UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento di
Ingegneria Gestionale, dell'Informazione e della Produzione

Corso di laurea in
Ingegneria informatica

Classe n. L-8

Guida alla simulazione di robot mobili in ambiente Unity – ROS 2

Relatore:
Chiar.mo Prof. Davide Brugali

Candidati:	Matricola n.
Sean Pellegrinelli	*1065868*
Nico Pellegrinelli	*1065869*

Anno Accademico
2021/2022

Indice

Elenco delle figure

Capitolo 1

Introduzione

Nell'ambito della robotica è fondamentale poter simulare i robot e l'ambiente con cui entrano in contatto, così da poter testare in sicurezza nuove funzionalità o modifiche prima di applicarle al robot reale. Per questo è necessario disporre di un ambiente di simulazione robusto che sia in grado di modellare nel modo più fedele possibile i robot, il loro comportamento e gli ambienti in cui operano.

Sono presenti sul mercato diversi software per simulazioni robotiche, tra i quali compare Unity, un popolare motore grafico utilizzato anche nell'ambito della robotica.

L'obiettivo principale di questo lavoro è analizzare gli strumenti che Unity mette a disposizione per creare simulazioni robotiche, in modo da comprenderne le potenzialità ed i limiti. Sono di particolare interesse i seguenti fattori:

- l'integrazione del simulatore con il framework ROS 2 (la seconda versione del Robot Operating System (ROS)) e di conseguenza con il middleware DDS, utilizzato da ROS 2 per la comunicazione tra componenti;
- la semplicità di realizzazione di nuovi scenari di simulazione;
- la varietà di sensori disponibili e la semplicità di creazione di nuovi sensori;
- l'accuratezza della simulazione.

Per valutare questi aspetti è stata realizzata una simulazione in Unity di un semplice robot mobile su due ruote, attraverso la quale sono state sperimentate le varie opzioni e funzionalità messe a disposizione dal simulatore.

Questo lavoro è anche utile come introduzione e guida all'utilizzo di Unity come strumento per la simulazione robotica, soprattutto per quanto riguarda robot mobili su due ruote.

Capitolo 2

Tecnologie utilizzate

Il lavoro è stato svolto sul sistema operativo Ubuntu 20.04 LTS Focal Fossa. Il software di simulazione utilizzato è Unity, che implementa meccanismi di comunicazione basati su ROS 2. ROS 2 è un framework open-source per lo sviluppo di applicazioni legate alla robotica che utilizza come middleware lo standard DDS. Per far comunicare una simulazione Unity con componenti esterne al simulatore è stato utilizzato sia ROS 2 che eProsima Fast DDS 2.2.0, un'implementazione in C++ dello standard DDS.

2.1 Unity

Sviluppato e mantenuto da Unity Technologies a partire dal 2005, Unity è un motore grafico, o game engine, multipiattaforma. Un motore grafico è un framework che comprende diversi componenti software fondamentali per lo sviluppo di videogiochi, come il motore fisico, o physics engine, un ambiente di sviluppo grafico e diverse librerie che ne facilitano e ottimizzano lo sviluppo.

È attualmente utilizzato a livello internazionale principalmente per lo sviluppo di videogiochi ma si presta anche allo sviluppo di contenuti multimediali non appartenenti al mondo videoludico, quali visualizzazioni architettoniche e la produzione di film. Unity Technologies ha anche investito nell'ambito della simulazione robotica, espandendo il già vasto universo le-

gato al motore grafico con plug-in, tutorial e materiale per la simulazione di robot.

Unity permette di sviluppare applicazioni sia in due dimensioni (2D) che in tre dimensioni (3D), nel primo caso il motore fisico utilizzato è Box2D, nel secondo Nvidia PhysX. Sono questi componenti software che si occupano di simulare la fisica all'interno di quello che è, in questo caso, l'ambiente di simulazione. I physics engine si occupano quindi di gestire interazioni come collisioni, accelerazioni e gravità.

Unity mette a disposizione l'applicazione Unity Hub, dalla quale è possibile accedere all'ecosistema di Unity, permette infatti di gestire i progetti Unity e di installare e gestire le versioni dell'Editor, cioè l'ambiente grafico di sviluppo che consente la gestione di tutti gli aspetti della simulazione.

Un progetto in Unity è una collezione di cartelle e file. Tra questi file sono fondamentali quelli con estensione unity, che descrivono le scene. Per lavorare in Unity è necessario creare una scena. Una scena è un insieme di oggetti, chiamati *GameObject*. Un *GameObject* è un "contenitore" di *component*. I *component* sono gli elementi che implementano le funzionalità dell'oggetto, come le collisioni con altri oggetti o la forma dell'oggetto stesso. L'Editor mette a disposizione, oltre all'oggetto vuoto, ovvero privo di *component*, altri oggetti con alcuni *component* già aggiunti, come il cubo, il piano e la camera. Gli oggetti possono essere organizzati in una struttura gerarchica, ad esempio un robot può essere costituito da una gerarchia di oggetti in cui compaiono, tra gli altri, i sensori e le ruote. Ogni *GameObject* è caratterizzato dalla sua *Transform*, ovvero tre triplette (X, Y e Z) di informazioni riguardanti la posizione (*Position*), la rotazione (*Rotation*) e le dimensioni (*Scale*) dell'oggetto stesso. La posizione e la rotazione sono relative all'oggetto padre nella gerarchia. Nel caso in cui l'oggetto non abbia un padre, le informazioni di *Transform* sono relative al centro della scena.

I *component* possono anche essere degli script. Gli script vengono utilizzati sia per far interagire i vari oggetti tra di loro all'interno della scena, sia per far comunicare Unity con l'esterno. Unity supporta nativamente due linguaggi di programmazione: C# e UnityScript. Quest'ultimo è un linguaggio progettato specificamente per Unity ed influenzato da JavaScript. Per que-

sto lavoro è stato utilizzato C#. Ogni script C# in Unity implementa una classe che eredita dalla classe base definita all'interno di Unity chiamata MonoBehaviour. Questa prevede metodi fondamentali come Start(), chiamato quando lo script viene abilitato, e Update(), chiamato ad ogni frame.

I file utilizzabili nel progetto sono contenuti nella cartella Assets, che può essere esplorata direttamente attraverso l'Editor, nella finestra Project.

Nella Figura 2.1 si vede un esempio di una scena nell'Editor di Unity, si noti la gerarchia degli oggetti, chiamata *Hierarchy*, sul lato sinistro dell'Editor. Nella finestra sulla destra, chiamata *Inspector*, si trovano invece le informazioni relative a *Transform* e i *component* relativi al *GameObject* Cube, tra i quali si individua uno script C#.

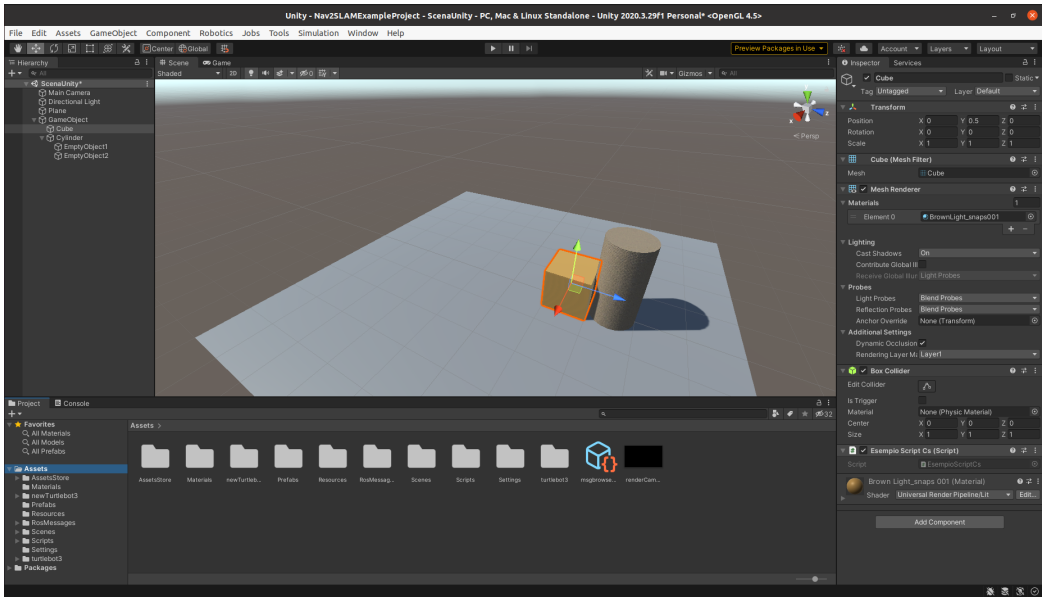

Figura 2.1: Esempio di una scena nell'Editor Unity

2.2 ROS 2

Robot Operating System (ROS) è un insieme di librerie e strumenti open source per lo sviluppo di applicazioni legate al mondo della robotica rilasciato per la prima volta nel 2007. Successivamente, nel 2017, è stata rilasciata

la seconda versione di ROS, ovvero ROS 2, che introduce diverse migliorie volte ad adeguarsi ai cambiamenti avvenuti nell'ambito della robotica. Unity fornisce strumenti per la comunicazione sia con ROS che con ROS 2. Per questo lavoro si è utilizzato solamente ROS 2, in particolare la distribuzione Foxy Fitzroy [2]. Gli aspetti fondamentali di ROS 2 vengono descritti di seguito.

Un sistema robotico basato su ROS 2 è composto da diversi nodi, ovvero processi computazionali in grado di comunicare tra loro. Ogni nodo del sistema è, o almeno dovrebbe essere, responsabile di un particolare compito, come ad esempio il controllo delle ruote del robot o la gestione di un sensore laser. In ROS 2 un singolo eseguibile può contenere più nodi. Per poter scrivere programmi che implementano nodi ROS 2 è necessario utilizzare delle apposite API (chiamate client libraries), disponibili in diversi linguaggi di programmazione. Le uniche client libraries sviluppate direttamente dal team di ROS 2 sono rclcpp (per codice C++) e rclpy (per codice Python), le altre sono scritte dalla community di ROS 2. Diversi nodi scritti con linguaggi differenti possono comunque comunicare tra loro.

I nodi possono scambiare dati attraverso tre meccanismi:

- topic: i dati vengono scambiati attraverso il paradigma di comunicazione publisher-subscriber;
- servizi: la comunicazione è basata sul modello call-and-response;
- azioni: il concetto è simile a quello dei servizi, ma più articolato, vengono sfruttati infatti sia i servizi che i topic. Un nodo "action client" invia un *goal* ad un nodo "action server", che lo riconosce. Successivamente il nodo "action server" invia, attraverso un topic, uno stream di dati di feedback. Infine, il risultato viene inviato da "action server" verso "action client".

Inoltre, ROS 2 introduce il concetto di parametri, ovvero valori relativi a ciascun nodo utilizzati per configurare il nodo stesso durante l'inizializzazione o a runtime, senza modificarne il codice.

