Università Politecnica delle Marche

Ingegneria Informatica e dell'Automazione

RoboSimCloth

Corso di

Laboratorio di Automazione

Anno accademico 2024-2025

Studenti:
Pizzuto Andrea
Meloccaro Lorenzo
Percipalle Noemi

Professore:
Andrea Bonci

Dottorandi:
Serafini Andrea
Pellicani Ilaria
Di Biase Alessandro

Dipartimento di Ingegneria dell'Informazione

Indice

1 Introduzione

Il capitolo fornisce una panoramica del progetto, descrivendo l'obiettivo principale di sviluppare e simulare la movimentazione di un robot TM-900 con un tessuto, attraverso l'integrazione di ROS2 e Unity. Viene fornita una panoramica delle motivazioni per l'uso di questi software e delle principali fasi del progetto.

Il presente progetto ha come obiettivo lo sviluppo e la simulazione della manipolazione dei tessuti con il robot collaborativo TM5-900, utilizzando l'integrazione di software avanzati come ROS2 e Unity. Il contesto applicativo del progetto si colloca nell'ambito industriale e robotico, ponendo particolare attenzione sull'ottimizzazione dei processi di smantellamento di oggetti deformabili, come abiti e tessuti. L'azienda coinvolta utilizza il software Cloth 3D per la modellazione degli abiti, mentre Unity e Gazebo sono impiegati per simulare la movimentazione del robot e l'interazione con gli oggetti. Uno degli obiettivi di questo progetto è quello di analizzare in dettaglio l'integrazione tra ROS2 e Unity, scelta motivata dalle potenzialità di questi due strumenti. ROS2 offre una gestione efficiente della comunicazione tra il robot e il sistema di simulazione, mentre Unity fornisce la possibilità di simulare l'effetto della gravità sui tessuti e i movimenti del robot in tempo reale tramite una grafica avanzata. L'integrazione di questi due software è stata scelta per soddisfare la necessità di creare un ambiente simulativo realistico, utile non solo per eseguire test simulativi, ma anche per implementare successivamente il sistema su un robot reale. In questo modo, l'integrazione tra ROS2 e Unity consente di gestire il robot e simularlo in ambiente virtuale, migliorando l'efficienza e la precisione nelle operazioni robotiche. Il progetto affronta diversi task, tra cui la valutazione e la compatibilità dei tessuti con i vari software, la simulazione della movimentazione del robot e dell'oggetto da disassemblare in un ambiente condiviso, l'acquisizione e la definizione di una sequenza di fasi di smantellamento, l'esecuzione delle fasi in simulazione con il robot e, infine, l'esecuzione delle stesse fasi su un robot reale.

2 Materiali e Metodi

In questo capitolo saranno presentati i materiali(hardware e software) e i metodi, quindi le tecniche utilizzate, inoltre vengono dettagliate le tecnologie impiegate e le procedure seguite, in modo da permettere la riproducibilità del progetto. In particolare la prima sezione, ***Hardware*** *comprende la descrizione del robot utilizzato in simulazione Omron TM5-900 e della camera RealSense D435i. Successivamente saranno presentati i vari* ***Software****, utilizzati all'interno del progetto e che hanno permesso la riuscita dell'applicazione, tra i quali Ubuntu, ROS2, Moveit, Gazebo, Unity e in particolare lo strumento Cloth di Unity. Infine, verranno descritte nel dettaglio la* ***Predisposizione dei Software nel PC*** *e le modalità di* ***Interfacciamento tra ROS 2 e Unity****, con focus su strumenti fondamentali come ROS-TCP Connector, ROS-TCP Endpoint e URDF Importer.*

2.1 Hardware

In questa sezione verranno presentati i principali componenti hardware utilizzati nel progetto, in particolare il robot Omron TM5-900 e la Stereo-Camera RealSense D435i. Saranno forniti dettagli tecnici e funzionali su ciascun componente, evidenziando le loro caratteristiche principali e il loro ruolo all'interno del sistema di simulazione.

2.1.1 Omron TM5-900

Un robot è un manipolatore riprogrammabile e multifunzionale progettato per spostare materiali, parti, strumenti o dispositivi specializzati attraverso vari movimenti programmati per l'esecuzione di una varietà di compiti. Il robot **OMRON TM5-900** è un robot industriale collaborativo (cobot) di ultima generazione, progettato per applicazioni che richiedono flessibilità, precisione e la capacità di operare in ambienti dinamici e collaborativi. È dotato di una serie di caratteristiche avanzate che lo rendono adatto a una varietà di compiti in ambienti industriali, inclusi il montaggio, il pick-and-place, e operazioni di smontaggio come quelle previste in questo progetto.

Tra le caratteristiche principali vi sono:

- **Design compatto e flessibile**: Il TM5-900 è progettato per lavorare in spazi ristretti, con un braccio robotico che può essere facilmente integrato in linee di produzione esistenti senza necessitare di ampi spazi operativi. La sua flessibilità gli consente di essere facilmente usato per diverse applicazioni.
- **Portata e capacità di carico**: Il TM5-900 ha una capacità di carico utile fino a 5 kg, che lo rende adatto a manipolare una varietà di oggetti e strumenti, come i tessuti utilizzati in questo progetto.
- **Gradi di libertà (DOF)**: Il TM5-900 è dotato di 6 gradi di libertà (DOF), che gli consentono di eseguire movimenti complessi e articolati, simili a quelli di un braccio umano.

Figure 2.1: *Omron TM5-900*

Questo permette al robot di raggiungere e manipolare oggetti in spazi tridimensionali con alta precisione.

- **Giunti e movimenti**: Il robot è equipaggiato con giunti motorizzati, che consentono di controllare i movimenti di rotazione in modo fluido e preciso. I giunti, insieme ai sensori di forza e coppia, permettono di eseguire operazioni delicate, come il manipolamento di oggetti morbidi (ad esempio i tessuti) senza danneggiarli.
- **Facilità di programmazione e utilizzo**: Questo robot è dotato di un'interfaccia utente intuitiva e di software di programmazione che ne facilita l'uso anche da parte di personale non specializzato. La programmazione è resa semplice grazie alla modalità di "teaching", che consente di insegnare al robot i movimenti direttamente sul campo.
- **Tecnologia di sensori avanzati**: Il TM5-900 è equipaggiato con sensori di forza e coppia, che permettono di eseguire operazioni delicate con una precisione millimetrica, ideale per manipolare oggetti senza danneggiarli.
- **Compatibilità con ROS2**: Il robot è compatibile con il sistema ROS2, che consente di integrarlo facilmente in ambienti di simulazione come Gazebo e Unity, come previsto nel progetto. Questo permette di controllare il robot tramite comandi remoti e di simulare il suo comportamento in ambienti virtuali.

In questo progetto, il robot OMRON TM5-900 è stato scelto per la sua versatilità e capacità di manipolare oggetti come tessuti in modo preciso e sicuro. Il suo utilizzo si concentra sulla simulazione della movimentazione del robot, dove l'integrazione con ROS2

assicura una gestione ottimale della comunicazione e delle operazioni. Inoltre, l'interazione con Unity permette di visualizzare in tempo reale il comportamento del robot, monitorando e ottimizzando il suo funzionamento durante le diverse fasi del progetto.

2.1.2 Stereo-Camera RealSense D435i

La camera Intel RealSense D435i è un dispositivo avanzato per la rilevazione della profondità, dotato di sensore IMU integrato che consente l'acquisizione simultanea di dati visivi e inerziali. Grazie alla combinazione di una coppia di sensori stereo e di una fotocamera RGB, la D435i è ampiamente utilizzata in applicazioni di robotica, visione artificiale, mappatura 3D e realtà aumentata.

2.2 Software

In questo capitolo verranno analizzati i software principali utilizzati per il progetto: Ubuntu, Robot Operating System 2, Moeveit, Gazebo e Unity. Verranno fornite una panoramica generale, informazioni sul loro utilizzo e sulle versioni adottate. Inoltre, si approfondirà l'uso del componente Cloth di Unity, che ha rivestito un ruolo centrale nella simulazione della fisica dei tessuti.

2.2.1 Ubuntu

Figure 2.2: *Logo Ubuntu*

Per lo sviluppo del progetto è stato utilizzato il sistema operativo **Ubuntu 22.04.5**, una distribuzione Linux molto diffusa nel campo della robotica grazie alla sua ampia compatibilità con **ROS2 (Robot Operating System 2)**.

L'installazione di Ubuntu è avvenuta tramite WSL (Windows Subsystem for Linux), una funzionalità disponibile su Windows 10 e 11 che consente di eseguire nativamente un ambiente GNU/Linux all'interno di Windows, senza dover ricorrere a una macchina virtuale o a un sistema dual boot. In particolare, è stata utilizzata la versione WSL2, che offre prestazioni migliorate rispetto a WSL1 grazie all'integrazione di un vero e proprio kernel Linux. Questa

soluzione permette di combinare la potenza e la flessibilità di Ubuntu con la praticità degli strumenti di sviluppo offerti dall'ambiente Windows.

Di seguito, i principali passaggi eseguiti per l'installazione: [1]

1. Abilitazione delle funzionalità WSL e Virtual Machine Platform:
   - Aprire il menu Start e cercare *Windows PowerShell*.
   - Avviare PowerShell come amministratore.
   - Digitare il comando:

     ```
     wsl --install
     ```

     Questo comando installa WSL2 come versione predefinita. Al termine dell'installazione, sarà necessario riavviare il computer.
2. Installazione di Ubuntu 22.04
   - Dopo aver abilitato WSL2, è possibile installare Ubuntu digitando:

     ```
     wsl --install -d Ubuntu-22.04
     ```

   Al termine dell'installazione, sarà necessario riavviare il computer.
3. Configurazione iniziale:
   - Al primo avvio, verrà richiesto di creare un utente e una password Linux.
   - Infine, è consigliato aggiornare i pacchetti eseguendo:

     ```
     sudo apt update && sudo apt upgrade
     ```

Questa configurazione ha permesso l'installazione di ROS2 e l'integrazione con librerie e strumenti specifici per la robotica, mantenendo una buona compatibilità con gli altri software utilizzati nel progetto, come Unity e Gazebo.

2.2.2 Robot Operating System 2

Figure 2.3: *Logo ROS2*

ROS 2 (Robot Operating System 2) [2] è un framework open-source per lo sviluppo di applicazioni robotiche. Fornisce una serie di strumenti, librerie e convenzioni per facilitare la comunicazione tra i componenti software di un sistema robotico. ROS 2 è progettato per migliorare la scalabilità, la sicurezza e la compatibilità con i sistemi distribuiti rispetto al suo predecessore, ROS 1. Il suo utilizzo è diffuso in ricerca e industria per il controllo di robot mobili, bracci robotici e veicoli autonomi. In questo progetto, ROS 2 è stato utilizzato per gestire la comunicazione tra il robot simulato in Unity e i componenti di controllo. ROS 2 permette lo scambio di messaggi tra i nodi, abilitando il controllo del robot da Unity e viceversa. In particolare, il framework ha permesso l'integrazione con il sistema di simulazione, la gestione delle traiettorie e il controllo dei giunti del manipolatore. Inizialmente, il progetto è stato avviato con ROS 2 Jazzy, ma si sono riscontrati diversi problemi di compatibilità e instabilità, in particolare con l'integrazione del ROS-TCP Connector per Unity. Per questo motivo, è stato deciso di passare a ROS 2 Humble, una versione più stabile e ampiamente supportata, che ha garantito una migliore compatibilità con gli strumenti di sviluppo utilizzati.

2.2.3 Moveit

Figure 2.4: *Logo Moveit*

MoveIt [3] [4] è una delle librerie più utilizzate per la pianificazione del movimento in ROS. Consente di controllare bracci robotici, eseguire pianificazioni di traiettoria, gestire collisioni e implementare strategie di manipolazione avanzate. Grazie alla sua integrazione con ROS 2, MoveIt permette di generare e simulare movimenti in modo efficiente, facilitando l'implementazione di algoritmi di pianificazione e controllo.

MoveIt utilizza il modello del robot descritto tramite URDF per comprendere la struttura cinematica del sistema e genera automaticamente il corrispettivo SRDF (Semantic Robot Description Format) contenente la generazione di movimenti validi evitando collisioni. Il suo framework è composto da diversi moduli, tra cui:

- Motion Planning: per la generazione di traiettorie.
- Collision Checking: per evitare urti tra il robot e l'ambiente.
- Kinematic Solvers: per calcolare i movimenti articolari necessari a raggiungere una determinata posizione.
- Trajectory Execution: per inviare comandi al robot fisico o alla simulazione.

Nel nostro progetto, MoveIt è stato utilizzato per pianificare e simulare i movimenti del manipolatore Omron TM5-900. Tramite il MoveIt Setup Assistant è stato possibile definire i gruppi cinematici, i giunti controllabili e le geometrie di collisione.

MoveIt è stato integrato con ROS 2 Humble tramite la versione Moveit2 e tramite questa libreria sono state svolte le seguenti operazioni:

- Creazione del file SRDF tramite MoveIt Setup Assistant.
- Definizione dei controller nel file $ros2\_controllers.yaml$.
- Definizione delle pose iniziali e finali del robot.
- Invio delle traiettorie a Unity per visualizzare e simulare il movimento del robot in un contesto 3D realistico.

Alla sezione 2.5 è presente una descrizione dettagliata dei passaggi svolti con il MoveIt Setup Assistant per la creazione del file SRDF e la definizione dei controller.

