidabilità soddisfacente: l'errore medio di tracciamento si è mantenuto nell'ordine del mezzo centimetro o, al massimo, di un centimetro. Questo grado di precisione può essere considerato adeguato per scenari di simulazione e di interazione virtuale, anche se non sufficiente per applicazioni robotiche di tipo industriale ad alta precisione. Allo stesso tempo, sono emersi alcuni limiti tecnici. Il primo riguarda i sensori del visore VR, che non sono progettati per compiti di misura metrologica. Ne derivano inevitabilmente rumore, deriva (*drift*) e una risoluzione limitata. Un secondo limite riguarda la gestione dei sistemi di riferimento: il frame di SteamVR non coincide con il frame mondo di Isaac Sim, e ciò ha reso necessarie trasformazioni aggiuntive di riallineamento per evitare incoerenze nei movimenti. Nonostante queste criticità, l'approccio adottato ha permesso di ottenere un'infrastruttura coerente e riutilizzabile, che può fungere da base per scenari più complessi o per applicazioni su robot reali.

---

## Approccio al problema

L'intero lavoro è stato sviluppato secondo una metodologia di tipo *bottom-up*, che ha consentito di affrontare separatamente i diversi aspetti del sistema e ridurne la complessità complessiva. Il punto di partenza è stato lo studio preliminare degli strumenti necessari: *Isaac Sim* come ambiente di simulazione, *ROS2* come piattaforma di comunicazione, *OpenVR* e *ALVR* per l'interfacciamento con il visore. Questo studio iniziale ha permesso di delineare con chiarezza quali funzioni fossero disponibili e come potessero essere integrate nel progetto. In un secondo momento, il problema complessivo è stato scomposto in sottoproblemi più gestibili. La prima sfida riguardava l'acquisizione dei dati provenienti dal visore VR e la loro trasmissione attraverso nodi ROS2. Una volta consolidato questo passaggio, è stata affrontata la questione dei sistemi di riferimento: per garantire che un movimento della mano nel mondo virtuale corrispondesse a un movimento coerente nel simulatore, è stato necessario introdurre trasformazioni di coordinate che riallineassero gli assi del visore con quelli di Isaac Sim. Superata questa fase, è stato possibile integrare la cinematica inversa. Grazie a una libreria dedicata, i dati cartesiani acquisiti dal visore sono stati tradotti in configurazioni articolari del braccio robotico. Questo passaggio è stato essenziale per disaccoppiare il controllo dallo spazio dei giunti e rendere più intuitivo il collegamento tra l'utente in VR e l'end-effector del robot. Infine, l'intero sistema è stato validato attraverso sessioni di test, con l'obiettivo di verificare stabilità, affidabilità e coerenza dei movimenti. L'approccio modulare e progressivo ha garantito una maggiore robustezza e ha consentito di isolare rapidamente eventuali problematiche emerse.

## Difficoltà incontrate

Il percorso non è stato privo di ostacoli. Una delle principali difficoltà è stata la gestione dei sistemi di riferimento: la differenza tra gli assi del visore VR e quelli del simulatore ha imposto un'analisi dettagliata delle trasformazioni di coordinate, senza la quale i dati acquisiti sarebbero stati incoerenti. Un'ulteriore complessità è stata l'implementazione della cinematica inversa. Pur disponendo di librerie dedicate, è stato necessario selezionare configurazioni che garantissero soluzioni stabili e con errori ridotti. Anche l'integrazione del *ROS2 Bridge* in Isaac Sim ha presentato alcune difficoltà, richiedendo un lavoro di configurazione non banale per stabilire una comunicazione affidabile. Dal punto di vista pratico, un problema significativo ha riguardato l'allineamento dell'orientamento della mano. Anche in questo caso, il disallineamento tra frame di riferimento ha reso necessario un trattamento specifico per ottenere una corrispondenza credibile dei movimenti. Infine, non va trascurata la fase di reperimento dei file URDF: trovare descrizioni accurate del robot in formato standard non si è rivelato immediato, poiché i file disponibili non erano sempre coerenti o completi. Nel complesso, queste difficoltà hanno rappresentato sfide formative e hanno contribuito a consolidare l'approccio metodologico seguito.

## 4.2 Sviluppi futuri

Il lavoro apre la strada a possibili sviluppi. Un primo ambito di miglioramento riguarda la precisione: l'adozione di filtri più sofisticati, come i filtri di Kalman, potrebbe ridurre in maniera significativa rumore e deriva dei sensori. Un secondo passo naturale consiste nell'integrazione con un robot reale, trasferendo la pipeline dal contesto simulato a quello fisico. Questo consentirebbe di validare l'approccio in scenari applicativi concreti, aprendo la possibilità di utilizzare la realtà virtuale come interfaccia naturale per il controllo diretto.



# Bibliography

- [1] Quigley M., Gerkey B., Smart W.D., “Programming Robots with ROS: A Practical Introduction to the Robot Operating System”, O’Reilly Media, 2015.
- [2] ROS 2 Documentation, <https://docs.ros.org/en/>, consultato nel 2024–2025.
- [3] NVIDIA, “Isaac Sim: Robotics Simulation and Synthetic Data Generation”, <https://developer.nvidia.com/isaac-sim>, consultato nel 2024–2025.
- [4] Valve Corporation, “OpenVR SDK”, <https://github.com/ValveSoftware/openvr>, consultato nel 2024–2025.
- [5] Valve Corporation, “SteamVR Developer Documentation”, <https://developer.valvesoftware.com/wiki/SteamVR>, consultato nel 2024–2025.
- [6] ALVR Project, “Air Light VR (ALVR)”, <https://github.com/alvr-org/ALVR>, consultato nel 2024–2025.
- [7] ROS Wiki, “URDF (Unified Robot Description Format)”, <http://wiki.ros.org/urdf>, consultato nel 2024–2025.
- [8] KUKA Robotics, “KUKA LBR iiwa R800 – Technical Data Sheet”, Documentazione tecnica ufficiale, 2020.
- [9] Craig J.J., “Introduction to Robotics: Mechanics and Control”, 4th Edition, Pearson, 2018.
- [10] Sciavicco L., Siciliano B., “Modelling and Control of Robot Manipulators”, Springer, 2nd Edition, 2000.
- [11] Siciliano B., Sciavicco L., Villani L., Oriolo G., “Robotics: Modelling, Planning and Control”, Springer, 2009.
- [12] LaValle S.M., “Virtual Reality”, Cambridge University Press, 2017.
- [13] Meta (Oculus), “Oculus Quest 2 Specifications and Developer Documentation”, <https://developer.oculus.com/quest/>, consultato nel 2024–2025.

- 
- [14] INVERSE Project, “INteractive VR Environment for Robotic Simulation and Education”, <https://www.inverse-project.org>, consultato nel 2024–2025.
  - [15] Tutorial online: integrazione di Isaac Sim con ROS2, materiale didattico e guide reperite dalla community NVIDIA e ROS, 2023–2024.
  - [16] Unity Technologies, “VR Development Documentation”, <https://docs.unity3d.com/Manual/VR.html>, consultato nel 2024. Epic Games, “Unreal Engine VR Development”, <https://docs.unrealengine.com/>, consultato nel 2024.