In questo lavoro è stato utilizzato solamente il meccanismo dei topic, rappresentato nella Figura 2.2, che viene quindi descritto più nel dettaglio. Ciascun nodo può pubblicare e ricevere messaggi su un qualsiasi numero di

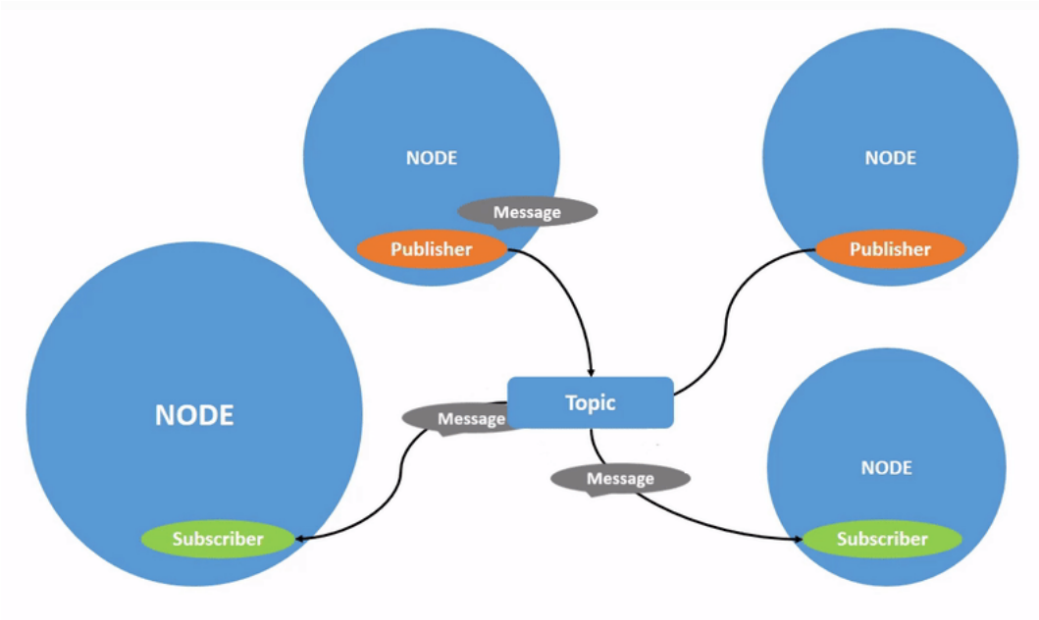

Figura 2.2: Esempio di comunicazione in ROS 2 tramite topic

topic. Per ricevere messaggi su un topic è necessario che il nodo abbia attiva un'entità subscriber. Analogamente, per inviare messaggi è necessaria un'entità publisher. I nodi che hanno attivo un subscriber di un determinato topic riceveranno tutti i messaggi inviati da un qualsiasi publisher che abbia pubblicato su tale topic. Ciascun topic è caratterizzato da un nome e da un tipo (topic type). Il tipo non è altro che la struttura dei dati contenuti nei messaggi scambiati sul topic. Per poter comunicare correttamente, tutti i publisher e subscriber di un dato topic devono utilizzare lo stesso topic type. ROS 2 mette a disposizione diversi tipi di messaggi per i casi più frequenti, ma si possono comunque definire tipi personalizzati se necessario. La comuncazione tramite topic sfrutta anche il concetto di QoS. I Quality of Service (QoS) sono dei parametri configurabili che controllano le caratteristiche della comunicazione. Un esempio è *Reliability*, che può essere impostato a *Best effort* o *Reliable*. Nel primo caso il messaggio potrebbe andare perso, nel

secondo è garantito che il messaggio venga trasmesso, se necessario provando più volte. Ciascun publisher o subscriber può impostare i propri QoS, ma per far sì che la comunicazione avvenga con successo essi devono essere compatibili, ovvero i QoS del subscriber devono essere uguali o meno "stringenti" di quelli del publisher. Per esempio nel caso di *Reliability* la compatibilità è la seguente:

Publisher	Subscriber	Compatibilità
Best effort	Best effort	Sì
Best effort	Reliable	No
Reliable	Best effort	Sì
Reliable	Reliable	Sì

ROS 2 mette a disposizione anche un framework per interfaccia grafica, chiamato rqt, che implementa diversi tool ed interfaccie sotto forma di plugin. Ad esempio il plugin rqt_graph permette di visualizzare un grafico contenente tutti i nodi, i topic e le connessioni tra di essi attivi nel sistema.

Un altro tool disponibile per ROS 2 è rviz2, il quale permette di visualizzare, se possibile, i messaggi pubblicati sui topic sotto forma grafica in un ambiente tridimensionale. Permette anche di pubblicare messaggi su determinati topic.

2.3 eProsima Fast DDS

eProsima fast DDS è un'implementazione C++ dello standard DDS (Data Distribution Service), emanato dall'Object Management Group (OMG). DDS definisce un middleware per comunicazione di tipo publisher-subscriber rivolto a sitemi real-time ed embedded. In particolare, eProsima fast DDS è stato scelto come middleware di deafult di ROS 2. Per questo lavoro è stata utilizzata la versione 2.2.0 di eProsima Fast DDS [4].

La comunicazione attraverso eProsima Fast DDS è molto simile a quella descritta per il meccanismo dei topic in ROS 2 che, come già menzionato, si basa proprio su quest'implementazione di DDS. Sono infatti presenti gli

stessi concetti di topic, topic type, QoS, publisher e subscriber, anche se leggermente più articolati. I Publisher, in questo contesto, sono le entità che creano e configurano i DataWriter, ovvero le entità che pubblicano i messaggi su un determinato topic (ciascun DataWriter può pubblicare su un solo topic). In modo del tutto analogo i Subscriber creano e configurano i DataReader, che leggono i messaggi pubblicati su un topic. I Publisher e Subscriber sono contenuti nei DomainParticipant (concettualmente simili ai nodi in ROS 2). Tutti i DomainParticipant che scambiano tra loro messaggi sui medesimi topic appartengono allo stesso DDS Domain. Ogni entità descritta, fatta eccezione per il DDS Domain, ha i propri QoS. Questa struttura è rappresentata nella Figura 2.3.

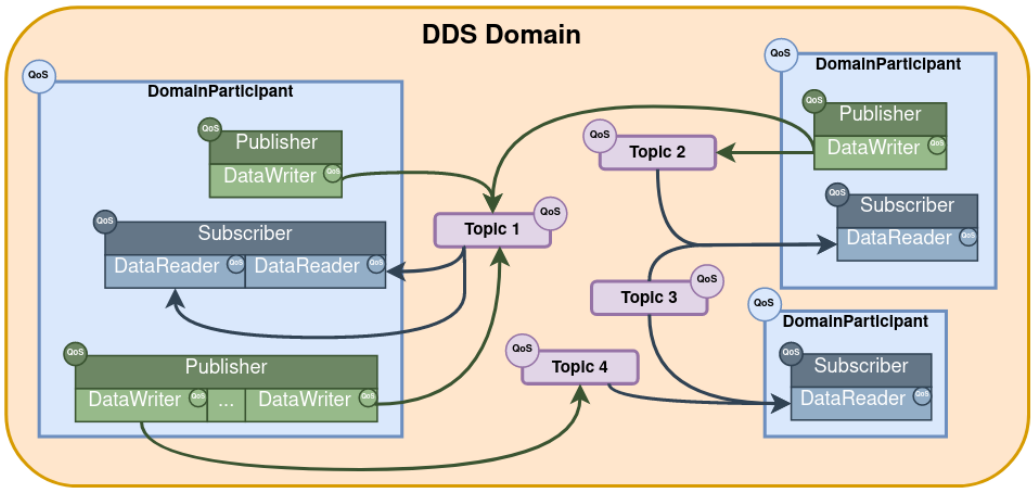

Figura 2.3: Organizzazione delle entità in un DDS Domain

Per poter comunicare correttamente su un topic, tutte le entità coinvolte devono utilizzare la stessa struttura per i dati scambiati. eProsima Fast DDS mette a disposizione un tool, chiamato Fast DDS-Gen. Questo strumento genera, a partire da un file IDL (Interface Definition Language), il codice sorgente necessario a definire il tipo di dato, che può di conseguenza essere scambiato su un topic.

Capitolo 3

Integrazione ROS 2 - Unity

Prima di entrare nella descrizione dettagliata dell'utilizzo di Unity come simulatore per applicazioni robotiche è utile comprendere come avviene l'integrazione tra il motore grafico Unity ed il framework ROS 2.

Unity Technologies mette a disposizione strumenti, documentazione, risorse e tutorial per la simulazione robotica in Unity attraverso una serie di repository GitHub. La principale è *Unity Robotics Hub* [1]. Il file README.md di questa repository contiene anche i riferimenti a *ROS TCP Connector* e *ROS TCP Endpoint*, due repository contenenti il materiale necessario all'integrazione tra ROS 2 e Unity.

Il materiale in ognuna di queste repository si occupa di un lato della comunicazione che si instaura tra il lato Unity (*ROS TCP Connector*) e il lato ROS 2 (*ROS TCP Endpoint*). Lo scambio di informazioni tra i due ambienti avviene tramite scambio di messaggi serializzati sfruttando i protocolli TCP/IP. In *ROS TCP Connector* viene implementato il lato client e in *ROS TCP Endpoint* il lato server.

3.1 ROS TCP Endpoint

Il lato server della comunicazione TCP/IP è implementato nei file contenuti nella repository *ROS TCP Endpoint*. Il linguaggio di programmazione utilizzato è Python.

Il server TCP/IP vero e proprio viene implementato nella classe TcpServer, contenuta nel file server.py. Questo importa una serie di file, anch'essi contenuti in *ROS TCP Endpoint*, per gestire publisher, subscriber e servizi dell'ambiente ROS 2 ed altre funzioni di supporto. Questi file utilizzano la libreria python rclpy, messa a disposizione da ROS 2.

Alla richiesta, da parte del client, di creare un nuovo publisher su un determinato topic, il server crea un'istanza della classe RosPublisher, la quale crea effettivamente un nodo ROS 2. Per ogni publisher viene creato un nodo ROS 2 dedicato. Quando il client comunica al server un messaggio da publicare su un topic, il server comunica i dati all'istanza corretta di RosPublisher, che si occupa di pubblicare effettivamente il messaggio sul topic. Analogamente, la richiesta di creare un nuovo subscriber da parte del client viene gestita creando una nuova istanza della classe RosSubscriber, che crea un nuovo nodo ROS 2 contenente un subscriber sul topic desiderato. Ogni qualvolta questo subscriber riceve un messaggio sul topic, lo passa al server che lo serializza e lo trasmette al client.

3.2 ROS TCP Connector

I programmi che si occupano del lato Unity della comunicazione, contenuti nella repository *ROS TCP Connector*, sono scritti in C#, nativamente supportato da Unity.

Il componente principale è la classe ROSConnection, implementata nello script ROSConnection.cs. L'istanza di questa classe assume il ruolo di client nella comunicazione TCP/IP.

Inoltre, questa classe si occupa di fornire un'interfaccia agli script C# che si occupano di simulare i sensori e gli attuatori dei robot in Unity. Per poter comunicare sfruttando ROS 2, questi script devono ottenere l'istanza della classe ROSConnection, che implementa il design pattern *singleton*. I metodi di questa classe, che gli script chiamano una volta ottenutane l'istanza, nascondono la comunicazione TCP/IP tra Unity e ROS 2 e forniscono una struttura del tutto simile a quella dei metodi implementati nelle librerie fornite da ROS 2 per iscriversi come publisher o subscriber ad un

topic e per pubblicare e ricevere messaggi. È infatti ROSConnection che si occupa di serializzare i messaggi ROS 2 ricevuti dagli script (per creare publisher, subscriber o publicare su un topic) e di inviarli al server. Analogamente, de-serializza i messaggi ricevuti dal server per passarli poi agli script interessati.