2.2.4 Gazebo

Modellazione e simulazione robotica in ROS 2 con URDF, XACRO, SDF e Gazebo

In ambiente **ROS 2**, la modellazione e la simulazione robotica si basano su un insieme di strumenti e formati standardizzati che consentono la rappresentazione dettagliata dei robot e del mondo simulato. Il formato **URDF** (Unified Robot Description Format) viene utilizzato per descrivere la struttura fisica e cinematica di un robot, includendo elementi come i *link* (parti rigide), le *joint* (articolazioni mobili), le geometrie visuali e di collisione, e le proprietà fisiche quali massa, inerzia e materiali. URDF è nativamente supportato da ROS 2 ed è lo standard di riferimento per la descrizione dei modelli robotici.

Per migliorare la modularità e la riusabilità del codice, viene spesso impiegata l'estensione **XACRO** (XML Macros), che consente di definire macro, parametri e inclusioni di file esterni. XACRO permette una gestione più efficiente di modelli complessi, sebbene sia necessario convertirne i file in formato URDF standard prima dell'utilizzo in Gazebo, tramite un semplice comando da terminale:

```
xacro my_robot.urdf.xacro > my_robot.urdf
```

La simulazione fisica e visuale viene gestita da **Gazebo**, un simulatore open source sviluppato originariamente presso la University of Southern California e successivamente mantenuto dalla *Open Source Robotics Foundation*. Gazebo permette la simulazione realistica dell'interazione tra robot e ambiente, includendo dinamiche fisiche, sensori virtuali, rendering 3D e supporto a plugin personalizzati.

Per la descrizione dell'ambiente simulato, Gazebo utilizza un formato nativo denominato **SDF** (Simulation Description Format), più completo rispetto a URDF e adatto a rappresentare terreni, luci, oggetti dinamici o statici, proprietà ambientali e plugin. Nell'ambito ROS 2, l'approccio più comune consiste nell'utilizzare URDF (o XACRO) per la descrizione del robot e SDF per la definizione del mondo simulativo.

Il collegamento tra ROS 2 e Gazebo è garantito dal pacchetto `gazebo_ros`, che funge da interfaccia tra i due ambienti. Questo bridge consente l'integrazione di modelli URDF/XACRO nella simulazione, la pubblicazione dei dati sensoriali simulati su topic ROS 2, l'interazione con i robot tramite comandi ROS e la sincronizzazione del tempo simulato con il

clock del sistema ROS. L'adozione combinata di URDF/XACRO, SDF, Gazebo e `gazebo_ros` rappresenta dunque una soluzione completa e flessibile per lo sviluppo, il test e la validazione di sistemi robotici complessi all'interno di ambienti simulati.

2.2.5 Unity e UnityHUB

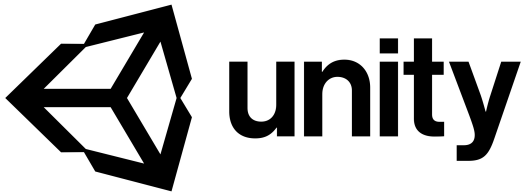

Figure 2.5: *Logo Unity*

Unity [5] è un motore di gioco e simulazione ampiamente utilizzato per la creazione di ambienti interattivi in tempo reale. Sebbene sia nato per lo sviluppo di videogiochi, il suo utilizzo si è esteso ad applicazioni di simulazione, robotica, realtà virtuale e aumentata. Unity permette di creare ambienti 3D realistici e interattivi grazie al suo motore grafico avanzato e alle sue funzionalità di scripting basate su C#. In questo progetto, Unity è stato utilizzato come ambiente di simulazione per il robot. Grazie alla sua compatibilità con ROS2 tramite il ROS-TCP Connector, Unity ha permesso di visualizzare il robot, simulare i suoi movimenti e interagire con l'ambiente circostante. Il motore fisico di Unity è stato sfruttato soprattutto per replicare le dinamiche fisiche della maglietta, garantendo una simulazione più realistica delle interazioni tra un indumento e gli oggetti della scena. Per lo sviluppo del progetto, abbiamo utilizzato Unity 6, la versione più recente al momento di sviluppo del progetto.

2.2.6 Strumento Cloth di Unity

Il sistema **Cloth di Unity** offre una soluzione basata sulla fisica per simulare tessuti e materiali flessibili all'interno di ambienti 3D. Sebbene sia stato progettato principalmente per rappresentare abbigliamento su personaggi, può essere utilizzato anche per altri scopi, come bandiere, tende o qualsiasi altro oggetto che richieda una simulazione realistica del comportamento dei tessuti[6]. In questo progetto, il componente Cloth riveste un ruolo centrale, poiché il compito finale prevede che l'indumento manipolato dal robot presenti una fisica il più possibile realistica, simulando accuratamente il comportamento del tessuto. A tal fine, sono stati forniti tre file principali relativi a una maglietta: un file **OBJ**, un file **FBX** e un file **Collada**. L'analisi di questi file ha permesso di ottenere informazioni utili riguardo al formato da utilizzare e alle caratteristiche della mesh per applicare efficacemente il componente Cloth.
Tra la documentazione di Unity è possibile trovare anche quella relativa al componente Cloth

e in seguito verranno riassunti i passaggi fondamentali per utilizzarlo al meglio. Per implementare una simulazione di tessuto in Unity, è necessario aggiungere il componente Cloth a un oggetto mesh. Ecco i passaggi fondamentali:

- **Preparazione della Mesh**: è importante assicurarsi che l'oggetto a cui si desidera applicare il tessuto abbia una mesh adeguata, e quindi, come visto in precedenza, anche un numero adeguato di poligoni della mesh.
- **Aggiunta del Componente**: una volta importato il file su Unity è sufficiente selezionare l'oggetto nella gerarchia di Unity e, nel pannello Inspector, cliccare su "Add Component", selezionare "Physics" e poi "Cloth" dalla lista dei componenti disponibili.
- **Configurazione**: Una volta aggiunto il componente, sarà possibile modificare vari parametri per ottenere una simulazione personalizzata e accurata.

Affinchè la maglietta abbia una movimentazione guidata dalla fisica, deve chiaramente avere una parte della maglietta fissata e che quindi non è soggetta alla gravità. Questi su Unity vengono chiamati "Cloth Constraint" e quindi "vincoli del Cloth". Di seguito una descrizione dei passaggi per applicarli correttamente alla maglietta:

1. sul componente Cloth si clicca su "Edit Cloth Constraint" come possiamo vedere dalla figura 2.6

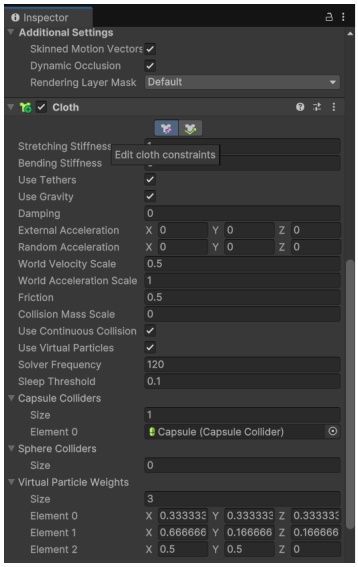

Figure 2.6: *Schermata Unity per l'applicazione dei vincoli*

2. successivamente si aprirà una schermata che permetterà di selezionare (tramite "Select") oppure colorare (tramite "Paint") le parti che saranno vincolate e quindi rimarranno maggiormente rigide. E' importante specificare una distanza minima nel parametro "Max Distance", in figura 2.7 è stato assegnato il valore 0.2 ed ha permesso un buon risultato.

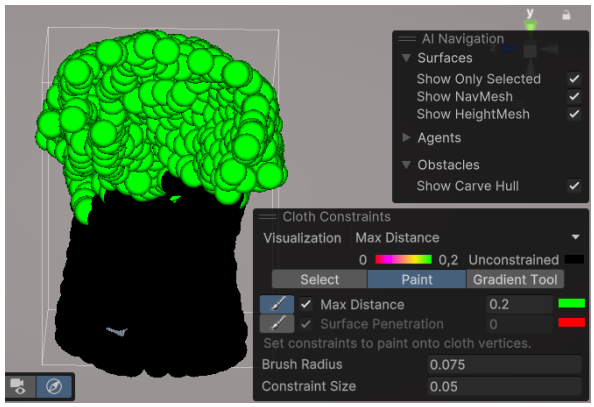

Figure 2.7: *Schermata Unity per la selezione delle parti della mesh a cui applicare i vincoli*

Infine una descrizione delle principali proprietà del componente Cloth e di come sono state utilizzate nell'implementazione del tessuto realistico della maglietta:

- **Stretching Stiffness**: Determina la resistenza del tessuto all'allungamento. Valori più alti rendono il tessuto meno incline a estendersi. Un parametro accettabile per la maglietta è risultato compreso tra 0,5 e 0,7.
- **Bending Stiffness**: Controlla la rigidità alla flessione del tessuto. Un valore elevato riduce la capacità del tessuto di piegarsi. Un parametro accettabile per la maglietta è risultato compreso tra 0,7 e 0,9.
- **Use Tethers**: Applica vincoli che aiutano a prevenire che le particelle mobili del tessuto si allontanino troppo da quelle fisse, riducendo l'eccessiva elasticità.
- **Use Gravity**: Indica se la gravità deve influenzare il tessuto. Da applicare nel caso in cui si voglia vedere una movimentazione fisica della maglietta e condizionata dalla gravità.
- **Damping**: Coefficiente che determina quanto velocemente il movimento del tessuto si smorza nel tempo. Un parametro accettabile per la maglietta è risultato compreso tra 0,2 e 0,4.
- **External Acceleration**: Applica un'accelerazione costante esterna al tessuto, utile per simulare effetti come il vento.
- **Random Acceleration**: Introduce un'accelerazione casuale al tessuto, aggiungendo variazioni imprevedibili nel movimento.
- **World Velocity Scale**: Determina quanto il movimento in spazio globale dell'oggetto influisce sui vertici del tessuto.
- **World Acceleration Scale**: Controlla l'influenza dell'accelerazione globale dell'oggetto sui vertici del tessuto.

- **Friction**: Imposta il coefficiente di attrito del tessuto durante le collisioni. Un parametro accettabile per la maglietta è risultato compreso tra 0,5 e 0,7.
- **Collision Mass Scale**: Determina l'incremento di massa delle particelle durante le collisioni.
- **Use Continuous Collision**: Abilita la collisione continua per migliorare la stabilità delle interazioni.
- **Use Virtual Particles**: Aggiunge particelle virtuali per migliorare la stabilità delle collisioni.
- **Solver Frequency**: Specifica il numero di iterazioni del solver per secondo, influenzando la precisione della simulazione. Un parametro accettabile per la maglietta è risultato 60 Hz.
- **Sleep Threshold**: Definisce la soglia sotto la quale il tessuto entra in stato di "sonno", interrompendo la simulazione fino a nuove interazioni. Un parametro accettabile per la maglietta è risultato 0,1.
- **Capsule Colliders**: Array di collisori a capsula con cui il tessuto può interagire.
- **Sphere Colliders**: Array di coppie di collisori sferici con cui il tessuto può interagire.

2.3 Predisposizione dei Software nel PC

In questo capitolo viene presentata la configurazione dell'ambiente di lavoro adottato per lo sviluppo del progetto, con particolare attenzione alle versioni e alle modalità di integrazione dei principali software utilizzati: Ubuntu, ROS2, Gazebo, Unity (insieme a Unity Hub) e MoveIt2.

Per la parte robotica e di simulazione, è stato scelto di adottare **Ubuntu 22.04** installato tramite una macchina virtuale WSL2, in quanto offre un ambiente stabile e supportato ufficialmente per lo sviluppo con **ROS2 Humble** e **MoveIt2**. ROS2 Humble è stato selezionato per la sua affidabilità e le migliorate capacità di integrazione, che hanno risolto i problemi riscontrati con precedenti versioni (come Jazzy).**Gazebo Fortress**, la cui scelta della versione è dovuta alla compatibilità con ROS2[7], garantisce simulazioni avanzate e un'elevata accuratezza nella riproduzione delle dinamiche robotiche, tuttavia non ha un ruolo centrale nel progetto in quanto non è possibile applicare la fisica ai tessuti come in Unity.

Parallelamente, per lo sviluppo dell'ambiente 3D e della visualizzazione in tempo reale, si è mantenuto **Unity 6**(ultima versione) su Windows, supportato da **Unity Hub** che permette una gestione semplificata dei vari progetti Unity. La scelta di utilizzare Windows come sistema operativo per Unity è motivata dalla sua eccellente compatibilità con gli strumenti grafici e i driver necessari per un'interfaccia utente fluida e performante, in quanto Unity è un software nativo Windows. La comunicazione tra l'ambiente robotico gestito tramite Ubuntu e quello grafico su Windows è resa possibile attraverso il ROS-TCP Connector, che funge da ponte tra le due piattaforme, garantendo lo scambio efficiente dei dati.

Questa configurazione ibrida, che sfrutta al massimo le potenzialità di ogni sistema operativo e software specifico, consente di ottimizzare sia le prestazioni delle simulazioni che

l'interattività dell'interfaccia grafica, contribuendo a rendere il progetto complessivamente robusto e flessibile.

2.4 Interfacciamento ROS2-Unity

Il capitolo "Interfacciamento ros2-Unity" si propone di illustrare come integrare in maniera efficace ROS2 con l'ambiente grafico e simulativo offerto da Unity, al fine di realizzare una piattaforma in grado di gestire in tempo reale sia la simulazione fisica che la comunicazione dei dati. In particolare, saranno mostrati il ROS-TCP Connector e il ROS-TCP Endpoint per lo scambio di messaggi tra ROS2 e Unity, nonché l'URDF Importer per la corretta rappresentazione del robot all'interno di Unity.

2.4.1 ROS-TCP Connector & ROS-TCP Endpoint

Unity Technologies fornisce un pacchetto che consente di integrare ROS2 con Unity, facilitando la comunicazione tra i due ambienti. Questo pacchetto è composto da due nodi principali e complementari tra loro: il **ROS-TCP Connector** e il **ROS-TCP Endpoint**. In particolare il **ROS-TCP Connector** è un pacchetto Unity che consente di inviare e ricevere messaggi tra Unity e ROS2, mentre il **ROS-TCP Endpoint** è un nodo ROS2 che funge da server TCP, ascoltando le connessioni in entrata da Unity e gestendo lo scambio di messaggi tra i due ambienti.

Il pacchetto ROS-TCP Connector include:

- **ROS-TCP Connector**: permette l'invio e la ricezione di messaggi tra Unity e ROS2.
- **VISUALIZATIONS PACKAGE**: utile per la visualizzazione dei messaggi in entrata e in uscita nella scena di Unity, facilitando il debug e l'analisi dei dati.

Inoltre, il pacchetto offre diverse funzionalità:

- **ROSConnection**: gestisce l'intero processo di comunicazione tra Unity e ROS2.
- **Message Generation**: permette di generare automaticamente classi in C# che rappresentano i messaggi ROS2, facilitando l'interazione tra i due ambienti.
- **Visualizations**: un set di API e configurazioni predefinite per rappresentare le informazioni scambiate.
- **ROSGeometry**: una serie di estensioni utili per la conversione delle geometrie tra Unity e altri sistemi che semplificano la compatibilità tra gli ambienti di simulazione.

Per utilizzare il ROS-TCP Connector, è necessario installarlo all'interno di Unity e configurarlo correttamente affinché possa dialogare con ROS2.

Il pacchetto è disponibile su GitHub e può essere importato in Unity tramite il Package Manager. Ecco come fare:

- Aprire Unity e accedere al menu "Window", poi selezionare "Package Manager".

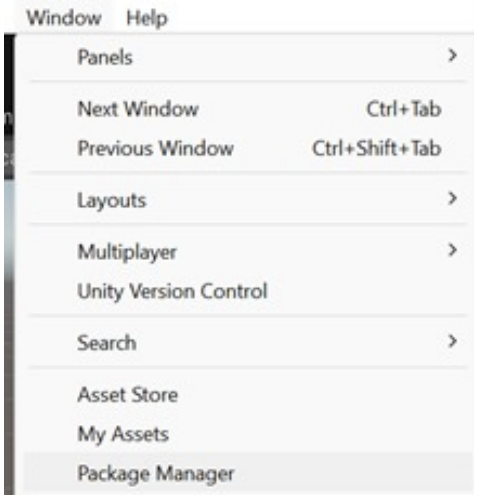

Figure 2.8: *Package Manager in Unity*

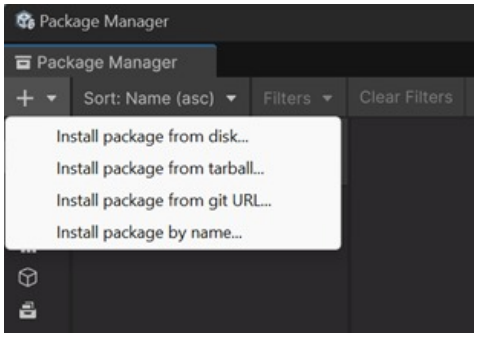

Figure 2.9: *Package Manager in Unity*

- Nella finestra del Package Manager, cliccare sul pulsante "+" in alto a sinistra e selezionare "Add package from git URL...".
- Inserire l'URL del pacchetto da installare:
  - **ROS-TCP Connector**: `https://github.com/Unity-Technologies/ROS-TCP-Connector.git?path=/com.unity.robotics.ros-tcp-connector`

  In alternativa, se si ha una copia locale del pacchetto, è possibile installarlo seguendo la guida ufficiale di Unity per l'installazione di pacchetti locali.

Dopo aver installato il pacchetto, è necessario configurarlo per far sì che possa comunicare con ROS2. Questo passaggio è essenziale per garantire che Unity e ROS2 possano scambiarsi dati senza problemi:

- Accedere al menu "Robotics" e selezionare "ROS Settings".

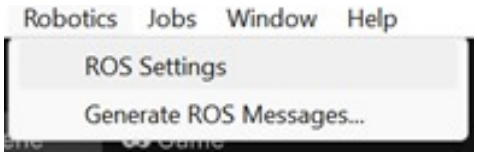

Figure 2.10: *Menu "Robotics" e "ROS Settings"*

- Nel campo "ROS IP Address", inserire l'indirizzo IP della macchina su cui è in esecuzione ROS2.

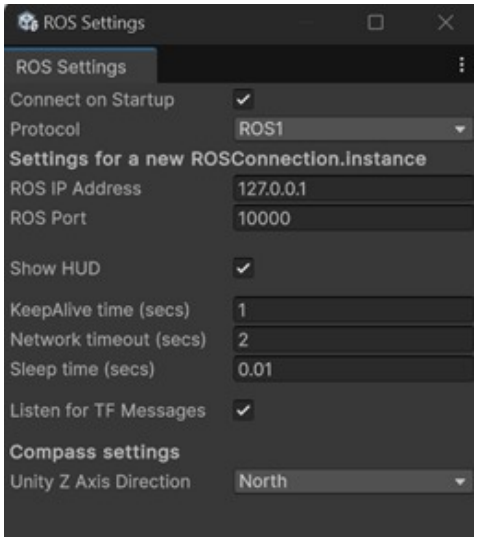

Figure 2.11: *Configurazione dell'indirizzo IP di ROS2*

Per reperire l'indirizzo IP di ROS (ROS IP Address) sulla macchina con WSL, è sufficiente aprire un terminale e digitare:

```
hostname -I
```

Ora lato Unity la comunicazione è pronta, quindi procediamo con l'installazione e la configurazione del **ROS-TCP Endpoint** lato ROS2. Per utilizzarlo, è necessario installarlo all'interno del proprio workspace ROS2, clonando la repository tramite il comando: [8]

```
git clone https://github.com/Unity-Technologies/ROS-TCP-Endpoint.git
```

Dopo aver completato l'installazione e la configurazione, Unity sarà in grado di comunicare con ROS2 attraverso il protocollo TCP. Questo avviene tramite due modalità principali:

- **Pubblicazione di Messaggi**: Unity può inviare dati a ROS2 utilizzando il componente `ROSPublisher`.
- **Sottoscrizione di Messaggi**: Unity può ricevere dati da ROS2 grazie al componente `ROSSubscriber`.

A questo punto, per verificare che le installazioni e le configurazioni siano avvenute con successo è possibile seguire un breve tutorial per testare la connessione tra Unity e ROS2. In particolare l'obiettivo sarà quello di inviare le coordinate di un oggetto di Unity a ROS2 tramite un Publisher e ricevere le coordinate di un oggetto di ROS2 in Unity tramite un Subscriber, il link è in bibliografia.[9]

Tuttavia, seguono i comandi generali che saranno poi nuovamente riproposti alla sezione 5.

Inizialmente, è necessario avviare il nodo `ros_tcp_endpoint` all'interno di ROS2. Questo nodo funge da intermediario e garantisce che i messaggi possano fluire correttamente tra i due sistemi.

Per avviare il nodo **ros_tcp_endpoint**, bisogna eseguire il comando:

```
ros2 run ros_tcp_endpoint default_server_endpoint --ros-args -p ROS_IP:=YOUR_IP
```

Dove per conoscere l'indirizzo IP del proprio computer è possibile utilizzare il comando:

```
hostname -I
```

In questo modo si avvia il server che ascolta sulla porta TCP (di default 10000) per ricevere le connessioni da Unity.

A questo punto, se nella Scena di Unity è presente un Publisher o un Subscriber configurato correttamente, Unity tenterà di connettersi al nodo **ros_tcp_endpoint** e inizierà a inviare o ricevere messaggi. In particolare una volta avviata la simulazione in Unity dovremmo vedere le due frecce rosse come mostrato in figura 2.12.

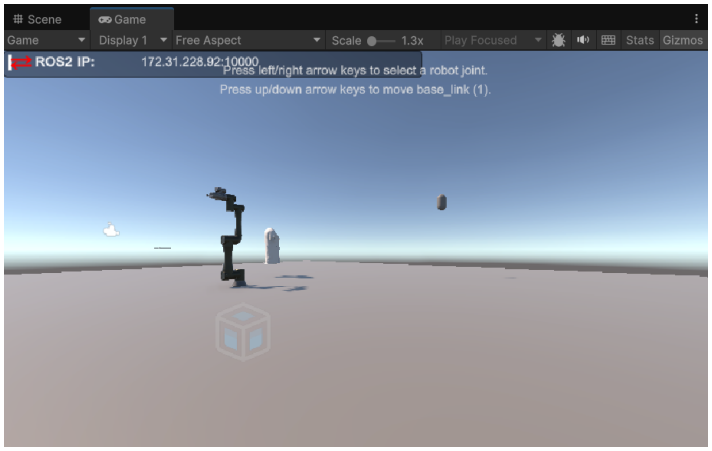

Figure 2.12: *Test della connessione una volta avviata la simulazione*

Successivamente, una volta che è stato avviato il ROS-TCP-Endpoint le frecce cambieranno colore in blu, indicando che la connessione è stata stabilita con successo, come mostrato in figura 2.13.

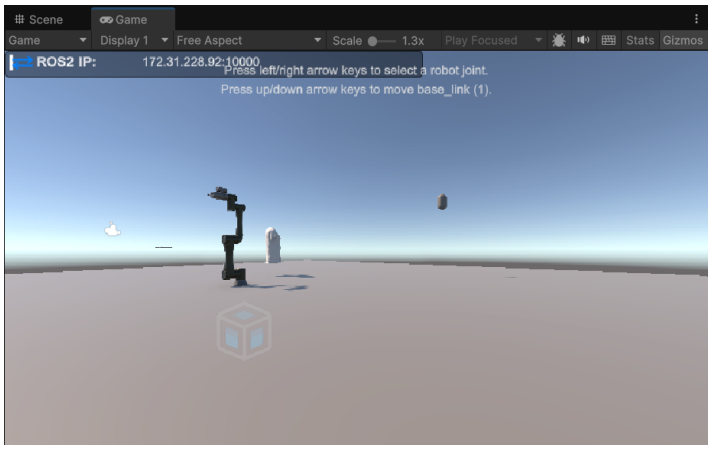

Figure 2.13: *Test della connessione dopo aver avviato il ROS-TCP-Endpoint*

2.4.2 URDF Importer

L'**URDF Importer** [10] di Unity è uno strumento che permette di importare file URDF (Unified Robot Description Format) direttamente in Unity, consentendo così di visualizzare e interagire con modelli robotici 3D all'interno dell'ambiente di simulazione. Questo strumento risulta particolarmente utile nel progetto perché permette di integrare ROS 2 e Unity tramite il file URDF fortemente utilizzato in robotica, inoltre semplifica la gestione della cinematica e della fisica dei robot senza dover ricostruire manualmente i modelli.

Il formato URDF è ampiamente utilizzato nel mondo della robotica per descrivere la struttura dei robot, inclusi i links, i joints, i materiali e i sensori. L'URDF Importer converte questi modelli in GameObjects di Unity, mantenendo la gerarchia dei componenti e assegnando i giunti come oggetti fisici controllabili.

Nel nostro progetto, il robot utilizzato è un Omron TM5-900, un braccio robotico con sei gradi di libertà. Il modello URDF di questo robot contiene:

- Link per ogni segmento del braccio, dalla base all'end-effector.
- Joints di tipo rotoidale che consentono il movimento.
- Materiali per rappresentare visivamente il robot in Unity.
- Collisioni e inerzia, che permettono una simulazione fisica realistica.

L'importazione di questo URDF in Unity è essenziale per poterlo visualizzare correttamente e per interagire con esso tramite ROS 2.

Per importare installare lo strumento URDF Importer è possibile seguire i seguenti passaggi:

- Cliccare su "Window" poi "Package Manager"

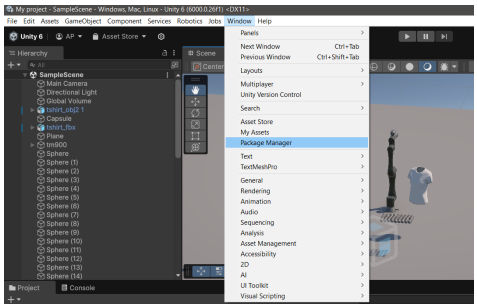

Figure 2.14: *URDF Importer Tutorial*

- Cliccare su "+" e selezionare "Add package from git URL..."