3.3 Esempio e architettura

Per chiarire il meccanismo e l'architettura ideati da Unity Technologies per integrare ROS 2 nel game engine si procede analizzando un esempio pratico, rappresentato graficamente nella Figura 3.1. I blocchi *ROS TCP Connector* e *ROS TCP Endpoint* rappresentano i programmi contenuti nelle repository descritte in sezione 3.2 e sezione 3.1, rispettivamente.

L'obiettivo è che le immagini JPEG provenienti da un *GameObject* interno alla simulazione Unity vengano pubblicate su un topic ROS 2.

Internamente alla scena Unity viene eseguito lo script MyCameraPub.cs, che esegue i metodi Start() e Update(), ereditati dalla classe base di Unity MonoBehaviour, si veda a riguardo la sezione 2.1. Nel metodo Start(), lo script MyCameraPub.cs ottiene l'istanza di ROSConnection e ne chiama il metodo relativo alla creazione di un publisher su un determinato topic, ovvero RegisterPublisher(). Nel metodo Update() chiama il metodo di ROSConnection relativo alla pubblicazione dei messaggi, ovvero Publish(). I messaggi sono di tipo CompressedImage e contengono l'immagine ottenuta chiamando il metodo CamCapture() di MyCamera.cs.

Il client serializza i messaggi e li invia al server, che si occupa della creazione di un nuovo nodo ROS 2 contenente un publisher (nel caso della chiamata di RegisterPublisher()) e alla pubblicazione dei messaggi sul topic (nel caso delle chiamate di Publish()). Per una descrizione più accurata del meccanismo necessario ad implementare una camera in Unity, si veda il Capitolo 6.

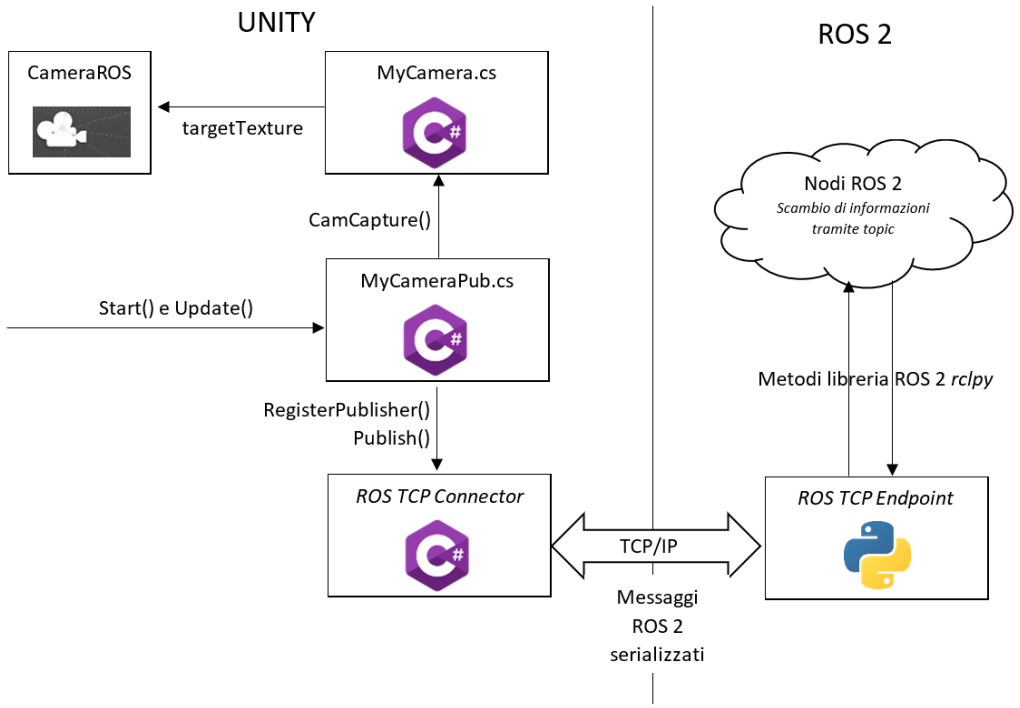

Figura 3.1: Integrazione ROS 2 - Unity con un esempio

Capitolo 4

Introduzione alla simulazione in Unity

In questo capitolo vengono descritti gli elementi fondamentali per creare e configurare una simulazione robotica in Unity sfruttando la comunicazione con ROS 2. Per iniziare una nuova simulazione bisogna accedere all'Unity Hub e creare un nuovo progetto (di tipo 3D Core) o aprirne uno preesistente (per maggiori dettagli si veda l'Appendice A). Nel primo caso si aprirà l'Editor con una scena praticamente vuota. Gli elementi fondamentali di una simulazione sono il robot e l'ambiente in cui si trova. Le sezioni 4.2 e 4.3 si concentrano su questi due aspetti. La sezione 4.1 invece descrive i passi da seguire per configurare la comunicazione tra Unity e ROS 2.

4.1 Setup della comunicazione ROS 2 - Unity

Come detto nel Capitolo 3, per fare in modo che la simulazione comunichi con componenti esterni attraverso ROS 2, Unity mette a disposizione due repository GitHub: *ROS TCP Connector* (disponibile al link: `https://github.com/Unity-Technologies/ROS-TCP-Connector`) e *ROS TCP Endpoint* (disponibile al link: `https://github.com/Unity-Technologies/ROS-TCP-Endpoint`).

Per poter usufruire di *ROS TCP Connector*, nell'Editor bisogna aprire Window→Package Manager, da qui bisogna premere sul + in alto a sinistra, premere Add package from git URL... ed inserire il seguente url: `https://github.com/Unity-Technologies/ROS-TCP-Connector.git?path=/com.unity.robotics.ros-tcp-connector`. A questo punto per configurare la connessione: Robotics→ROS Settings, indicare come Protocol ROS2 e come ROS IP Adress l'indirizzo IP della propria macchina, per il resto si possono mantenere le impostazione di default. A questo punto è disponibile in Assets→Resources il prefab ROSConectionPrefab. Un prefab in Unity è un *GameObject*, completo di eventuali componenti e altri oggetti figli, riutilizzabile a piacimento all'interno del progetto. Modificare un prefab porta alla modifica di tutte le sue istanze. Il prefab ROSConectionPrefab (compresi anche i suoi componenti, in particolare lo script ROSConnection.cs) rappresenta il lato client della comunicazione TCP/IP descritta nel Capitolo 3.

Per quanto riguarda *ROS TCP Endpoint*, è necessario creare uno workspace ROS 2 (per maggiori informazioni si vedano i tutorial nella documentazione di ROS 2 [2]). Nella cartella src di tale workspace eseguire il git clone del branch *main-ros2* della repository. Dopodichè si può procedere eseguendo il comando `colcon build` dal terminale. Per eseguire il codice che prende parte alla cominicazione TCP/IP come server bisogna eseguire dal terminale, dopo i comandi per il source dei file di setup di ROS 2 e dello workspace appena creato, il comando:

```
$ ros2 launch ros_tcp_endpoint endpoint.py
```

A questo punto, avviando la simulazione con il tasto Play dell'Editor, l'ambiente ROS 2 e la simulazione in Unity possono comunicare.

4.2 Modello del robot e URDF Importer

In Unity i robot sono modellati come *GameObject* con i propri *component* ed eventuali altri oggetti figli. Di seguito viene presentata una veloce descrizione dei principali *component* utili per modellare le caratteristiche di un robot:

- Rigidbody: fa in modo che i movimenti dell'oggetto a cui è assegnato siano influenzati dal motore fisico di Unity, per esempio rendendolo soggetto alla forza di gravità. Si possono settare alcuni parametri attraverso l'Editor, come ad esempio la massa dell'oggetto associato.

- Collider: definiscono la forma dell'oggetto per quanto riguarda le collisioni con altri oggetti (i quali a loro volta devono avere un componente Collider). Esistono diversi tipi di Collider, ciò che li differenzia è la loro forma, ad esempio i BoxCollider hanno la forma di un parallelepipedo retto, mentre gli SphereCollider di una sfera. I Collider non influenzano in alcun modo la forma visibile dell'oggetto nella simulazione.

- Mesh Collider: sono un particolare tipo di Collider la cui forma è definita in un file che descrive un Mesh. Un Mesh è un reticolo che definisce un oggetto nello spazio e permette quindi di definire forme complesse. Il problema è che i Mesh Collider non possono interagire tra loro (non possono collidere). È possibile indicare un Mesh Collider come "convex" (attraverso l'Editor) in modo che si comporti come un normale Collider, questa opzione però semplifica la forma del Collider, rendendola meno precisa.

- Mesh Filter e Mesh Renderer: Questi due componenti devono essere usati in coppia sullo stesso oggetto. Il primo è semplicemente un riferimento ad un Mesh, il secondo ha il compito di "renderizzarlo", ovvero definire la forma visibile dell'oggetto nella simulazione. Il Mesh Renderer ha diversi parametri utili a modificare l'aspetto dell'oggetto, si può specificare ad esempio l'aspetto della superficie dell'oggetto indicando nell'Editor un Material (descritto in un apposito file che deve essere tra gli Asset del progetto). Ad esempio, un Material può simulare l'aspetto di un materiale metallico.

- Articulation Body: permette di creare articolazioni tra gli oggetti che compongono una gerarchia. Sono stati realizzati appositamente da Unity per modellare articolazioni fisiche realistiche per applicazioni industriali.

- Script: permettono di modellare il comportamento del robot attraverso programmi scritti in C# (o UnityScript). Nel codice degli Scipt si possono leggere e modificare le proprietà degli altri *component* della simulazione. I valori delle variabili dichiarate come pubbliche all'interno del codice di uno Script possono essere modificati manualmente attraverso l'Editor, anche quando la simulazione è in esecuzione.

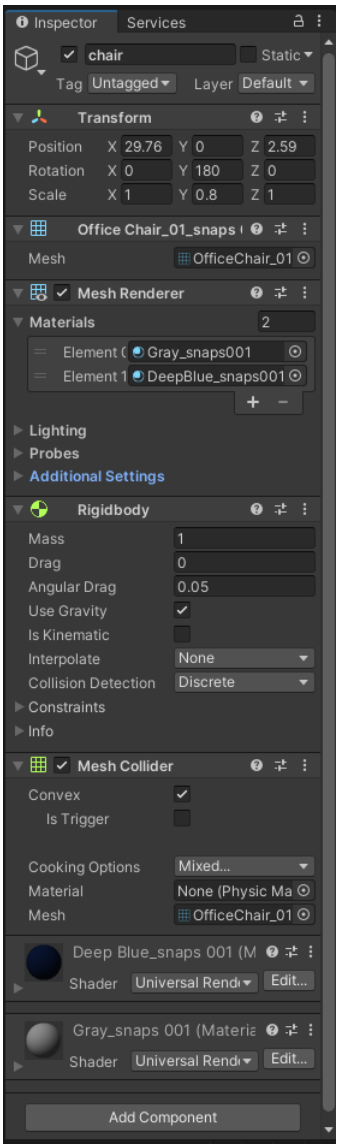

Figura 4.1: Alcuni *component* di un *GameObject* nell'*Inspector* dell'Editor

Per aggiungere un componente ad un *GameObject* bisogna selezionarlo nella *Hierarchy* nell'Editor, allora nell'*Inspector* selezionare Add Component e scegliere il *component* che si vuole aggiungere all'oggetto. I *component* attivi su un oggetto sono visibili attraverso l'*Inspector*, così come i loro parametri.