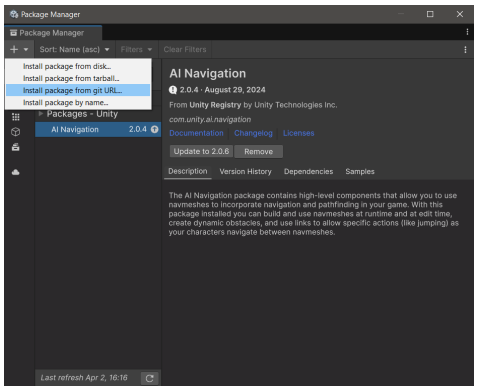

Figure 2.15: *URDF Importer Tutorial*

- Inserire il seguente URL della repository: `https://github.com/Unity-Technologies/URDF-Importer.git` e cliccare "install" per avviare l'installazione

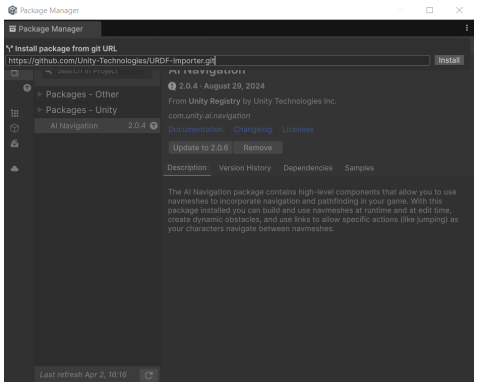

Figure 2.16: *URDF Importer Tutorial*

Per importare il modello URDF su Unity invece:

- Importare il file URDF e le varie mesh sia visual che collision nella cartella "Assets" del progetto Unity

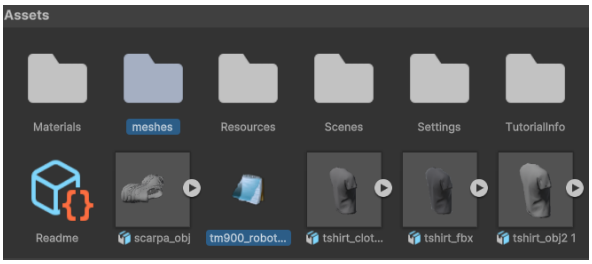

Figure 2.17: *URDF Importer Tutorial*

- Verificare che i vari path delle mesh all'interno del file URDF siano del tipo ""package://path/delle/meshes" come mostrato in figura 2.18

```
<link name="base_link">
  <visual>
    <geometry>
      <mesh filename="package://meshes/tm900/visual/Base.STL"/>
    </geometry>
    <material name="Grey">
      <color rgba="0.5 0.5 0.5 1.0"/>
    </material>
  </visual>
  <collision>
    <geometry>
      <mesh filename="package://meshes/tm900/collision/base.STL"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="1.0"/>
    <insert_block name="origin"/>
    <inertia ixx="0.00110833289" ixy="0.0" ixz="0.0" iyy="0.00110833289" iyz="0.0" izz="0.0018"/>
  </inertial>
</link>
```

Figure 2.18: *URDF Importer Tutorial*

- Tasto destro sul file URDF e cliccare su "Import Robot from selected URDF file"

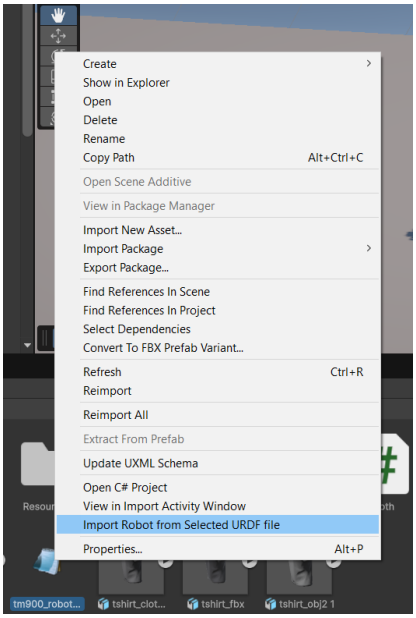

Figure 2.19: *URDF Importer Tutorial*

A questo punto, se è stato utilizzato il file urdf del TM5-900 corretto, sarà possibile visualizzare il robot come mostrato in figura 2.20.

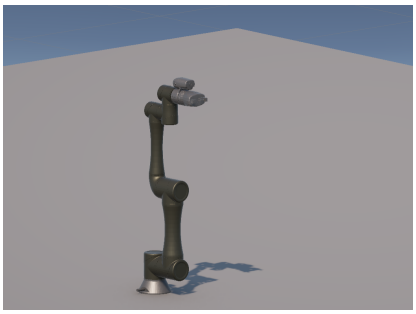

Figure 2.20: *URDF Importer Tutorial*

2.5 Configurazione del Moveit Setup Assistant

Di seguito sono riportati i passaggi svolti per la configurazione del package moveit_config_tm5_900 utilizzando il MoveIt Setup Assistant [11]:

1. Lanciamo il MoveIt Setup Assistant con il comando

   ```
   roslaunch moveit_setup_assistant setup_assistant.launch
   ```

2. Clicchiamo su "Create New MoveIt Configuration Package" e selezioniamo il file URDF del robot

3. Nella sezione "Self Collision" è possibile generare automaticamente il controllo delle auto-collisioni del robot, selezionando "Generate Collision Matrix".

4. Tramite "Virtual Joint" è possibile definire il giunto virtuale del robot, che rappresenta la connessione tra il robot e il mondo esterno. In questo caso, abbiamo selezionato "fixed" come tipo di giunto.

5. Nella sezione "Planning Groups" aggiungere il gruppo "Manipulator" che comprende tutti i giunti a partire da "shoulder_1_ joint" fino a "wrist_3_joint". Se nel URDF è presente la pinza è possibile aggiungere anche il gruppo "Gripper".

6. Successivamente tramite "Robot Poses" sono state definite due pose, una di riposo(Home) e una di lavoro(Work). In caso di aggiunta della pinza si possono definire due pose della pinza in modo che il robot possa afferrare oggetti e manipolarli.

7. Infine generiamo i controllori "Ros2 Controllers" e "Ros2 MoveIt Controllers" per la comunicazione tra il robot e MoveIt. Per il nostro progetto abbiamo selezionato rispettivamente il controller "JointTrajectoryController" e "FollowJointTrajectory" per il gruppo "Manipulator".

8. A questo punto rimane solo da generare il package tramite "Genera Package" nella sezione "Configurations Files".

MoveIt ha svolto un ruolo centrale nell'interazione tra ROS 2 e Unity e ci ha permesso di simulare la movimentazione del robot solo tramite due codici, uno in C# per Unity e uno in Python per ROS2, che verranno poi approfonditi nella sezione 6. Le traiettorie generate da MoveIt sono state pubblicate sul topic $/manipulator_controller/joint_trajectory$, successivamente sottoscritto in Unity tramite il pacchetto ROS-TCP Connector. Questo ha permesso una rappresentazione visiva coerente tra la pianificazione in ROS 2 e l'esecuzione nella simulazione 3D.

Inoltre è stato possibile utilizzare il package generato per avviare RViz, un potente strumento di visualizzazione per ROS, che consente di monitorare e analizzare i movimenti del robot in tempo reale. RViz è stato utilizzato per verificare la correttezza della pianificazione delle traiettorie e per eseguire test preliminari prima di passare alla simulazione in Unity.

Tramite il comando:

```
ros2 launch moveit_config_tm5_900 demo.launch.py
```

è possibile avviare il package generato e visualizzare il robot in RViz, in quanto l'URDF del robot è già presente nel package.

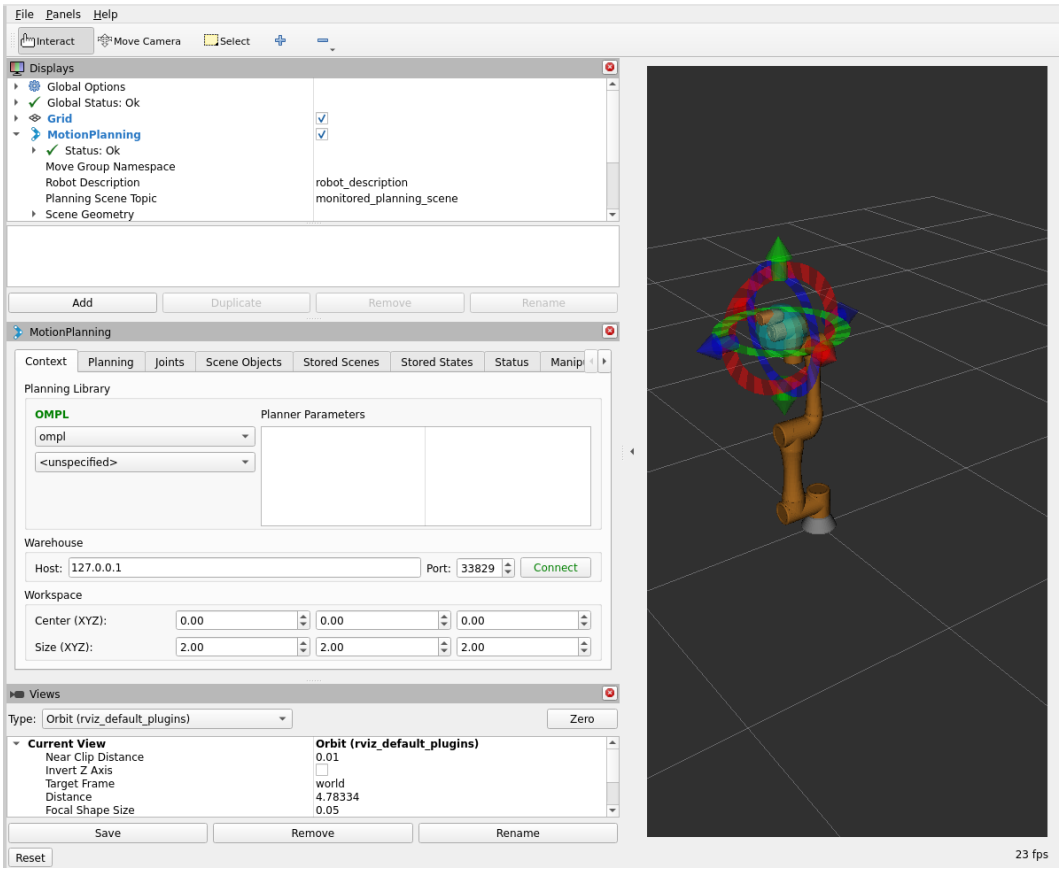

Figure 2.21: *Schermata iniziale RViz*

Dalla figura 2.21 è possibile vedere la schermata iniziale una volta lanciato il comando sopra indicato. A questo punto, nella sezione "Planning", indicando il gruppo di giunti, una

posizione iniziale e finale del robot, come mostrato in figura 2.22, RViz salverà le posizioni dei giunti di entrambe le pose. Successivamente, cliccando su "Plan & Execute" il robot si muoverà nella posizione finale indicata, simulanod la traiettoria.

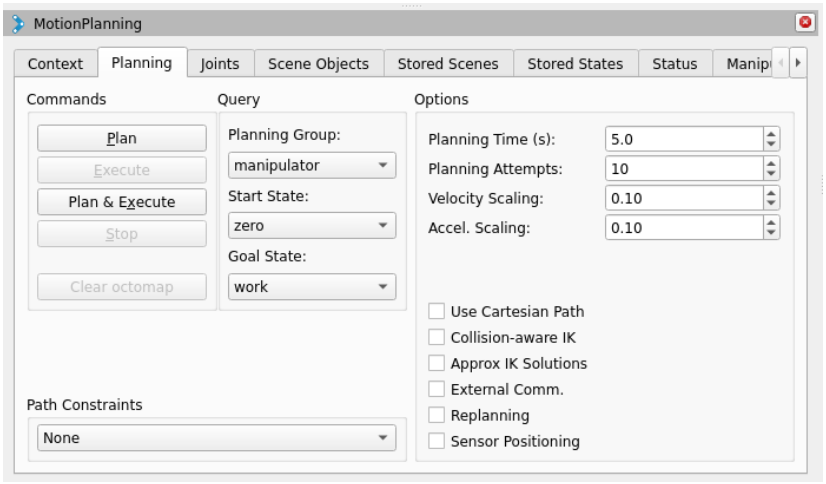

Figure 2.22: *Configurazione della pianificazione ed esecuzione della traiettoria in RViz*

Non solo, tramite RViz è inoltre possibile creare nuove pose come mostrato in figura 2.23, e testare le traiettorie in breve tempo.

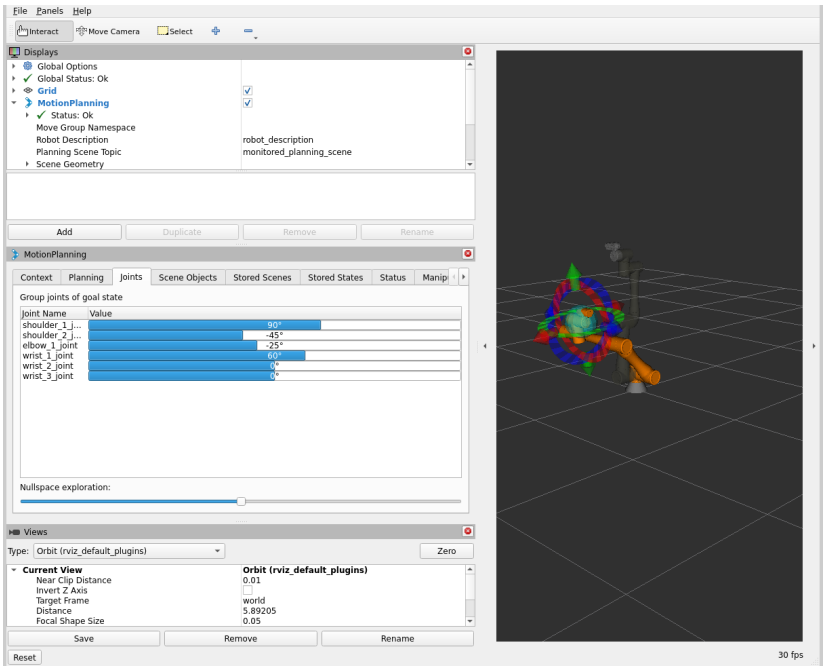

Figure 2.23: *Configurazione della posizione dei giunti in RViz*

Nel mentre, è risultato molto utile controllare il terminale durante queste operazioni

perché Moveit riportava eventuali errori o problemi di collisione tra il robot e l'ambiente circostante. Questo ha permesso di ottimizzare le traiettorie e garantire un movimento fluido e sicuro del robot.