È possibile creare un robot in Unity "manualmente", ovvero creando una gerarchia di *GameObject*, ciascuno con i propri *component*, attraverso l'Editor. Tuttavia Unity mette a disposizione un tool, ovvero *URDF Importer*, che permette di importare in una simulazione il modello di un robot generato a partire da un file URDF. Lo Unified Robot Description Format (URDF) è una specifica XML utilizzata per la descrizzione di robot in una struttura ad albero. URDF permette di specificare:

- la descrizione cinematica e dinamica del robot;
- la rappresentazione visuale del robot;
- il modello delle collisioni del robot.

Come detto, *URDF Importer*, il cui codice è disponibile al link `https://github.com/Unity-Technologies/URDF-Importer`, permette di importare un robot nella simulazione, andando a definire automaticamente l'aspetto visuale del robot (attraverso dei Mesh) ed i suoi attributi dinamici e cinematici. I robot importati con questo tool sono costituiti, prevalentemente, da oggetti contenenti componenti di tipo Articulation Body.

Per poter utilizzare *URDF Importer* nel proprio progetto Unity è necessario importarlo attraverso il Package Manager (come descritto per *ROS TCP Connector*). L'url da inserire è: `https://github.com/Unity-Technologies/URDF-Importer.git?path=/com.unity.robotics.urdf-importer#v0.5.2`. Per poter importare un robot a partire da un file URDF, è necessario che questo si trovi tra gli Asset del progetto. Per importare un file tra gli Asset si può semplicemente copiare tramite il file system del sistema operativo il file (o la cartella) desiderato nella cartella Assets. Una volta individuato, all'interno dell'Editor, il file URDF desiderato, lo si deve selezionare, premere il tasto destro del mouse e selezionare Import Robot from

Selected URD file. A questo punto si apre una finestra in cui si può selezionare l'orientamento dei file Mesh necessari e scegliere l'algoritmo che verrà utilizzato per calcolare i Mesh Collider da applicare al robot. Per quest'ultima opzione si può scegliere l'algoritmo di default usato da Unity oppure VAHD, che permette di ottenere Collider più fedeli per i Mesh Collider con l'opzione "convex" abilitata. Una volta scelte le opzioni si può importare il robot premendo import URDF. Per poter funzionare, *URDF Importer* ha bisogno dei file Mesh indicati nel file URDF stesso, che devono trovarsi tra gli Asset in una cartella specifica come definito nel file URDF. Solitamente questi file sono forniti insieme al file URDF che descrive il robot.

Spesso, una volta importato un robot attraverso *URDF Importer*, è necessario intervenire manualmente attraverso l'Editor per sistemare i diversi Collider e Articulation Body generati automaticamente, che potrebbero interferire tra di loro. Inoltre l'oggetto che rappresenta il robot sarà trattato come un prefab, non sarà quindi possibile modificarlo in libertà senza influenzare le altre istanze di tale prefab. Per ovviare a questo problema si puo procedere come segue: nella *Hierarchy* selezionare la radice dell'oggetto (che sarà colorata in blu per evidenziare che è un'istanza di un prefab), premere il tasto destro del mouse, selezionare Prefab e infine Unpack Completely. A questo punto sarà possibile modificare il robot in totale libertà aggiungendo nuove parti o componenti e rimuovendone altre.

4.3 Ambiente

Unity permette di modificare l'ambiente con cui il robot interagisce con molta libertà. A tale scopo si possono aggiungere alla simulazione dei *GameObject* che rappresentano oggetti di qualsivoglia natura, come muri, piani, sedie o alberi. Questi oggetti possono essere molto semplici e stilizzati oppure molto realistici. nel primo caso basta aggiungere alla simulazione oggetti base forniti da Unity, come cilindri e cubi, a cui si possono assegnare gli stessi *component* descritti per il robot per modellarne le caratteristiche fisiche. Questi oggetti possono essere aggregati in oggetti più complessi.

Se invece si vogliono utilizzare oggetti più realistici, è possibile realizzare dei modelli tridimensionali con software come Blender o Autodesk Maya, che possono essere importati nella simulazione. Un'alternativa molto più semplice è importare modelli già pronti per essere utilizzati, disponibili sul sito di Unity Asset Store, al link `https://assetstore.unity.com/`. Questo sito mette a disposizione una moltitudine di pacchetti scaricabili ed utilizzabili nei progetti Unity, che forniscono diversi tipi di risorse, come tool, scripts e oggetti. Per questo lavoro è stato, per esempio, utilizzato un pacchetto contenente oggetti (sotto forma di prefab) che modellano forniture tipiche di un ufficio. Anche se la maggior parte dei pacchetti disponibili sono a pagamento, esiste un nutrito gruppo di elementi gratuiti.

Per scaricare ed utilizzare gli elementi forniti da un pacchetto di Unity Asset Store bisogna entrare nel sito ed accedere con lo stesso account Unity con cui si è acceduti nell'Unity Hub. Una volta individuato il pacchetto desiderato premere Add to My Assets (se necessario prima bisogna procedere col pagamento). A questo punto entrare nell'Editor di Unity ed aprire il Package Manager, selezionare quindi in alto a sinistra Package: My Assets. A questo punto si può visualizzare la lista dei pacchetti associati al proprio account, selezionare quello interessato, premere Download e poi Import. Ora in Assets sarà presente una cartella contenente tutti i file associati a tale pacchetto.

Se il materiale scaricato non è particolarmente recente, gli eventuali Material contenuti in esso potrebbero apparire di colore viola. in questo caso è necessario aggiornarli: selezionare i Material del package tra gli Asset, poi selezionare Edit→Render Pipeline→Universal Render Pipeline→Upgrade Selected Materials to UniversalRP Materials e infine confermare.

Capitolo 5

Simulazione del robot mobile TurtleBot3

Se si vuole utilizzare Unity come ambiente di simulazione di robot mobili su due ruote, Unity Technologies mette a disposizione un progetto, uno workspace ROS 2 e dei tutorial che forniscono un buon punto di partenza. Quanto elencato è contenuto nella repository GitHub `https://github.com/Unity-Technologies/Robotics-Nav2-SLAM-Example`. Il materiale permette di simulare un robot mobile, il TurtleBot3 Waffle Pi, e fare localizzazione e mappatura simultanea (Simultaneous Localization And Mapping, SLAM), ovvero costruire e tenere aggiornata una mappa di un ambiente sconosciuto, tenendo contemporaneamente traccia della posizione del robot al suo interno.

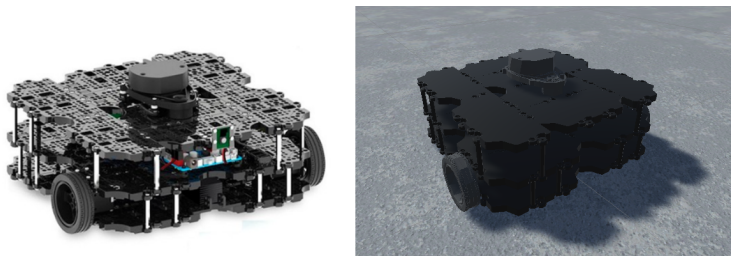

Figura 5.1: TurtlebBot3 Waffle Pi reale e nella simulazione

Per sviluppare il progetto Unity descritto nell'Appendice B (ovvero unity_project) è stato modificato appunto questo progetto fornito da Unity Tech-

nologies. Si ritiene utile fare una overview di questi materiali e una valutazione complessiva dell'ambiente messo a disposizione da Unity Technologies per la simulazione di robot mobili. Nei capitoli successivi si tratterà come espandere questo ambiente aggiungendo nuovi sensori (Capitolo 6) e l'odometria (Capitolo 8).

Una volta clonata la repository in locale si può aprire il progetto Unity, chiamato *Nav2SLAMExampleProject*, da Unity Hub. Il package *ROS TCP Connector* è già importato all'interno del progetto. Gli elementi fondamentali sono contenuti in Assets, nelle cartelle Prefabs, Resources, Scenes e Scripts. Nella cartella Prefabs è contenuto il prefab TurtleBot3ManualConfig, ovvero una particolare configurazione creata da Unity Technologies modificando l'oggetto generato dal tool *URDF Importer* a partire dal file URDF del robot. Questa modifica è risultata necessaria anche a causa di problemi con i Collider, che si sovrapponevano tra loro. Nella cartella Resources si trova il prefab dell'oggetto ROSConnectionPrefab, descritto nella sezione 4.1. In Scenes, la scena di maggiore interesse è SimpleWarehouseScene, che simula l'interno di un magazzino. Infine, nella cartella Scripts è contenuta la logica, in C#, necessaria a simulare i sensori e gli attuatori del robot:

- AGVController.cs: richiede di indicare, attraverso l'*Inspector* dell'Editor, due *GameObject*, ovvero le due ruote del robot, ed altre caratteristiche del robot, come il raggio delle ruote e la distanza tra esse. Crea un nodo subscriber sul topic /cmd_vel e calcola e applica la velocità di rotazione delle due ruote per far procedere il robot (sfruttando gli Articulation Body) in funzione dei messaggi di velocità lineare e angolare ricevuti sul topic.

- LaserScanSensor.cs: simula un sensore laser LIDAR. Ottiene le informazioni dagli oggetti circostanti solo se questi hanno un *component* di tipo Collider. Queste informazioni vengono pubblicate sul topic /scan. È possibile specificare diversi parametri, sempre utilizzando l'*Inspector*, relativi alle scansioni del sensore, come il numero di scansioni al secondo.

- Clock.cs e ROSClockPublisher.cs: necessari a mantenere sincronizzati temporalmente ROS 2 e la simulazione in Unity.
- ROSTransformTreePublisher.cs: insieme a TimeStamp.cs, TransformExtensions.cs, e TransformTreeNode.cs permette di creare l'albero delle Transform del robot e di pubblicarlo sul topic /tf.
- FreeCam.cs: non è necessario ai fini della simulazione robotica, permette semplicemente di muoversi all'interno della simulazione con una camera.

La comunicazione tramite topic tra questi script e due tool esterni, Nav2 e slam_toolbox, permette di implementare le funzioni di navigazione e mappatura. I due tool possono essere lanciati dai launch file contenuti nel package unity_slam_example dello workspace ROS 2 colcon_ws (contenuto nella cartella ros2_docker della repository). Questo workspace contiene anche il package *ROS TCP Endpoint*. Questo materiale ha lo scopo di sostituire il simulatore di robotica 3D open source Gazebo nel tutorial al link `https://navigation.ros.org/tutorials/docs/navigation2_with_slam.html`.

Questi aspetti, che possono essere esplorati seguendo il tutorial messo a disposizione nella repository, non sono di interesse per questo lavoro, che si limita all'interazione diretta con gli script presenti nel progetto Unity.

Unity Technologies mette anche a disposizione un package (Robotics Visualization, già importato in questo progetto) per la visualizzazione dei messaggi all'interno della scena. La visualizzazione può essere sia grafica, simile quindi ad rviz2, che testuale, attraverso delle finestre. Questo package fornisce un prefab contenente gli script necessari all'implementazione di queste funzioni, che deve essere quindi inserito come oggetto nella scena. In Figura 5.2 si possono vedere due finestre che mostrano i messaggi pubblicati sui topic /tf e /cmd_vel e la visualizzazione grafica delle informazioni pubblicate sul topic /map. Questo package sembra mostrare diversi problemi con alcuni topic. Ad esempio la visualizzazione grafica del topic /scan, che dovrebbe essere un point cloud, non funziona. I tutorial nella repository, accennati precedentemente, coprono anche l'utilizzo di questo package e indicano anche come personalizzare la visualizzazione. Per questo lavoro non è

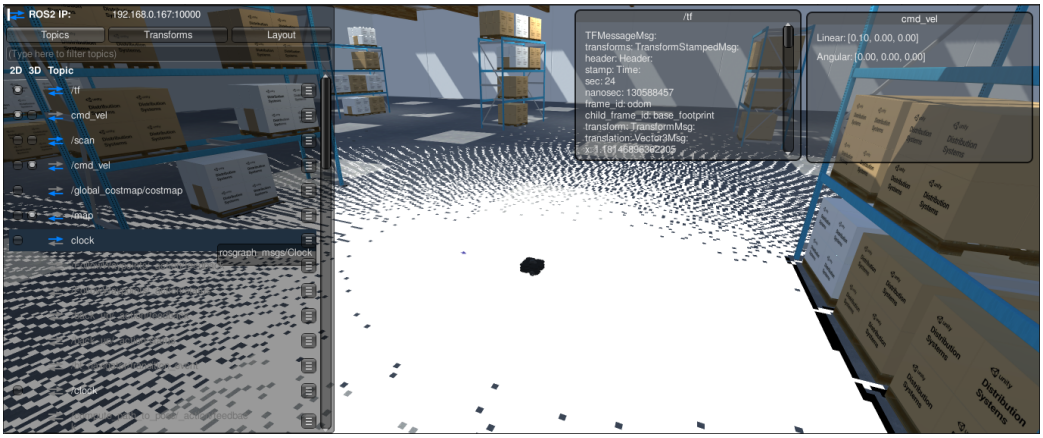

Figura 5.2: Esempio di utilizzo del package Robotics Visualization

stato utilizzato Robotics Visualization ma rviz2 (anche a causa del problema riscontrato con /scan).

Risultano di interesse per questo lavoro solamente il progetto Unity e in particolar modo gli script sopra elencati ed il modello del robot TurtleBot3. L'obiettivo infatti è poter pubblicare direttamente comandi riguardanti la velocità lineare ed angolare del robot per farlo muovere e ricevere i dati del sensore LIDAR e, in fasi successive del lavoro, di Tranform.

Sono stati scritti a tal proposito due programmi C++ che, attraverso la libreria ufficiale rclcpp, implementano altrettanti nodi ROS 2:

- cmd_vel_publisher.cpp: registra un publisher sul topic /cmd_vel e invia un messaggio di tipo Twist ogni 0.5 secondi. Il tipo Twist fa parte del package geometry_msgs, messo a disposizione da ROS 2. Contiene due vettori (linear e angular), ciascuno composto da tre float64 (x, y e z), che specificano la velocità lineare e angolare del robot. I messaggi pubblicati da questo publisher vengono ricevuti dal nodo subscriber implementato in AGVController.cs. Essendo il TurtleBot3 un robot con due ruote fisse, questo script utilizza solo i valori di linear x e angular z per calcolare la velocità di rotazione delle ruote. Questo publisher publica messaggi con linear x = 0.1 e angular z = 0.0. Quindi si comanda al robot di procedere dritto ad una velocità di 0.1m/s.

- scan_subscriber.cpp: registra un subscriber sul topic /scan e attende messaggi di tipo LaserScan. Questo tipo di messaggio fa parte del package sensor_msgs, messo anch'esso a disposizione da ROS 2. Il tipo LaserScan contiene un array (ranges[]) di float32. L'array rappresenta l'insieme delle misurazioni fatte in una scansione. Inoltre, LaserScan contiene altre informazioni relative alle scansioni, come la massima e minima distanza misurabile (range_max e range_min) e l'angolo tra una misurazione e l'altra (angle_increment). Per ogni messaggio ricevuto lo script stampa su console il primo e l'ultimo valore di ranges[]. Sul questo topic pubblica messaggi il nodo publisher implementato in LaserScanSensor.cs.

Questi due programmi si possono trovare nella cartella ros_unity_sim, nello workspace ROS 2 ros_ws, package unity_pub_sub. A tal proposito si faccia riferimento all'Appendice B.

Per visualizzare i messaggi di tipo LaserScan è possibile utilizzare il tool rviz2, che li rappresenta con un point cloud. Per avviare rviz2 bisogna, dopo aver fatto il source del file di setup di ROS 2, eseguire il seguente comando nel terminale:

```
$ ros2 run rviz2 rviz2
```

Una volta aperto rviz2 sarà sufficiente aggiungere il topic /scan per visualizzare graficamente i messaggi publicati da LaserScanSensor.cs.

Utilizzando il programma cmd_vel_publisher.cpp il robot dovrebbe proseguire dritto senza curvare, dal momento che non si applica nessuna velocità angolare. Il comportamento osservato nella simulazione, però, è una curvatura della traiettoria del robot il cui angolo diminuisce all'aumentare della velocità lineare. Questo problema è trattato nel Capitolo 7.

Capitolo 6

Aggiunta di sensori al robot

I sensori sono una parte fondamentale di ogni robot ed è quindi importante poterli simulare. In Unity i sensori vengono modellati attraverso gli Script C#, assegnabili agli oggetti. Ad esempio, nella simulazione discussa nel Capitolo 5 il robot è dotato di un sensore LIDAR. Questo è realizzato come un albero di *GameObject*, la cui radice è un oggetto (chiamato base_scan) che contiene lo script LaserScanSensor.cs, il quale modella il comportamento del sensore. Quindi per aggiungere nuovi sensori è necessario disporre degli Script che ne implementino il comportamento.

In rete non c'è molta disponibilità di Script per simulare sensori in Unity. Alcuni sensori possono essere trovati alle seguenti repository GitHub: `https://github.com/Field-Robotics-Japan/UnitySensors` e `https://github.com/Field-Robotics-Japan/UnitySensorsROS`, che congiuntamente permettono di simulare diversi sensori e pubblicare i dati ottenuti su topic di ROS 2. Una prova di questo materiale ha però evidenziato diversi problemi (per esempio gli Script non vengono compilati correttamente per un errore del tipo di una variabile) che ne precludono l'utilizzo.

Per aggiungere nuovi sensori ad una simulazione Unity potrebbe quindi risultare necessario scrivere il codice da zero. Per questo lavoro si è deciso di scrivere degli Script, prendendo come spunto quelli già forniti da Unity e quelli delle repository sopraccitate, che modellino una camera da applicare al TurtleBot3. L'obbiettivo di questo sensore è di pubblicare messaggi di

tipo CompressedImageMsg, contenenti le immagini che cattura, sul topic /image/compressed.

Gli Script non sono però sufficienti a realizzare questo sensore. Per prima cosa è necessario creare un oggetto vuoto (chiamato CameraROS), aggiungerlo nella gerarchia del robot come figlio dell'oggetto base_link e spostarlo in modo che si "appoggi" su una superficie del robot. Dopodiché va aggiunto a CameraROS il *component* Camera che permette di raccogliere immagini dalla simulazione. Per rendere il sensore più realistico si può impostare il parametro Clipping Planes Near del componente appena aggiunto al valore minimo 0.01, in modo che la camera non "tagli" gli oggetti vicini ad essa. Infine si deve creare tra gli Assets un Render Texture (nella finestra Project dell'Editor, premere tasto destro in Assets→Create→Render Texture) e rinominarlo renderCamera. Una volta selezionato renderCamera nell'*Inspector* si possono modificare i suoi parametri. In particolare bisogna impostare il valore di Dimension a 2D e scegliere la risoluzione dell'immagine che verrà trasmessa modificando Size. Per applicare renderCamera a CameraROS bisogna selezionare CameraROS nella gerarchia degli oggetti e trascinare renderCamera sull'attributo Output Texture del componente Camera nell'*Inspector*.

Gli Script scritti per questo sensore sono due: MyCamera.cs e MyCameraPub.cs, entrambi disponibili in Scripts/myRGBCamera tra gli Assets del progetto unity_project (si veda l'Appendice B). Ambedue gli Script contengono una classe che eredita dalla classe MonoBehaviour.

MyCamera, la classe implementata in MyCamera.cs, contiene il solo metodo pubblico CamCapture(), questo accede al *component* Camera (e di conseguenza al Render Texture associato), ottiene da esso un'immagine e salva in codifica JPEG un array di byte in una variabile pubblica (chiamata data).

La classe implementata in MyCameraPub.cs contiene i metodi Start() e Update(), ereditati da MonoBehaviour. In Start() si ottiene un'istanza della classe MyCamera e si configura la comunicazione ROS 2 ottenendo l'istanza di ROSConnection e registrando un publisher sul topic /image/subscriber. Nel metodo Update() si chiama il metodo CamCapture() di MyCamera, si configura il messaggio (che contiene anche i dati della variabile pubblica data di MyCamera) e infine lo si pubblica sul topic. Le immagini non vengono

pubblicate ogni volta che viene eseguito Update() (ovvero ad ogni frame della simulazione), ma ad una frequenza minore. Questa può essere impostata modificando il parametro Scan Rate dello Script attraverso l'*Inspector*.

Per completare la configurazione del sensore è sufficiente assegnare all'oggetto CameraROS, descritto precedentemente, lo Script MyCameraPub.cs; in automatico verrà aggiunto anche MyCamera.cs.

Analizzando il codice di MyCameraPub.cs si può notare la semplicità con cui è possibile creare un publisher ROS 2 in una simulazione Unity utilizzando la classe ROSConnection. È sufficiente infatti creare una variabile di tipo ROSConnection:

```
private ROSConnection _ros;
```

successivamente creare un publisher nel metodo Start():

```
this._ros = ROSConnection.GetOrCreateInstance();
this._ros.RegisterPublisher<CompressedImageMsg>(
    this._topicName);
```

infine, ogni qualvolta si vuole pubblicare un messaggio, chiamare il metodo Publish():

```
this._ros.Publish(this._topicName, this._message);
```

Per testare questo sensore è stato scritto image_subscriber.cpp, un programma che implementa un nodo ROS 2 con un subscriber sul topic /image/compressed. Ogni qualvolta il subscriber riceve un messaggio su questo topic, il programma, sfruttando la libreria open source OpenCV, salva un file JPEG in locale contenente l'immagine racchiusa nel messaggio. Le immagini vengono salvate nella cartella in cui si lancia il comando per eseguire il programma. La Figura 6.1 mostra un esempio di immagine catturata nella simulazione, pubblicata tramite ROS 2 e salvata con OpenCV.

Le immagini pubblicate da MyCameraPub.cs possono essere visualizzate anche con il plugin rqt_image_view, mentre rviz2 riconosce il topic ma non riceve le immagini.