2.6 Importazione TM5-900 in Gazebo

In questo capitolo viene presentata la configurazione del robot TM5-900 in Gazebo, con particolare attenzione alla creazione del file URDF e alla definizione dei plugin necessari per la simulazione.

URDF in Gazebo

Il file URDF (*Unified Robot Description Format*) è un file XML usato per rappresentare la struttura fisica, cinematica e dinamica di un robot. Questa descrizione è fondamentale per simulazioni, visualizzazioni 3D e per l'interazione con i sistemi ROS.
Tutti gli elementi sono racchiusi nel tag `<robot>`:

```
<robot name="tm900">
    ...
</robot>
```

Ogni `link` rappresenta una parte rigida del robot. Ogni link include:

- `<visual>`, descrizione visiva (mesh o primitiva)
- `<collision>`, forma per la collisione
- `<inertial>`, massa, centro di massa e inerzia

Ecco un esempio per il link `shoulder_1_link`:

```
<link name="shoulder_1_link">
  <visual>
    <geometry>
      <mesh filename="package://tm900/meshes/shoulder_1.stl" />
    </geometry>
    <material name="grey" />
  </visual>
  <collision>
    <geometry>
      <mesh filename="package://tm900/meshes/shoulder_1.stl" />
    </geometry>
  </collision>
  <inertial>
    <mass value="2.5" />
    <origin xyz="0 0 0" rpy="0 0 0" />
    <inertia
      ixx="0.03" ixy="0.0" ixz="0.0"
```

```
        iyy="0.03"  iyz="0.0"
        izz="0.03" />
  </inertial>
</link>
```

I `joint` collegano due link tra loro e ne definiscono il tipo di movimento relativo. Ogni joint contiene:

- `name`, `type` (revolute, fixed, prismatic, ecc.)
- `<parent>` e `<child>`
- `<origin>`, posizione e orientamento del giunto
- `<axis>`, direzione dell'asse di rotazione
- `<limit>`, limiti meccanici (per giunti mobili)

Ecco un esempio:

```
<joint name="shoulder_1_joint" type="revolute">
  <parent link="base_link" />
  <child link="shoulder_1_link" />
  <origin xyz="0 0 0.1" rpy="0 0 0" />
  <axis xyz="0 0 1" />
  <limit lower="-3.14" upper="3.14" effort="20" velocity="1.0" />
</joint>
```

Le proprietà fisiche sono definite nei blocchi `<inertial>` e `<collision>` di ogni link. Queste includono:

- **Massa** – specificata in kg
- **Momento d'inerzia** – matrice 3x3, usata per simulazioni dinamiche
- **Collisioni** – geometrie per il rilevamento delle collisioni
- **Materiali** – colore e aspetto visivo

Il controllo avviene tramite plugin ROS. Ogni giunto attuato ha una trasmissione:

```
<transmission name="elbow_trans">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="elbow_joint">
    <hardwareInterface>PositionJointInterface</hardwareInterface>
  </joint>
  <actuator name="elbow_motor">
    <hardwareInterface>PositionJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>
```

Per la simulazione in Gazebo si definisce il plugin:

```
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so"/>
</gazebo>
```

Per ogni link si può abilitare la self-collision:

```
<gazebo reference="wrist_link">
  <selfCollide>true</selfCollide>
</gazebo>
```

Per importare il modello del TM5-900 in Ignition Gazebo abbiamo innanzitutto predisposto un environment minimale, avviando un mondo vuoto tramite il comando:

```
ign gazebo empty.sdf
```

In questo modo si crea uno spazio di simulazione "pulito", privo di elementi superflui, che ci permette di verificare in modo isolato il corretto caricamento del robot. Successivamente, abbiamo impiegato il servizio "EntityFactory" di Gazebo per "spawnare" il robot a partire dal file URDF prodotto dal pacchetto "tm_description" di ROS2 Humble. Nello specifico, il comando utilizzato è stato:

```
ign service -s /world/empty/create \
--reqtype ignition.msgs.EntityFactory \
--reptype ignition.msgs.Boolean \
--timeout 1000 \
--req "sdf_filename: '/home/pizzu01/ros2_humble/install/tm_description/share/tm_descript
```

Con –reqtype abbiamo definito rispettivamente il tipo di messaggio di richiesta e di risposta, mentre –timeout imposta un tempo massimo di attesa per la conferma del caricamento. La chiave sdf_filename punta al file URDF compilato durante l'installazione del pacchetto, garantendo che vengano letti correttamente tutti i link, i joint e i plugin definiti. Infine, assegnando al modello il nome "tm900" ci assicuriamo di poterlo indirizzare univocamente nelle successive chiamate di spawn o nei nodi ROS2 che interagiranno con esso. Alla conclusione della procedura, il TM5-900 è dunque presente nella scena di Gazebo e pronto per essere controllato e validato tramite i controller ROS2 e i plugin di interfaccia.

Durante il processo di sviluppo del modello URDF del robot TM900, è stato riscontrato un problema relativo alla visualizzazione delle mesh in Gazebo. Il problema e la soluzione adottata è documentata alla sezione **??**.

2.7 Camera

Questo capitolo descrive il task riguardante la camera e la soluzione ottenuta. Il problema consisteva nell'acquisizione e nella trasmissione in tempo reale delle immagini della scena virtuale (sia RGB sia di profondità) da Unity al framework ROS2. L'obiettivo è duplice: da un lato, permettere a ROS2 di ricevere flussi video ad alta frequenza per eventuali elaborazioni e algoritmi di visione artificiale; dall'altro, garantire una struttura modulare, facilmente

estendibile e conforme alle best practice di comunicazione offerta da ROS2. Successivamente, grazie a questo task è possibile integrare nel nella realtà la camera Realsense D455, che fornisce flussi video RGB e di profondità. In particolare, saranno descritti i passaggi per la preparazione del RenderTexture e la spiegazione del codice C# per l'acquisizione delle immagini. Infine, verranno presentati sia il codice Publisher di Unity che il Subscriber di ROS2.

La soluzione adottata per rispondere a questa esigenza, è la RenderTexture di Unity, utilizzata come fonte di acquisizione dei frame. Sono stati sviluppati due componenti software distinti:

- **Publisher Unity** uno script in C# che cattura ad intervalli regolari le texture della scena virtuale, estrae separatamente i dati RGB e quelli di profondità e li pubblica su due topic ROS2 distinti.
- **Subscriber ROS2** un nodo Python che si iscrive ai due topic, riceve i pacchetti di dati, li converte in immagini tramite OpenCV e infine li salva come video nel workspace di lavoro.

Inoltre è stato implementata una soluzione iniziale interamente in Unity che permette di esportare i dati RGB sia in formato .png che in .csv, in modo da avere un file di testo con le coordinate RGB di ogni pixel. Questo è stato utile per testare la funzionalità del Publisher e per verificare la veridicità dei dati.

2.7.1 Setup della Camera Virtuale

Prima di procedere con la scrittura del codice, è necessario configurare correttamente la scena in Unity. In particolare, è necessario creare un oggetto Camera che simuli il funzionamento della Realsense D455. Per fare ciò, seguire i seguenti passaggi:

- Eliminare l'oggetto `mainCamera` predefinito
- Creare un nuovo oggetto Camera dal menu GameObject, vedi figura 2.24
- Rinominarlo `realSenseCamera`
- Impostare il Field Of View a 87°, vedi figura 2.25

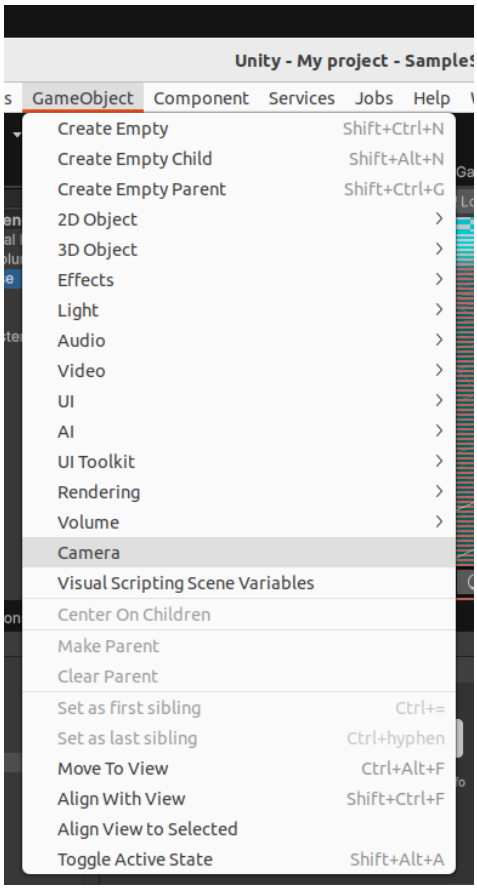

Figure 2.24: *Creazione dell'oggetto camera in Unity*

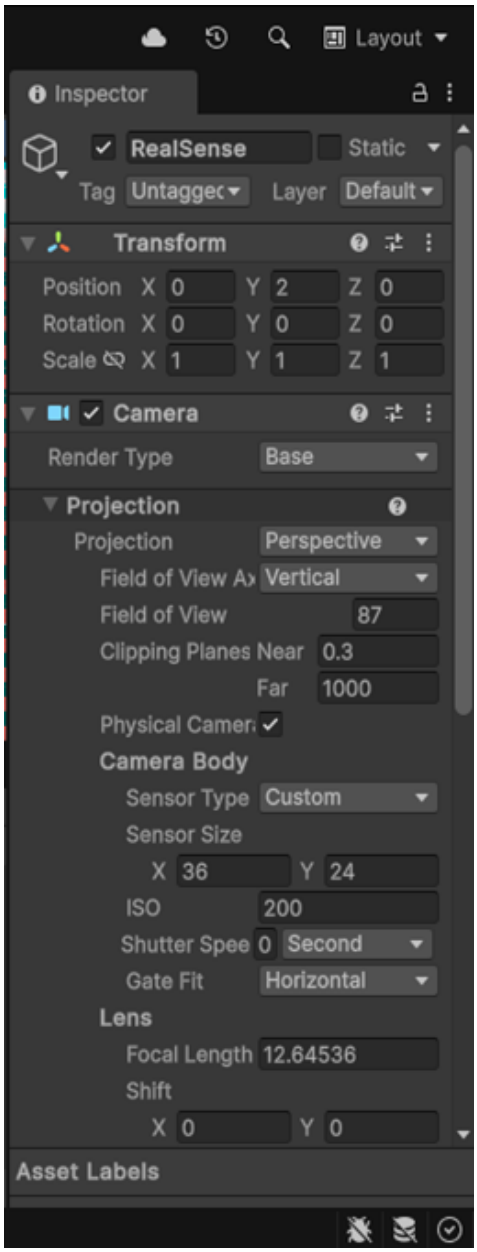

Figure 2.25: *Impostazione del Field Of View*

A questo punto, è possibile procedere con la creazione della scena. Per testare il funzionamento del sistema, è sufficiente creare un oggetto 3D (ad esempio una sfera) e posizionarlo nella scena. In questo modo, sarà possibile verificare che il sistema acquisisca correttamente le immagini e i dati di profondità. I passaggi da seguire sono i seguenti:

- Aggiungere una sfera come oggetto di test
- Assegnare materiali e colori
- Configurare l'illuminazione base