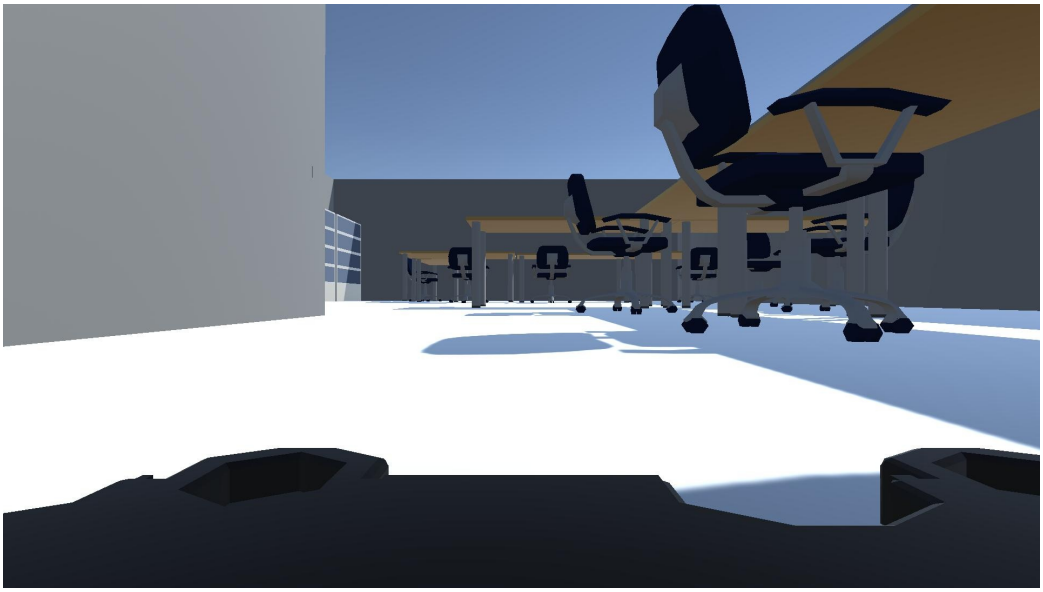

Figura 6.1: Immagine pubblicata da MyImagePub.cs e salvata da image_subscriber.cpp

Capitolo 7

Movimento di TurtleBot3

Come già accennato, il TurtleBot3 della simulazione discussa nel Capitolo 5 presenta un'anomalia nel movimento. Infatti, quando riceve un comando di procedere dritto sul topic /cmd_vel, il robot si muove seguendo una traiettoria curva. La curvatura è tanto maggiore quanto più è alta la velocità lineare (linear x) specificata nel messaggio. Questo problema non consente di modellare fedelmente il comportamento del robot.

Lo Script che riceve i messaggi pubblicati sul topic /cmd_vel e comanda il movimento del robot è AGVController.cs. Analizzandone il codice si può notare che questo programma, quando la variabile ControlMode è impostata al valore ROS, utilizza i componenti linear x e angular z dei messaggi pubblicati sul topic per calcolare la forza da applicare alle ruote del robot. Per calcolare questa forza utilizza come parametri il raggio delle ruote e la distanza tra di esse. La forza viene applicata al parametro xDrive dei *component* Articulation Body associati alle ruote, che iniziano quindi a ruotare. Il movimento delle ruote influisce sugli altri oggetti del robot con un Articulation Body attivo. Tutto questo permette al robot di muoversi. Si fa notare che un messaggio pubblicato sul topic aziona le ruote per un massimo di 0.5 secondi, dopodiché il robot viene arrestato in attesa di un nuovo messaggio. Questo tempo può essere personalizzato modificando il parametro ROS Timeout di AGVController.cs.

Per verificare il funzionamento dello Script, si può avviare la simulazione

e pubblicare su /cmd_vel ogni 0.5 secondi un messaggio con linear x pari, per esempio, a 0.5 e tutti gli altri valori a 0. Il robot inizierà quindi a muoversi, ma non seguendo una traiettoria retta. Selezionando nella *Hierarchy* dell'Editor l'oggetto della ruota destra che contiene l'Articulation Body, si può notare nell'*Inspector* il valore di xDrive applicato alla ruota. Seguendo la stessa procedura per la ruota sinistra si verifica che i valori applicati al parametro xDrive delle ruote sono identici. Ci si aspetterebbe quindi che il robot prosegua in linea retta, ma così non è.

Si pensa che questa anomalia sia dovuta al modello del robot, in particolar modo all'interazione tra i *component* di tipo Collider ed Articulation body. Per ovviare a questo problema si è quindi provato a modificare la struttura e la forma del robot, prestando particolare attenzione ai Collider. Le varie configurazioni ottenute mostravano comportamenti diversi: in alcuni casi la curvatura era particolarmente importante, in altri era meno evidente. In nessun caso però il problema è stato completamente risolto. Inoltre, si è provato sia a creare nuovi robot molto semplici direttamente attraverso l'Editor che ad importare e modificare altri modelli di robot su due ruote con *URDF Importer*, ma il risultato non è cambiato.

È stato quindi cambiato approccio. Si è deciso di muovere il robot modificandone direttamente la posizione assoluta rispetto al centro della scena simulata. Per prima cosa è stato modificato il robot (partendo dalla configurazione iniziale fornita da Unity) rimuovendo tutti i *component* Articulation Body e gli script generati da *URDF Importer* che dipendevano da essi. Dopodiché è stato rimosso caster_center_manual_config dalla gerarchia degli oggetti e si è modificato il Box Collider principale del robot (presente nella sottogerarchia di base_link). È stato anche agggiunto a base_footprint un RigidBody per modellare alcuni aspetti del robot che prima dipendevano dagli Articulation Body, come ad esempio la massa. Infine è stato sostituito AGVController.cs con il nuovo Script GTController.cs.

Questo programma ha una struttura simile ad AGVController.cs per quanto riguarda la ricezione dei comandi pubblicati su /cmd_vel, ma differisce nel modo in cui fa muovere il robot. GTController.cs contiene un metodo, denominato FixedUpdate() ed ereditato dalla classe MonoBehaviour, che vie-

ne eseguito ogni lasso di tempo fissato. Questo tempo può essere visualizzato e modificato attraverso l'Editor: Edit→Project Settings→Time→Fixed Timestep. In questo metodo viene ottenuta la posizione assoluta attuale del robot, che viene poi utilizzata insieme ai comandi di velocità per calcolare la posizione finale da applicare al robot. Per questi calcoli è necessario disporre della distanza tra le due ruote (che viene specificata attraverso il parametro track Width, modificabile attraverso l'Editor) e del Fixed Timestep descritto prima, a cui si può accedere all'interno del programma come segue:

```
float step = Time.deltaTime;
```

Per fare in modo che GTController.cs funzioni correttamente è necessario assegnare attraverso l'Editor l'oggetto base_footprint al parametro Gt dello Script. Questo permette al programma di accedere alla posizione del robot e di modificarla.

7.1 Implementazione di zone scivolose

Nello Script GTController.cs è stata anche implementata la logica per simulare la presenza di zone scivolose nel piano su cui si muove il robot. L'idea è che quando una ruota del robot si trova su una di queste zone, essa perda trazione ed inizi a scivolare. Questo influenza ovviamente sia la velocità angolare che lineare del robot.

Per modellare le zone scivolose è stato scelto di utilizzare dei *GameObject* di tipo Plane, che possono essere sovrapposti al piano principale nelle zone desiderate. In Figura 8.1 si possono notare due piani (di colore blu) che modellano altrettante zone scivolose. A questi piani va applicato un nuovo Script, ovvero PlaneFriction.cs. Questo programma implementa una classe che contiene la sola variabile pubblica friction. Questa variabile è modificabile attraverso l'Editor e può assumere valori compresi tra 0 e 1. Rappresenta l'attrito del piano: più basso è il suo valore e più le ruote scivolano, rallentando di conseguenza il movimento del robot.

GTController.cs è stato modificato per poter accedere alla posizione delle ruote e alle caratteristiche di un numero arbitrario di piani (a cui deve es-

sere applicato PlaneFriction.cs). Questi oggetti vanno assegnati, attraverso l'*Inspector*, al *component* relativo a GTController.cs, come in Figura 7.1.

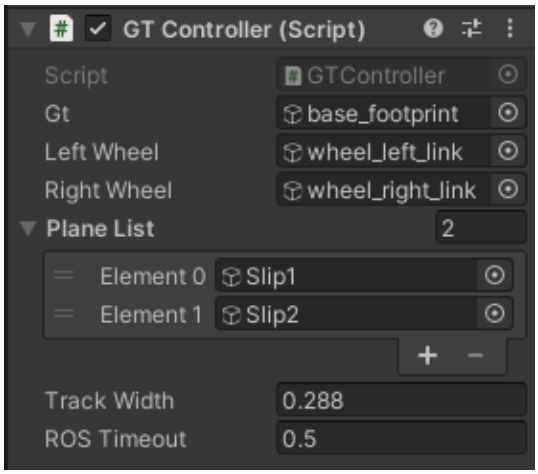

Figura 7.1: Esempio di GTController.cs nell'*Inspector*

GTController.cs presenta tre problematiche:

- le zone scivolose possono avere solo la forma di piani rettangolari non ruotati;
- se le due ruote si trovano contemporaneamente su due piani scivolosi con valori di attrito diversi, il robot si comporta come se entrambe le ruote fossero su una zona con attrito pari a quello della zona su cui si trova la ruota sinistra;
- quando una sola ruota è su un'area scivolosa l'angolo di curvatura del robot cambia continuamente.

Gli Script GTController.cs e PlaneFriction.cs si trovano in Scripts/myControllers tra gli Assets del progetto unity_project (si veda l'Appendice B).

Capitolo 8

Aggiunta dell'odometria

L'odometria permette di stimare la posizione attuale del robot rispetto a quella iniziale attraverso i dati sul movimento (rotazione) delle ruote, ottenuti attraverso dei sensori, ad esempio degli encoder ottici. La posizione reale del robot e quella ottenuta con l'odometria possono differire a causa di zone scivolose sulle quali le ruote del robot continuano a girare ma il robot stesso avanza più lentamente (come conseguenza dello slittamento). L'odometria si discosta dalla posizione effettiva del robot anche nel caso in cui il robot reale sia bloccato da un ostacolo ma le ruote continuano a girare.

L'odometria è un aspetto fondamentale nell'ambito dei robot mobili e quindi risulta di interesse poterla aggiungere all'interno dell'ambiente di simulazione di Unity. L'implementazione dell'odometria in questo progetto prevede di simulare e visualizzare all'interno della scena la posizione del robot che verrebbe calcolata a partire dagli encoder in una situazione reale. All'interno della scena saranno presenti due robot diversi, uno che simula il robot reale, l'altro che simula la posizione relativa all'odometria. Il primo robot (chiamato ground_truth) collide con gli oggetti e slitta su zone scivolose, il secondo (chiamato odom), invece, non collide con nessun oggetto e procede sempre come se si trovasse su un terreno non scivoloso. Entrambi devono partire nella stessa posizione e non devono collidere tra di loro. In Figura 8.1 il robot semitrasparente simula l'odometria ed è più avanti rispetto al robot reale perché quest'ultimo si trova su una zona scivolosa (area blu).

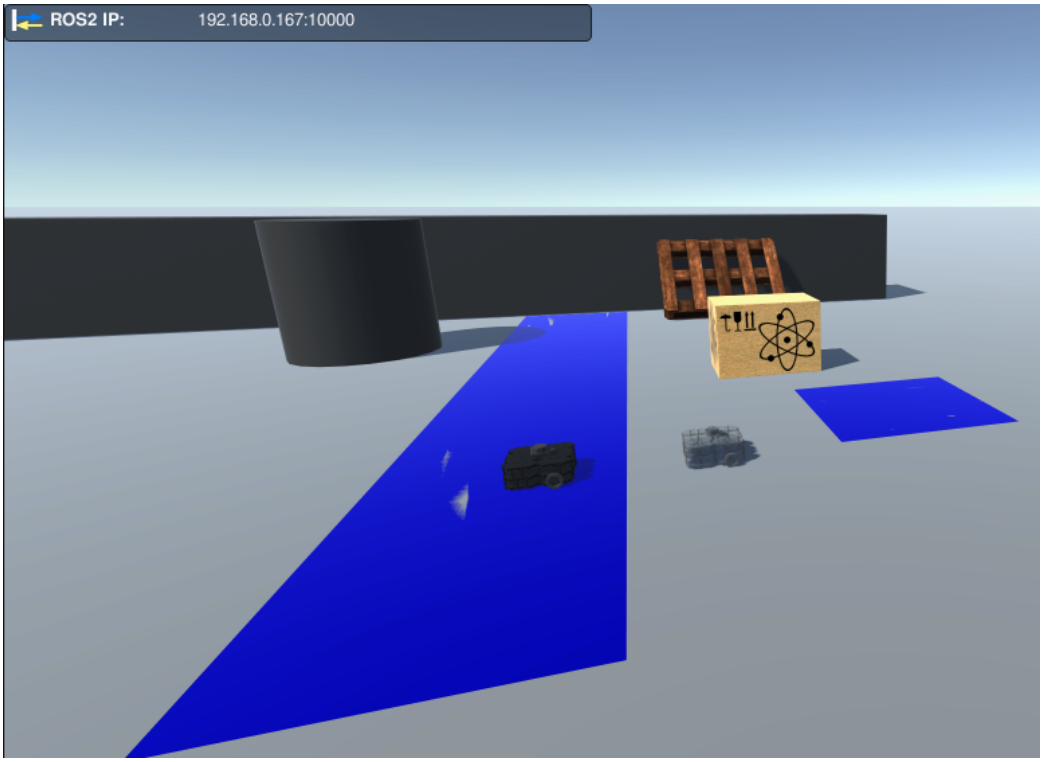

Figura 8.1: Simulazione dell'odometria in Unity

In questo capitolo si tratterà inoltre come pubblicare le informazioni riguardanti la posizione dei due robot (quello reale e l'odometria) rispetto alla posizione iniziale tramite topic ROS 2. Per far ciò si sfrutterà il topic /tf.

8.1 Implementazione dell'odometria

Per simulare l'odometria si deve ottenere all'interno della scena Unity una gerarchia come quella mostrata in Figura 8.2. Per ottenere tale gerarchia è sufficiente creare un nuovo *GameObject* (turtlebot3) e aggiungere come suoi figli gli oggetti rappresentanti i due robot, ovvero groud_truth e odom. Il primo è stato ottenuto rinominando l'oggetto turtlebot3_manual_config fornito da Unity e modificato come indicato nel Capitolo 6 e nel Capitolo 7. Invece, per ottenere odom è necessario:

- creare una copia di gorund_truth all'interno della scena e rinominarla odom;

- rinominare i figli base_footprint, base_link, base_scan, imu_link, wheel_right_link e wheel_left_link aggiungendo il prefisso virtual_ davanti al nome. Questa operazione è necessaria per pubblicare correttamente le informazioni su /tf;

- eliminare gli elementi superflui, ovvero l'oggetto CameraROS e i *component* ROSClockPublisher in ROSPublisher e LaserScanSensor in virtual_base_scan.

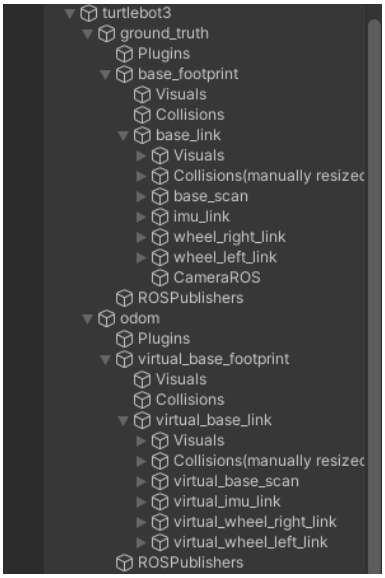

Figura 8.2: Gerarchia dell'oggetto turtlebot3

Per far sì che il robot che simula l'odometria non collida con gli oggetti presenti nella scena e con il robot ground_truth è possibile sfruttare i layer di Unity:

- per creare un nuovo layer nel progetto navigare in Edit→Project Settings →Tags and Layers, a questo punto è possibile dare un nome, ad esempio "virtuale", ad uno dei layer disponibili;

- per far sì che gli oggetti del nuovo layer non collidano con gli altri, navigare in Edit→Project Settings→Physics→Layer Collision Matrix, a questo punto si devono togliere le spunte tra "virtuale" e tutti gli altri layer eccetto "virtuale" stesso, come mostrato in Figura 8.3;

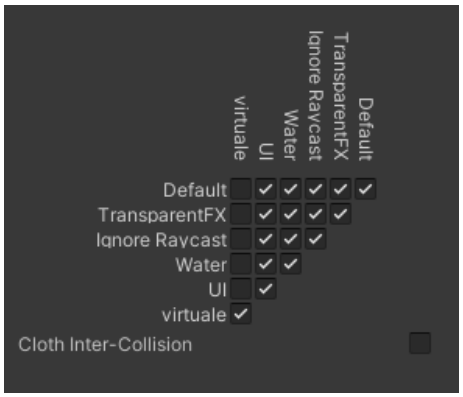

Figura 8.3: Matrice dei layer in Unity

- per modificare il layer dell'albero di oggetti odom bisogna selezionare l'oggetto odom stesso e, nell'*Inspector*, in alto a destra, cambiare il layer in "virtuale". Alla richiesta di applicare la modifica a tutti i figli rispondere affermativamente;

- per far sì che anche il robot odom abbia un piano su cui appoggiarsi bisogna aggiungere alla scena un *GameObject* di tipo Plane, a cui applicare il layer "virtuale". Questo piano deve essere sovrapposto a quello "reale". Per evitare che il nuovo piano sia visibile è sufficiente togliere il Material al *component* Mesh Renderer;

- infine, per evitare che l'oggetto CameraROS di ground_truth veda il robot odom, bisogna selezionare CameraROS e, nell'*Inspector*, deselezionare il layer "virtuale" in Rendering→Culling Mask.

Per differenziare i due robot è possibile cambiare colore ad uno dei due, oppure, dal momento che il robot odom non simula un robot reale, lo si può rendere semitrasparente:

- creare un nuovo Material: Assets→Materials→tasto destro→Create →Material;

- selezionare questo Material, nell'*Inspector* cambiare shader come segue: Shaders→Shader Graphs→ArnoldStandardSurfaceTrasparent, a questo punto si possono cambiare colore (Base Color) e livello di trasparenza (Opacity);

- a questo punto, navigare nella *Hierarchy*: odom→virtual_base_footprint →virtual_base_link→Visuals→unnamed→waffle_base, selezionare uno alla volta i 5 oggetti che ci sono all'interno e, dall'*Inspector*, cambiare il Material del *component* Mesh Renderer con quello appena creato. Applicare la stessa procedura per le due ruote e per virtual_base_scan (scendendo la gerarchia fin quando si trova un oggetto con Mesh Renderer come *component*).

Nel progetto unity_project contenuto nella cartella ros_unity_sim, nelle scene SlipScene e Labs questi passaggi sono già stati eseguiti.

A questo punto l'unico elemento che manca è la logica necessaria a muovere il robot odom. Per implementarla è sufficiente scrivere un nuovo script, chiamato OdomController.cs, che sostituisca GTController.cs (si veda il Capitolo 7) come *component* in odom. Questo nuovo script può essere ottenuto da GTController.cs eliminando tutto ciò che riguarda le zone scivolose. Una volta assegnato OdomController.cs ad odom, è necessario specificare nell'*Inspector* l'oggetto comandato (ovvero virtual_base_footprint), la distanza tra le ruote e ROS Timeout. Lo Script si trova in Scripts/myControllers tra gli Assets del progetto unity_project (si veda l'Appendice B)

8.2 Pubblicazione su topic dell'odometria

Per pubblicare le informazioni riguardanti la posizione di ground_truth e di odom rispetto al comune punto di partenza, si sfrutta il topic /tf. Su questo topic pubblica messaggi lo script ROSTransformTreePublisher.cs (si veda il Capitolo 5). Le transform pubblicate su /tf dallo script ROSTransformTree-

Publisher.cs sono organizzate in un albero e per ogni coppia padre-figlio si hanno a disposizione tre valori per le traslazioni (x, y e z) e quattro per le rotazioni (x, y, z e w). Questi valori indicano la posizione nello spazio (e la rotazione) del figlio rispetto al padre. L'esecuzione di ROSTransformTree-Publisher.cs crea un albero di tf come segue:

```
map
   odom
      --albero del root game object--
```

map e odom sono la stessa cosa, ovvero il riferimento (0 0 0), il centro della scena. A seguito delle operazioni svolte nella sezione precedente sono due gli Script ROSTransformTreePublisher.cs in esecuzione: uno per odom e uno per ground_truth. In entrambi i casi lo Script è un *component* dell'oggetto ROSPublishers. Per far sì che si pubblichi un albero come quello in Figura 8.4, è necessario selezionare l'oggetto ROSPublishers interno alla gerarchia di ground_truth e modificare, nell'*Inspector*, il *component* ROSTransformTree-Publisher.cs, eliminando odom dai Global Frame Ids e mettendo come Root Game Object l'oggetto ground_truth stesso. Analogamente bisogna modificare l'oggetto ROSPublishers interno alla gerarchia di odom, con la differenza che in questo caso il Root Game Object deve essere l'oggetto odom. Inoltre, è necessario aprire lo script ROSTransformTreePublisher.cs e, nel campo m_GlobalFrameIds, eliminare dalla lista la stringa "odom".

A questo punto, tra le informazioni pubblicate sul topic /tf, compaiono la Tranform con frame_id=ground_truth e child_frame_id=base_footprint e quella con frame_id=odom e child_frame_id=virtual_base_footprint, che contengono, rispettivamente, la posizione dei robot ground_truth e odom rispetto alla posizione iniziale.

in Figura 8.5 si può notare la visualizzazione con rviz2 dei messaggi pubblicati su /tf. É visibile il centro della scena, a cui sono collegate le posizioni iniziali dei robot (che coincidono) a cui a loro volta sono collegati tutti gli elementi dell'albero di ciascun robot.