2.7.2 Implementazione codice per l'acquisizione delle immagini

Listing 2.1: Script base di acquisizione immagini

```
using UnityEngine;
using System.IO;

[RequireComponent(typeof(Camera))]
public class RealSenseSimulator : MonoBehaviour
{
    public Camera realSenseCamera;
    private RenderTexture renderTexture;
    private Texture2D colorTexture2D;
    private Texture2D depthTexture2D;

    void Start()
    {
        renderTexture = new RenderTexture(1280, 720, 24,
            RenderTextureFormat.ARGB32);
        realSenseCamera.targetTexture = renderTexture;
        realSenseCamera.depthTextureMode = DepthTextureMode.Depth;

        colorTexture2D = new Texture2D(1280, 720,
            TextureFormat.RGB24, false);
        depthTexture2D = new Texture2D(1280, 720,
            TextureFormat.RGB24, false);
    }

    void Update()
    {
        RenderTexture.active = renderTexture;
        colorTexture2D.ReadPixels(new Rect(0, 0, 1280, 720), 0, 0);
        colorTexture2D.Apply();

        SaveRGBValuesToFile();
        SaveRGBImage();
        StartCoroutine(CaptureDepth());
    }
```

```
private void SaveRGBValuesToFile()
{
    string filePath = "Assets/rgb_values.csv";
    using (StreamWriter writer = new StreamWriter(filePath, false))
    {
        writer.WriteLine("X,Y,R,G,B");
        for (int y = 0; y < colorTexture2D.height; y++)
        {
            for (int x = 0; x < colorTexture2D.width; x++)
            {
                Color pixelColor = colorTexture2D.GetPixel(x, y);
                writer.WriteLine($"{x},{y},{
                    pixelColor.r * 255},{
                    pixelColor.g * 255},{
                    pixelColor.b * 255}");
            }
        }
    }
    Debug.Log("RGB values saved to " + filePath);
}

private void SaveRGBImage()
{
    byte[] rgbData = colorTexture2D.EncodeToPNG();
    File.WriteAllBytes(
        Application.dataPath + "/simulated_rgb_image.png",
        rgbData);
}

private IEnumerator CaptureDepth()
{
    yield return new WaitForEndOfFrame();

    RenderTexture depthRT = RenderTexture.GetTemporary(
        1280, 720, 24, RenderTextureFormat.Depth);
    Graphics.Blit(null, depthRT,
```

```
            new Material(Shader.Find(
                "Hidden/Internal-DepthNormalsTexture")));

        RenderTexture.active = depthRT;
        Texture2D tempDepth = new Texture2D(1280, 720,
            TextureFormat.RFloat, false);
        tempDepth.ReadPixels(new Rect(0, 0, 1280, 720), 0, 0);
        tempDepth.Apply();

        for (int y = 0; y < depthTexture2D.height; y++)
        {
            for (int x = 0; x < depthTexture2D.width; x++)
            {
                float depthValue = tempDepth.GetPixel(x, y).r;
                Color gray = new Color(depthValue,
                    depthValue, depthValue);
                depthTexture2D.SetPixel(x, y, gray);
            }
        }
        depthTexture2D.Apply();

        byte[] depthData = depthTexture2D.EncodeToPNG();
        File.WriteAllBytes(
            Application.dataPath + "/simulated_depth_image.png",
            depthData);

        RenderTexture.ReleaseTemporary(depthRT);
        Destroy(tempDepth);
        Debug.Log("Depth image saved.");
    }
}
```

Attraverso l'utilizzo di una camera virtuale e quindi di una RenderTexture, il sistema acquisisce i dati visivi della scena e li converte in texture gestibili tramite Texture2D. Ogni n frame, i dati RGB vengono esportati in formato .png e salvati anche in un file .csv, consentendo un'analisi dettagliata dei singoli pixel. Contestualmente, viene generata una mappa di profondità simulata mediante una funzione sinusoidale che varia nel tempo, permettendo una rappresentazione dinamica della distanza. Una sezione specifica dello script è inoltre dedicata

alla simulazione di variazioni visive e di profondità localizzate, al fine di testare la sensibilità del sistema nel rilevare modifiche. L'architettura del codice è orientata all'efficienza e alla modularità, risultando adatta a scenari di test e sviluppo per sistemi di visione artificiale basati su dati RGB-D. La simulazione della profondità è stata realizzata tramite una funzione periodica per garantire un controllo preciso e dinamico dei valori di profondità, senza la necessità di un sensore fisico. Questo approccio consente di testare il comportamento del sistema in modo riproducibile. Inoltre, permette di verificare l'integrazione e la sincronizzazione tra il flusso visivo e quello di profondità in un contesto completamente simulato. Il metodo GenerateDepthData() simula una mappa di profondità generando valori di intensità in scala di grigi per ogni pixel, utilizzando una funzione sinusoidale che varia nel tempo. Questo crea un effetto ondulato e dinamico, utile per testare il comportamento di sistemi basati su dati di profondità senza l'uso di un sensore fisico.

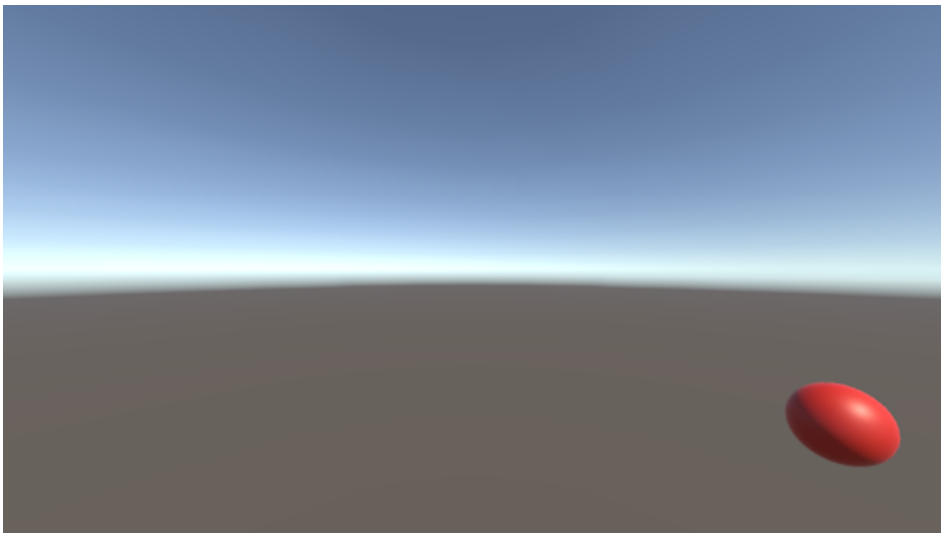

Figure 2.26: *Immagine creata in formato PNG*

2.7.3 Publisher di Unity

Di seguito, è riportato il codice del Publisher di Unity, che si occupa di pubblicare i dati RGB e Depth su due topic separati. Il codice è stato sviluppato in C# con le seguenti caratteristiche:

- Pubblicazione su due topic separati (RGB e Depth)
- Frequenza: 10Hz (0.1 secondi)
- Utilizzo del pacchetto Unity-Robotics-ROSTCPConnector
- Formato messaggi ROS standard

Listing 2.2: Publisher di Unity

```
using UnityEngine;
using System.IO;
```

```
using System.Collections;
using Unity.Robotics.ROSTCPConnector;
using RosMessageTypes.Sensor;

[RequireComponent(typeof(Camera))]
public class RealSenseSimulator : MonoBehaviour
{
    public Camera realSenseCamera;
    private RenderTexture renderTexture;
    private Texture2D colorTexture2D;
    private Texture2D depthTexture2D;

    public float publishInterval = 0.1f;
    private float nextPublishTime = 0f;
    public string colorTopic = "/camera/color/image_raw";
    public string depthTopic = "/camera/depth/image_raw";
    private ROSConnection ros;

    void Start()
    {
        ros = ROSConnection.instance;
        ros.RegisterPublisher<ImageMsg>(colorTopic);
        ros.RegisterPublisher<ImageMsg>(depthTopic);

        renderTexture = new RenderTexture(1280, 720, 24,
            RenderTextureFormat.ARGB32);
        realSenseCamera.targetTexture = renderTexture;
        realSenseCamera.depthTextureMode = DepthTextureMode.Depth;

        colorTexture2D = new Texture2D(1280, 720,
            TextureFormat.RGB24, false);
        depthTexture2D = new Texture2D(1280, 720,
            TextureFormat.R8, false);
    }

    void Update()
    {
```

```
        RenderTexture.active = renderTexture;
        colorTexture2D.ReadPixels(new Rect(0, 0, 1280, 720), 0, 0);
        colorTexture2D.Apply();

        StartCoroutine(CaptureDepth());

        if (Time.time >= nextPublishTime)
        {
            PublishRGBD();
            nextPublishTime = Time.time + publishInterval;
        }
    }

    private void PublishRGBD()
    {
        ImageMsg colorMsg = new ImageMsg
        {
            header = new StdMsgs.HeaderMsg {
                frame_id = "camera_color_frame" },
            height = (uint)colorTexture2D.height,
            width = (uint)colorTexture2D.width,
            encoding = "rgb8",
            step = (uint)(colorTexture2D.width * 3),
            data = colorTexture2D.GetRawTextureData()
        };

        ImageMsg depthMsg = new ImageMsg
        {
            header = new StdMsgs.HeaderMsg {
                frame_id = "camera_depth_frame" },
            height = (uint)depthTexture2D.height,
            width = (uint)depthTexture2D.width,
            encoding = "mono8",
            step = (uint)(depthTexture2D.width * 1),
            data = depthTexture2D.GetRawTextureData()
        };
```

```
        ros.Publish(colorTopic, colorMsg);
        ros.Publish(depthTopic, depthMsg);
    }

    private IEnumerator CaptureDepth()
    {
        yield return new WaitForEndOfFrame();

        RenderTexture depthRT = RenderTexture.GetTemporary(
            1280, 720, 0, RenderTextureFormat.ARGBFloat);
        Material depthMat = new Material(
            Shader.Find("Hidden/Internal-DepthNormalsTexture"));

        Graphics.Blit(null, depthRT, depthMat);
        RenderTexture.active = depthRT;

        Texture2D tempDepth = new Texture2D(1280, 720,
            TextureFormat.RGBAFloat, false);
        tempDepth.ReadPixels(new Rect(0, 0, 1280, 720), 0, 0);
        tempDepth.Apply();

        for (int y = 0; y < depthTexture2D.height; y++)
        {
            for (int x = 0; x < depthTexture2D.width; x++)
            {
                float d = tempDepth.GetPixel(x, y).a;
                depthTexture2D.SetPixel(x, y, new Color(d, 0f, 0f));
            }
        }
        depthTexture2D.Apply();

        RenderTexture.ReleaseTemporary(depthRT);
        Destroy(tempDepth);
    }
}
```

Lo script `RealSenseSimulator.cs`, scritto in C#, è progettato per simulare il comporta-

mento di una telecamera RealSense all'interno di Unity, con la capacità di acquisire immagini a colori e mappe di profondità, e di pubblicarle su ROS2 utilizzando il pacchetto `Unity.Robotics.ROSTCPConnector`.
Lo script è associato ad un oggetto Unity contenente una `Camera`, il cui output visivo viene catturato e trasformato in messaggi `sensor_msgs/Image` compatibili con ROS2. Il comportamento dello script è articolato nei seguenti punti principali:

- **Inizializzazione:** Nella funzione `Start()`, viene creato un `RenderTexture` con risoluzione 1280×720 e formato ARGB32. Questo buffer viene assegnato come output della telecamera per la cattura dei frame a colori. Inoltre, viene abilitato il `DepthTextureMode.Depth` per permettere la generazione automatica della mappa di profondità. Vengono quindi inizializzate due texture 2D: una per il colore (`RGB24`) e una per la profondità (`R8`).
- **Integrazione con ROS 2:** Viene ottenuta l'istanza di `ROSConnection`, alla quale sono registrati due publisher per i topic `/camera/color/image_raw` e `/camera/depth/image_raw`, entrambi configurati per inviare messaggi di tipo `ImageMsg`.

All'interno della funzione `Update()`, chiamata ad ogni frame da Unity, avvengono le seguenti operazioni:

1. Il contenuto del `RenderTexture` attivo viene letto nella `colorTexture2D` mediante `ReadPixels`, catturando il frame RGB.
2. Viene avviata la coroutine `CaptureDepth()`, che esegue in modo asincrono la cattura della profondità.
3. Se è trascorso l'intervallo di pubblicazione definito (`publishInterval`), vengono pubblicati i dati RGB e di profondità tramite la funzione `PublishRGBD()`.

Il metodo `PublishRGBD()` crea due oggetti `ImageMsg`, uno per l'immagine RGB e uno per la profondità, impostando opportunamente il frame di riferimento, risoluzione, encoding e passo (`step`). I dati grezzi delle texture vengono letti tramite `GetRawTextureData()` e assegnati ai rispettivi messaggi. Infine, i messaggi vengono pubblicati sui topic ROS specificati.
La funzione `CaptureDepth()` è una coroutine che utilizza un `RenderTexture` temporaneo in formato `ARGBFloat` per acquisire la mappa di profondità. Questo viene ottenuto applicando un materiale con shader interno `Hidden/Internal-DepthNormalsTexture`, che consente di generare una texture contenente informazioni di profondità.
La profondità viene letta in una texture temporanea in formato `RGBAFloat`. Per ciascun pixel, il canale alfa rappresenta la profondità normalizzata, che viene poi convertita in un'immagine a canale singolo scrivendo il valore nel canale rosso della `depthTexture2D`. La texture risultante viene infine utilizzata per la pubblicazione ROS.
Va notato che la conversione della profondità pixel-per-pixel e la distruzione dinamica delle texture temporanee possono introdurre un overhead computazionale significativo, specialmente in scenari ad alta frequenza di pubblicazione. In contesti più esigenti, si potrebbe considerare l'ottimizzazione attraverso l'elaborazione GPU-side o tecniche di bufferizzazione.

2.7.4 Subscriber ROS2

Listing 2.3: Subscriber RGB-D ROS2 con salvataggio video

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
import cv2

class RGBDSubscriber(Node):
    def __init__(self):
        super().__init__('rgbd_subscriber')
        self.bridge = CvBridge()
        # File video di output
        fourcc = cv2.VideoWriter_fourcc(*'XVID')
        width, height = 1280, 720
        fps = 10.0  # 10 fps in linea con Unity
        self.rgb_writer = cv2.VideoWriter('output_rgb.avi', fourcc, fps, (width,
        self.depth_writer = cv2.VideoWriter('output_depth.avi', fourcc, fps, (wi

        if not self.rgb_writer.isOpened():
            self.get_logger().error('Impossibile aprire output_rgb.avi')
        if not self.depth_writer.isOpened():
            self.get_logger().error('Impossibile aprire output_depth.avi')

        #Subscriber per colore e profondita
        self.create_subscription(Image, '/camera/color/image_raw', self.color_cal
        self.create_subscription(Image, '/camera/depth/image_raw', self.depth_ca

        self.shutdown_timer = self.create_timer(
            60.0,                  # periodo in secondi
            self.shutdown_callback  # callback di chiusura
        )

    def color_callback(self, msg: Image):
        try:
            # Converte in BGR8 (OpenCV usa BGR)
            cv_image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='rgb8')
            cv_image = cv2.cvtColor(cv_image, cv2.COLOR_RGB2BGR)

            # Ruota di 180 gradi
            rotated = cv2.rotate(cv_image, cv2.ROTATE_180)
            cv2.imshow('Color View', rotated)
```

```
            # Scrive sul video
            self.rgb_writer.write(rotated)

        except CvBridgeError as e:
            self.get_logger().error(f'CVBridge Error: {e}')

        cv2.waitKey(1)

    def depth_callback(self, msg: Image):
        try:
            # Converte direttamente in gray-scale 8-bit
            gray = self.bridge.imgmsg_to_cv2(msg, desired_encoding='mono8')

            # Visualizza la depth
            cv2.imshow('Depth View', gray)

            # Scrive nel video
            self.depth_writer.write(gray)

        except CvBridgeError as e:
            self.get_logger().error(f'CVBridge Error (Depth): {e}')

        cv2.waitKey(1)

    def destroy_node(self):
        super().destroy_node()

    def shutdown_callback(self):
        self.get_logger().info('Timeout raggiunto: fermo acquisizione e chiudo fi
        self.rgb_writer.release()
        self.depth_writer.release()
        cv2.destroyAllWindows()
        self.shutdown_timer.cancel()
        self.destroy_node()
        rclpy.shutdown()

def main(args=None):
    rclpy.init(args=args)
    node = RGBDSubscriber()
    rclpy.spin(node)

if __name__ == '__main__':
    main()
```

Il nodo `RGBDSubscriber` in ambiente ROS2 ha il compito di ricevere i dati RGB e di profon-

dità inviati su due topic differenti dal Publisher e di trasformarli in formato video. Il nodo è implementato in Python e fa uso delle librerie `rclpy` per la comunicazione con ROS 2, `sensor_msgs.msg.Image` per la gestione dei messaggi di immagine, `cv_bridge` per la conversione tra formati ROS e strutture compatibili con OpenCV, e `opencv-python` per l'elaborazione e visualizzazione delle immagini.
Nel costruttore della classe `RGBDSubscriber`, viene inizializzato un oggetto `CvBridge` per la conversione tra messaggi ROS e immagini OpenCV. Successivamente, vengono creati due oggetti `cv2.VideoWriter` per la registrazione dei video RGB e di profondità, utilizzando il codec `XVID`, una risoluzione di 1280×720 pixel e una frequenza di 10 fps, in linea con la frequenza di pubblicazione del simulatore Unity.
Il nodo si sottoscrive ai seguenti topic:

- `/camera/color/image_raw` per le immagini RGB;
- `/camera/depth/image_raw` per le immagini di profondità.

Viene inoltre impostato un *timer* di spegnimento automatico a 60 secondi, che richiama una procedura di chiusura controllata.
Il metodo `color_callback` viene eseguito ad ogni ricezione di un messaggio RGB. L'immagine viene convertita nel formato `BGR8`, ruotata di 180° per correggere l'orientamento, visualizzata in una finestra OpenCV denominata `"Color View"` e registrata nel file `output_rgb.avi`. La chiamata a `cv2.waitKey(1)` permette l'aggiornamento dell'interfaccia grafica di OpenCV.
In modo analogo, il metodo `depth_callback` converte il messaggio ROS nel formato `mono8` (scala di grigi a 8 bit), visualizza il risultato nella finestra `"Depth View"` e lo registra nel file `output_depth.avi`. Anche in questo caso, `cv2.waitKey(1)` consente la corretta gestione delle finestre OpenCV.
Trascorsi 60 secondi, il metodo `shutdown_callback` viene eseguito automaticamente. Esso:

- stampa un messaggio di log;
- rilascia i `VideoWriter` per completare i file video;
- chiude tutte le finestre di OpenCV;
- cancella il timer ed esegue la distruzione del nodo;
- richiama `rclpy.shutdown()` per arrestare ROS 2.

La funzione `main()` inizializza l'ambiente ROS2, crea un'istanza del nodo e mantiene attiva l'elaborazione degli eventi attraverso `rclpy.spin()`, fino alla terminazione del nodo stesso.

Tramite quindi il Publisher di Unity e il Subscriber ROS2, è possibile acquisire e visualizzare i dati RGB e di profondità in tempo reale. I video risultanti possono essere utilizzati per analisi successive o per l'addestramento di modelli di visione artificiale.

Tuttavia, questo task ha mostrato alcune limitazioni. In particolare, la mappa di profondità generata non è stata in grado di fornire informazioni dettagliate e realistiche sulla scena. Questa discussione è approfondita alla sezione 3.5.

3 Analisi Preliminari e Problematiche Riscontrate

3.1 Problematiche sulle Compatibilità dei Software Utilizzati

Nel contesto del progetto, l'integrazione tra Unity e ROS2 rappresenta una sfida notevole dal punto di vista della compatibilità software. L'armonizzazione degli ambienti di simulazione grafica e del framework robotico comporta l'allineamento di versioni, librerie e tool, dove eventuali discrepanze possono causare malfunzionamenti, comportamenti inattesi e difficoltà nel mantenimento della comunicazione affidabile fra i sistemi. In generale, le principali problematiche riguardano:

- **Dipendenze e Librerie**: Ogni ambiente software si basa su set specifici di dipendenze e librerie. Una discrepanza nelle versioni o nelle configurazioni può portare a malfunzionamenti, crash o comportamenti inattesi.
- **Protocollo di Comunicazione**: L'interfacciamento tramite strumenti come il ROS-TCP Connector & Endpoint impone un rigoroso rispetto degli standard di serializzazione e deserializzazione dei messaggi. Differenze nelle versioni possono influenzare la compatibilità dei protocolli, rendendo critica la coerenza delle implementazioni.
- **Ambiente di Esecuzione e Simulazione**: La perfetta integrazione tra il motore fisico di Unity e il middleware ROS2 richiede una sincronizzazione accurata dei tempi di esecuzione, in modo da garantire una simulazione fluida e reattiva. Eventuali incompatibilità possono compromettere le performance e l'efficacia dei test simulativi.

Durante le fasi iniziali di sviluppo, il gruppo ha scelto di utilizzare ROS2 Jazzy, una distribuzione che, è supportata fino a maggio 2029, e rappresentava una versione più nuova e all'avanguardia. Tuttavia, in fase di integrazione con Unity sono emerse alcune criticità, in particolare durante i test si sono riscontrate carenze funzionali nelle librerie e nei tool di ROS2 Jazzy. Queste mancanze evidenziavano dei limiti, in particolare per quanto riguarda la robustezza delle comunicazioni e la gestione dei messaggi in ambienti complessi.
La migrazione verso ROS2 Humble è stata dettata dalla necessità di disporre di una versione più consolidata e con una maggiore compatibilità con gli strumenti attuali di Unity. ROS2 Humble, supportato fino a maggio 2027, ha offerto miglioramenti in termini di performance, stabilità e aggiornamenti, garantendo una maggiore robustezza nelle comunicazioni e nella gestione delle dipendenze. [12]
Chiaramente, con la scelta di passare a ROS2-Humble anche gli altri software utilizzati sono stati aggiornati, in particolare Gazebo è stato aggiornato alla versione Fortress, che ha mostrato una maggiore stabilità e compatibilità con ROS2 Humble. [7]
Con l'evolversi del progetto e l'avvicinarsi della scadenza del supporto per Humble, si dovrà valutare attentamente una futura migrazione verso ROS2 Jazzy. Passare a Jazzy in un secondo momento potrebbe permettere di sfruttare i miglioramenti continui offerti dalla nuova distribuzione, garantendo così una maggiore longevità e integrazione degli aggiornamenti nell'ecosistema software del progetto.

3.2 Compatibilità dei File 3D con Unity e Gazebo

Il formato **.obj** è stato testato con successo sia in Unity che in Gazebo, consentendo l'importazione corretta del modello della t-shirt in entrambi i software. Tuttavia, durante le simulazioni in Unity, l'applicazione del componente Cloth alla t-shirt ha evidenziato alcune difficoltà: il computer ha riscontrato problemi nel gestire correttamente i comandi impostati e, una volta avviata la simulazione, la t-shirt appariva deformata e non realistica, come è possibile vedere nella figura 3.1.

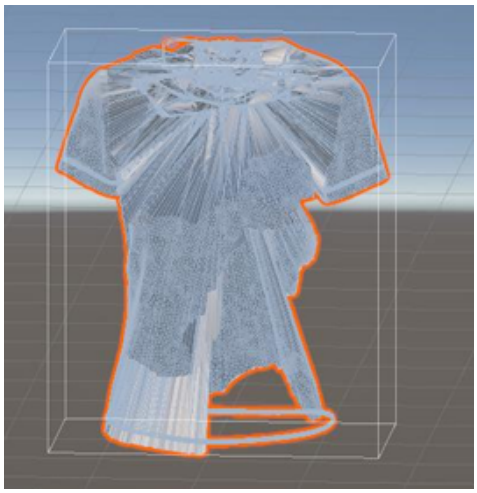

Figure 3.1: *Errore nell'applicazione del Cloth di Unity*

Per risolvere questo problema, l'oggetto è stato importato nel software Blender, dove è stato utilizzato il modificatore Decimate per ridurre il numero di poligoni della mesh. Diminuendo il parametro "Ratio" a un valore che ha portato la mesh a circa 5000 poligoni, è stato possibile reimportare l'oggetto in Unity e applicare correttamente il componente Cloth, ottenendo una simulazione più realistica.

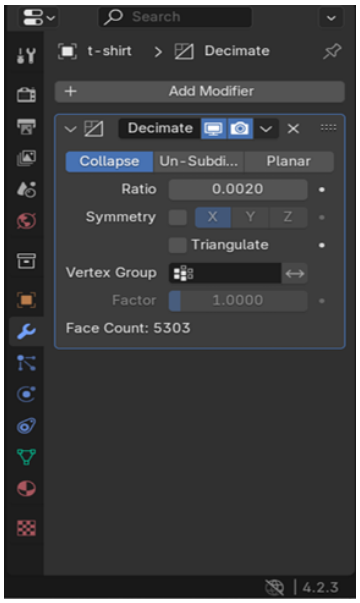

Figure 3.2: *Modifica della Mesh nel software Blender*

Il formato **.fbx** non è supportato da Gazebo; tuttavia, è compatibile con Unity e può essere convertito in altri formati, come .obj o .dae (Collada), utilizzando Blender. Seguendo lo stesso procedimento adottato per il file .obj, quindi riducendo il numero di poligoni delle mesh tramite il modificatore Decimate in Blender, la simulazione in Unity è stata eseguita correttamente.

Il formato **.dae** (Collada) è supportato da Gazebo e risulta essere ampiamente utilizzato in questo contesto. Importando i file .obj della t-shirt e della scarpa, nonché il file .fbx della t-shirt in Blender, è stato possibile esportarli nel formato .dae. Questa operazione ha permesso di caricare correttamente i vari modelli in Gazebo, facilitando la simulazione degli indumenti nel simulatore.

In sintesi, l'utilizzo combinato di Blender per l'ottimizzazione delle mesh e la conversione dei formati, insieme all'applicazione del componente Cloth in Unity, ha consentito di ottenere simulazioni più realistiche del comportamento dei tessuti, migliorando l'interazione tra il robot e gli indumenti nel contesto del progetto.

3.2.1 Componente Cloth e collisioni con altri materiali

Il Cloth di Unity oltre a realizzare una buona fisica per la maglietta è in grado di creare delle collisioni con altri materiali. In particolare, come è stato descritto nel capitolo precedente, si può inserire un oggetto Capsule o Sphere nel Cloth e quindi specificare alla maglietta quali e quanti oggetti dovranno avere un'interazione fisica opportuna durante la collisione. Tuttavia non è possibile far si che l'oggetto a cui è applicato il Cloth (in questo caso la maglietta) collida con un qualunque oggetto di qualunque forma geometrica. Infatti come anticipato è possibile utilizzare solo due tipi di oggetti 3D Unity: Capsule e Sphere. Questa restrizione rappresenta una problematica nel caso in cui il tessuto debba interagire con superfici piatte, angoli o forme geometriche più complesse, come un tavolo, una scatola o una struttura irregolare. Nel caso del progetto un'applicazione utile sarebbe quella di poter far interagire la maglietta con l'end

effector del robot che poi riesca ad appoggiarla su una superficie piana come un tavolo. Poiché non è possibile definire direttamente colliders di tipo Mesh o Box per il Cloth, è necessario trovare soluzioni alternative per ottenere un comportamento realistico nelle simulazioni.
Una soluzione consiste nel posizionare una griglia di piccole Sphere Colliders sulla superficie con cui il tessuto deve interagire. In questo modo, il tessuto percepirà la presenza di una superficie continua, pur utilizzando solo colliders sferici. Per implementare questa soluzione:

- Creare uno script che generi automaticamente una matrice di sfere sulla superficie desiderata.
- Assicurarsi che le sfere vengano generate prima che il Cloth venga configurato, in modo che possano essere assegnate correttamente ai "Sphere Colliders" della maglietta.
- Ottimizzare il numero e la dimensione delle sfere per ottenere una buona resa senza sovraccaricare il motore fisico.

3.3 Problemi di importazione del URDF del robot in Unity

Un altro problema riscontrato durante la fase di sviluppo del progetto, è nato durante la fase di importazione del file URDF del robot TM5-900 in Unity tramite lo strumento URDF-Importer. Come spiegato nella sezione 2.4.2, tramite questo package è possibile importare il robot in Unity con le varie meshes sia di visual che di collision in pochi e semplici click. Tuttavia è importante far notare un problema che è stato riscontrato durante l'importazione del file URDF del robot TM5-900. Infatti, una volta che il robot è stato importato per la prima volta, Unity creata automaticamente dei file .prefab all'interno della cartella "assets" e quindi sia dentro la cartella "visual" che "collision". In figura 3.3 e 3.4 vediamo rispettivamente le cartelle "visual" e "collision" una volta che il robot è stato importato, i file di colore rosa sono appunto i file che Unity genera automaticamente (.prefab).

Figure 3.3: *Cartella "visual" con i file .prefab generati automaticamente da Unity*

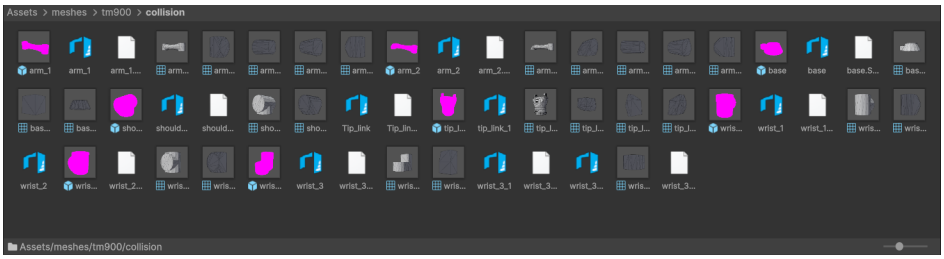

Figure 3.4: *Cartella "collision" con i file .prefab generati automaticamente da Unity*

Tuttavia, una volta che il robot veniva cancellato dalla scena e quindi lo si voleva reimportare questi file .prefab venivano sovrascritti e il robot non veniva più visualizzato correttamente. In particolare, in fase di importazione l'errore che si otteneva era il seguente:

Figure 3.5: *Errore durante l'importazione del file URDF del robot TM5-900*

Questo problema è stato risolto semplicemente eliminando i vari file .prefab all'interno della cartella "meshes" (sia in "visual" che in "collision").

3.4 Problemi di visualizzazione delle meshes in Gazebo e Unity

In questo progetto, sia su Gazebo che su Unity, si è utilizzato il file URDF per importare il robot TM5-900. Tuttavia, i due ambienti presentano differenze significative nell'interpretazione dei percorsi dei file che possono causare disallineamenti nella visualizzazione degli asset. Quella che segue è una soluzione pratica a un problema avvenuto durante l'importazione del file URDF del robot in Gazebo e Unity.

3.4.1 Problemi di visualizzazione delle meshes in Gazebo

Come già proposto nella sezione 2.6, è possibile importare il robot TM5-900 utilizzando il file URDF. Tuttavia, durante il caricamento del file in Gazebo, si è riscontrato un errore di visualizzazione delle meshes, che non venivano caricate correttamente. Questo problema è stato causato da un errore nei percorsi delle meshes all'interno del file URDF. In particolare, i percorsi delle meshes erano stati definiti in questo modo:

```
<link name="base_link">
<visual>
  <geometry>
    <mesh filename="package://tm_description/meshes/tm900/visual/Base.STL"/>
  </geometry>
  <material name="Grey">
    <color rgba="0.5 0.5 0.5 1.0"/>
  </material>
```

```
  </visual>
  <collision>
    <geometry>
      <mesh filename="package://tm_description/meshes/tm900/collision/base.STL"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="1.0"/>
    <insert_block name="origin"/>
    <inertia ixx="0.00110833289" ixy="0.0" ixz="0.0" iyy="0.00110833289" iyz="0.0" izz="0.
  </inertial>
</link>
```

tuttavia Gazebo non riusciva a riconoscere il path, così la soluzione è stata quella di inserire il path completo a partire dalla cartella "home" di Linux, come mostrato di seguito:

```
  <link name="base_link">
  <visual>
    <geometry>
      <mesh filename="/home/User/ros2_workspace/install/tm_description/share/tm_descriptio
    </geometry>
    <material name="Grey">
      <color rgba="0.5 0.5 0.5 1.0"/>
    </material>
  </visual>
  <collision>
    <geometry>
      <mesh filename="/home/User/ros2_workspace/install/tm_description/share/tm_descriptio
    </geometry>
  </collision>
  <inertial>
    <mass value="1.0"/>
    <insert_block name="origin"/>
    <inertia ixx="0.00110833289" ixy="0.0" ixz="0.0" iyy="0.00110833289" iyz="0.0" izz="0.
  </inertial>
</link>
```

3.4.2 Problemi di visualizzazione delle meshes in Unity

Analogamente si è riscontrato un problema di visualizzazione delle meshes in Unity, dove il robot non veniva caricato correttamente. Anche in questo caso, il problema era dovuto a un errore nei percorsi delle meshes all'interno del file URDF. La soluzione è stata quella di modificare i percorsi delle meshes nel file URDF come mostrato di seguito:

```
  <link name="base_link">
  <visual>
    <geometry>
```

```
      <mesh filename="package://meshes/tm900/collision/base.STL"/>
    </geometry>
    <material name="Grey">
      <color rgba="0.5 0.5 0.5 1.0"/>
    </material>
  </visual>
  <collision>
    <geometry>
      <mesh filename="package://meshes/tm900/collision/base.STL"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="1.0"/>
    <insert_block name="origin"/>
    <inertia ixx="0.00110833289" ixy="0.0" ixz="0.0" iyy="0.00110833289" iyz="0.0" izz="0.
  </inertial>
</link>
```

facendo attenzione a inserire nella cartella Assets di Unity la prima cartella definita dopo "package://", in questo caso "meshes". In questo modo Unity riesce a riconoscere il path e caricare correttamente le meshes del robot.

3.5 Problemi con l'acquisizione della profondità tramite la camera virtuale

4 Struttura del workspace ROS2

In questo capitolo viene presentata la struttura del workspace ROS2, con particolare attenzione alla disposizione dei pacchetti e dei file necessari per l'integrazione con Unity e la simulazione del robot. Come sotto riportato, i package sono evidenziati in rosso mentre i file Python sono evidenziati in blu.

```
Dynamics/
    |-- ros2_humble/
    |   |-- src/
    |   |   |-- moveit_config_tm5_900/
    |   |   |   |-- config/
    |   |   |   |-- launch/
    |   |   |   |   |-- demo.launch.py
    |   |   |   |   |-- move_group.launch.py
    |   |   |   |   |-- setup_assistant.launch.py
    |   |   |   |-- .setup_assistant
    |   |   |   |-- CMakeLists.txt
    |   |   |   `-- package.xml
    |   |   |-- my_ros2_package/
    |   |   |   |-- my_ros2_package/
    |   |   |   |   |-- build/
    |   |   |   |   |-- install/
    |   |   |   |   |-- log/
    |   |   |   |   |-- moveit_controller.py
    |   |   |   |   `-- pose_listener.py
    |   |   |   |-- resource/
    |   |   |   |-- test/
    |   |   |   |-- setup.cfg
    |   |   |   `-- setup.py
    |   |   |-- tm_description/
    |   |   |   |-- config/
    |   |   |   |-- launch/
    |   |   |   |-- meshes/
    |   |   |   |-- urdf/
    |   |   |   |-- CMakeLists.txt
    |   |   |   |-- package.xml
    |   |   `-- tm_moveit_config/
    |   |       |-- ...
    |   |-- install/
    |   |-- build/
    |   `-- log/
```

Nel diagramma sono state riportate solo le cartelle e i file considerati più rilevanti per la comprensione dell'architettura del workspace; la repository completa è disponibile a [13].

Si segnala, inoltre, che il package `tm_moveit_config` non è stato ulteriormente implementato, in quanto è stato creato automaticamente tramite il MoveIt Setup Assistant e presenta la stessa struttura di `moveit_config_tm5_900`, spiegata nel dettaglio alla sezione 2.2.3.

Questi due package influenzano direttamente il funzionamento del robot e sono stati creati per gestire la cinematica e la dinamica del braccio robotico. Tramite il comando

```
ros2 launch moveit_config_tm5_900 setup_assistant.launch.py
```

oppure

```
ros2 launch tm_moveit_config setup_assistant.launch.py
```

per i due package, si avvierà il Moveit Setup Assistant con il quale è possibile variare le pose del robot, le traiettorie e i giunti influenzati da esse. Il file `demo.launch.py` permette di avviare il MoveIt2 demo, che consente di testare le funzionalità del robot in un ambiente simulato (approfondimento alla sezione 2.2.3). Il file `move_group.launch.py` è il nodo principale che gestisce la comunicazione tra ROS2 e MoveIt2, mentre il file `setup_assistant.launch.py`, come già accenato, è utilizzato per configurare il robot e le sue proprietà cinematiche.
La cartella `my_ros2_package` contiene gli script Python che, una volta stabilita la connessione e avviati i controllori di Moveit, inviano i comandi da ROS2 a Unity, in particolare il file `moveit_controller.py` che si occupa di simulare la movimentazione del robot TM5-900 in Unity dalla posa di "Home" a quella di "Work" definite nel Moveit Setup Assistant. Questo file è fondamentale per l'integrazione tra ROS2 e Unity, in quanto consente di controllare il robot direttamente dall'interfaccia grafica di Unity, facilitando la simulazione e il test delle traiettorie.
La cartella `tm_description` contiene i file URDF e i modelli 3D necessari per rappresentare il robot in Gazebo e Unity. In questo package è importante che siano presenti le meshes all'interno della cartella `meshes`, sia di visualizzazione che di collisione, e il file URDF all'interno della cartella `urdf`. In particolare il file URDF è possibile generarlo a partire dal file xacro tramite il comando:

```
ros2 run xacro xacro nome_file.xacro > nome_file.urdf
```

Il file URDF è fondamentale per la rappresentazione del robot in Gazebo e Unity, in quanto definisce la struttura e le proprietà fisiche del robot, inclusi i link, i giunti e le collisioni. Inoltre, il file URDF deve essere correttamente configurato per garantire che il robot venga visualizzato e simulato correttamente in entrambi gli ambienti e quindi che i path relativi alle meshes siano corretti (problematiche riguardo ai path delle meshes nei vari software alla sezione 3.4).

5 Test Effettuati e Risultati Ottenuti

In questo capitolo si vuole guidare il lettore attraverso i test effettuati e i risultati ottenuti durante lo sviluppo del progetto. I test sono stati suddivisi in tre sezioni principali: test di comunicazione tra ROS2 e Unity, test di simulazione del movimento del robot in Unity e test di integrazione della camera. Ogni sezione presenta una descrizione dettagliata dei test effettuati, dei risultati ottenuti e delle eventuali problematiche riscontrate.

5.1 Test di comunicazione tra ROS2 e Unity

5.2 Test di simulazione del movimento del robot in Unity

5.3 Test di integrazione della camera

6 Conclusioni

6.1 Applicazione finale

A Appendice1

Se necessario ricorrete alle appendici per spiegare le parti "di contorno" dell'attività svolta e/o ciò che non riuscite ad inserire nello schema generale dei capitoli della relazione (es acquisizione dei dati con Matlab).

Bibliografia

[1] "Video tutorial per installazione di ubuntu." https://www.youtube.com/watch?v=bOwsWabST-0.

[2] "Documentazione di ros2." https://docs.ros.org/en/humble/index.html.

[3] "Documentazione di moveit." https://moveit.picknik.ai.

[4] "Installazione di moveit." https://moveit.picknik.ai/humble/doc/tutorials/getting_started/getting_started.html.

[5] "Documentazione di unity." https://docs.unity.com/.

[6] "Manuale del componente cloth di unity." https://docs.unity3d.com/Manual/class-Cloth.html.

[7] "Compatibilità gazebo-ros2." https://gazebosim.org/docs/harmonic/ros_installation/#summary-of-compatible-ros-and-gazebo-combinations.

[8] "Repository per il tcp endpoint di ros." https://github.com/Unity-Technologies/ROS-TCP-Endpoint.git.

[9] "Tutorial per test della connessione tra unity e ros2." https://github.com/Unity-Technologies/Unity-Robotics-Hub/blob/main/tutorials/ros_unity_integration/publisher.md.

[10] "Repository del urdf-importer." https://github.com/Unity-Technologies/URDF-Importer.

[11] "Tutorial per il setup assistant di moveit." https://docs.ros.org/en/kinetic/api/moveit_tutorials/html/doc/setup_assistant/setup_assistant_tutorial.html.

[12] "Documentazione per il supporto delle versioni di ros." https://docs.ros.org/en/humble/Releases.html.

[13] "Repository di ros2." https://github.com/Piz01/Dynamics.git.