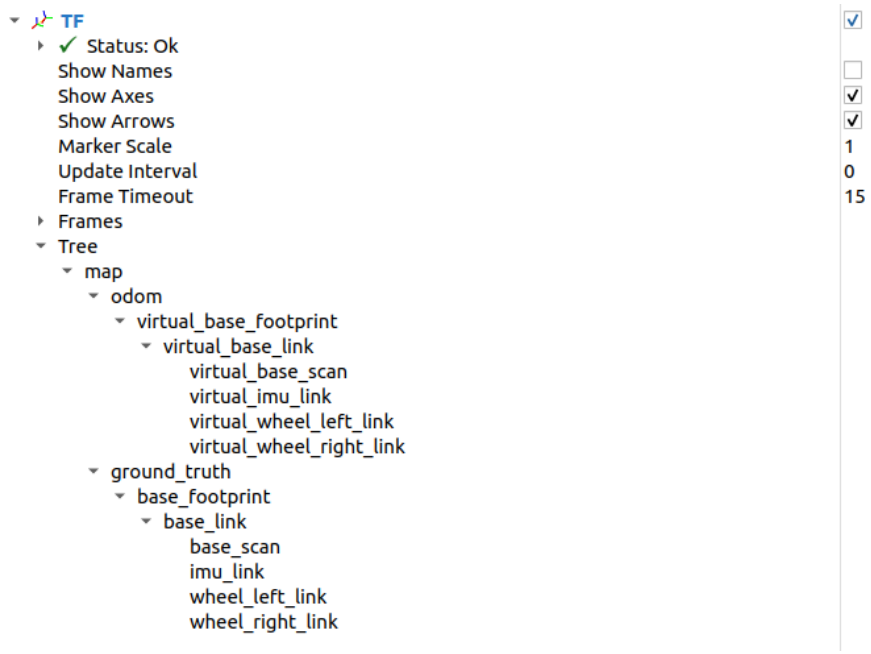

Figura 8.4: Albero di transform visualizzato con rviz2

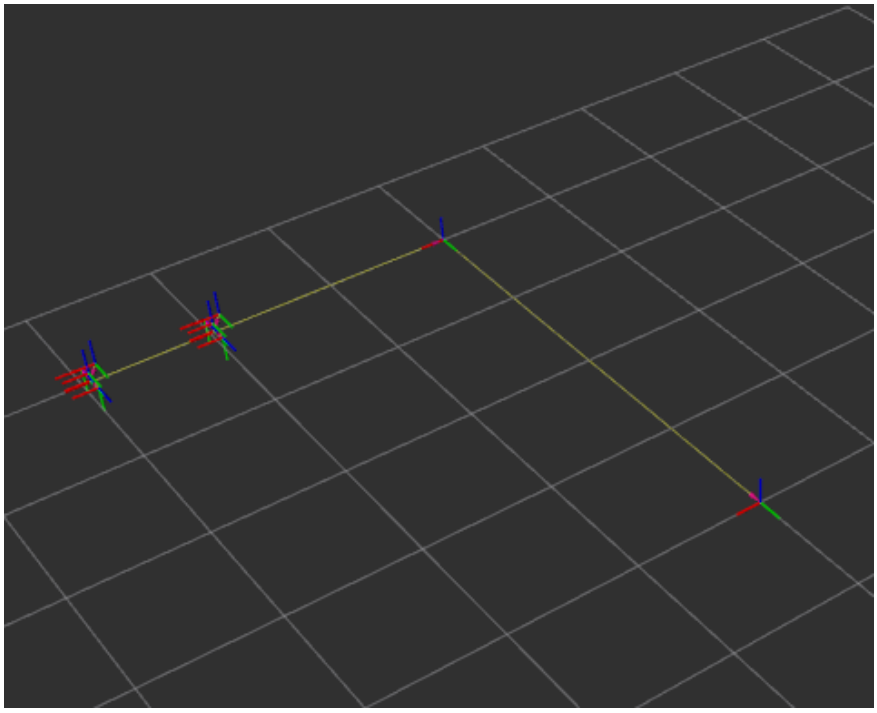

Figura 8.5: Visualizzazione di transform in rviz2

Capitolo 9

Comunicazione tra Unity e DDS

ROS 2 utilizza come middleware lo standard DDS. È quindi possibile utilizzare un'implementazione di DDS per comunicare con l'ambiente ROS 2. In questo modo, anziché scrivere programmi C++ che si appoggiano sulle librerie di ROS 2 per scambiare informazioni e comandi con il robot simulato in Unity, è possibile scrivere programmi, sempre in C++, che utilizzano le librerie di un'implementazione del DDS, come eProsima fast DDS. Dal momento che ROS 2 fornisce un'astrazione della comunicazione DDS, lo sforzo richiesto per implementare la comunicazione direttamente sfruttando eProsima è maggiore, come si può capire confrontando le Figure 2.2 e 2.3.

Per implementare un semplice publisher o subscriber è possibile seguire la documentazione di eProsima, al link `https://fast-dds.docs.eprosima.com/en/v2.2.0/fastdds/getting_started/simple_app/simple_app.html`. Per far sì che i nodi implementati in eProsima comunichino con l'ambiente ROS 2, è necessario che i nomi dei topic e i topic type nei due ambienti siano compatibili. Per illustrare come ottenere questa compatibilità si farà riferimento ad un esempio pratico, ovvero il topic /cmd_vel, utilizzato per comunicare comandi di velocità (lineare e angolare) al robot.

Per conoscere il nome del topic e il suo topic type si deve lanciare il server endpoint.py dal terminale (si veda la sezione 4.1) e avviare la simulazione.

A questo punto è possibile aprire un altro terminale e, dopo aver eseguito il comando per il source di ROS 2, eseguire il comando:

```
$ ros2 topic list -t
```

che lista tutti i topic attivi con i relativi topic type. Se lo script AGVController.cs (si veda il Capitolo 5) è abilitato all'interno della simulazione, si può individuare la seguente linea di testo:

```
/cmd_vel [geometry_msgs/msg/Twist]
```

dove /cmd_vel è appunto il nome del topic e geometry_msgs/msg/Twist il topic type. Twist è un type messo a disposizione da ROS 2 contenente due vettori da tre elementi float64 ciascuno.

Ad ogni topic ROS 2 corrisponde un topic DDS il cui nome viene ottenuto concatenando il prefisso rt con il nome del topic ROS 2. Nel programma che implementa un nodo DDS non bisogna quindi settare il nome del topic come /cmd_vel, ma come rt/cmd_vel.

In eProsima i topic type sono descritti in file IDL, utilizzati poi dal tool Fast DDS-Gen, descritto nella sezione 2.3. Il file Twist.idl relativo a questo esempio deve essere definito come segue:

```
#include "Vector3.idl"
module geometry_msgs {
  module msg {
    module dds_ {
      struct Twist_ {
        Vector3 linear;
        Vector3 angular;
      };
    };
  };
};
```

I primi due module e struct servono a replicare la struttura prevista da ROS 2: geometry_msgs/msg/Twist. Si noti che module dds_ e l'underscore dopo Twist non sono opzionali, ma servono per rendere compatibili i tipi di ROS

2 ed eProsima DDS. Il type Vector3, non essendo utilizzato direttamente da nessun nodo, può essere descritto in un file Vector3.idl più semplicemente:

```
struct Vector3 {
  double x;
  double y;
  double z;
};
```

All'interno di ros_unity_sim, si veda l'Appendice B, è stato creato uno workspace contenente due package: dds_unity e dds_tf. Il primo implementa in maniera pressoché identica i programmi scritti sfruttando le librerie di ROS 2 e descritti nel Capitolo 5 (scan_subscriber.cpp e cmd_vel_publisher.cpp) e nel Capitolo 6 (image_subscriber.cpp). L'unica differenza è che nel caso di DDS, image_subscriber.cpp salva in locale solamente le prime cinque immagini ricevute e le salva in ros_unity_sim/dds_ws/dds_unity. Nel package dds_tf, lo script tf_subscriber.cpp registra un subscriber sul topic /tf. Sullo stesso topic pubblica messaggi di tipo Tranform lo script ROSTransformTreePublisher.cs. Una volta ottenuto un messaggio, il programma stampa a console solamente le informazioni riguardanti gorund_truth e odom (si veda il Capitolo 8).

I nodi implementati da scan_subscriber.cpp e image_subscriber.cpp ricevono messaggi che contengono degli array. In eProsima, per far si che messaggi contenenti array vengano sempre ricevuti correttamente, bisogna modificare i QoS del DataReader. Questo perché la configurazione di default del DataReader consente di ricevere messaggi contenenti array con al massimo 100 elementi, poiché ne alloca precedentemente lo spazio. Per far sì che vengano ricevuti correttamente array con un numero arbitrario di elementi, è necessario che il DataReader gestisca dinamicamente la memoria allocata agli array. Per far ciò bisogna crearlo con i QoS corretti, come segue:

```
DataReaderQos readerQos;
readerQos.endpoint().history_memory_policy =
    eprosima::fastrtps::rtps::DYNAMIC_RESERVE_MEMORY_MODE;
```

```
reader_ = subscriber_->create_datareader(topic_,
    readerQos, &listener_);
```

Per implementare nodi con publisher che devono pubblicare messaggi contenenti array di lunghezza arbitraria si può adottare una soluzione del tutto analoga durante la creazione del DataWriter. Per quanto riguarda la pubblicazione di messaggi, inoltre, bisogna prestare attenzione a non eliminare il publisher immediatamente dopo aver pubblicato un messaggio. Infatti, in questo caso, il messaggio viene effettivamente pubblicato correttamente e, ad esempio, è possibile visualizzarlo con un comando di echo sul topic da terminale, ma non viene ricevuto da Unity.

Bibliografia

[1] Unity Technologies, *Unity Robotics Hub*, GitHub repository, `https://github.com/Unity-Technologies/Unity-Robotics-Hub`.

[2] Open Robotics, *ROS 2 Documentation: Foxy*, `https://docs.ros.org/en/foxy/index.html`.

[3] Open Robotics, *ROS Wiki: Documentation*, `http://wiki.ros.org/`.

[4] eProsima , *eProsima Fast DDS Documentation*, `https://fast-dds.docs.eprosima.com/en/v2.2.0/index.html`.

[5] Unity Technologies, *Robotics Nav2 SLAM Example*, GitHub repository, `https://github.com/Unity-Technologies/Robotics-Nav2-SLAM-Example`.

[6] Unity Technologies, *Unity User Manual 2020.3 (LTS)*, `https://docs.unity3d.com/2020.3/Documentation/Manual/index.html`.

Appendice A

Installazione ed uso

Per poter utilizzare il materiale contenuto nella cartella ros_unity_sim, descritta nell'Appendice B, è necessario predisporre l'ambiente di lavoro installando localmente ROS 2, eProsima e Unity. Di seguito si forniscono le informazioni relative a come installare i software necessari sul sistema operativo Ubuntu 20.04 LTS Focal Fossa.

- Per installare la distro Foxy Fitzroy di ROS 2 si devono seguire i passi indicati nella documentazione ufficiale: `https://docs.ros.org/en/foxy/Installation/Alternatives/Ubuntu-Install-Binary.html`.
- Analogamente, per eProsima Fast DDS 2.2.0 si può seguire la guida all'installazione della documentazione, presente al link `https://fast-dds.docs.eprosima.com/en/v2.2.0/installation/binaries/binaries_linux.html`.
- Alcuni programmi utilizzano la libreria OpenCV per la gestione delle immagini. Per installare questa libreria si segua la documentazione ufficiale al link `https://docs.opencv.org/4.x/d7/d9f/tutorial_linux_install.html`.
- Per quanto riguarda Unity è necessario creare un account sul sito ufficiale. Dopodiché si consiglia di installare l'Editor passando per Unity Hub, che facilita la gestione dei progetti e delle versioni dell'Editor. La guida per l'installazione di Unity Hub per Linux è disponibile al link

`https://docs.unity3d.com/hub/manual/InstallHub.html#install-hub-linux`, nella sezione Debian or Ubuntu. Una volta aperto Unity Hub, bisogna accedere con l'account creato in precedenza. A questo punto è possibile scaricare la versione LTS dell'Editor più recente andando in Installs→Official releases.

Inoltre, se si vogliono modificare i programmi contenuti nelle sottocartelle ros_ws e dds_ws di ros_unity_sim, è necessario prima seguire i seguenti passi. Per ros_ws:

- eliminare le cartelle build, install e log;
- aprire una finestra del terminale ed eseguire il source del file di setup di ROS 2;
- nella stessa finestra del terminale, navigare fino alla cartella ros_ws ed eseguire il comando:

```
$ colcon build
```

Per dds_ws:

- eliminare il contenuto (e solo il contenuto) delle cartelle build nei package dds_unity e dds_tf;
- in una finestra del terminale, per ogni package, navigare nella cartella build ed eseguire i comandi:

```
$ cmake ..
$ make clean && make
```

A.1 Esempio d'uso

Per testare alcuni dei programmi ed il progetto Unity contenuti in ros_unity_sim seguire i seguenti passi:

- Aprire Unity Hub e navigare nella sezione Projects. Se unity_project non compare nella lista dei progetti, importarlo premendo il pulsante Open e selezionando la cartella unity_project in ros_unity_sim.

- Sempre nella sezione Projects di Unity Hub, aprire il progetto unity_project.

- A questo punto si apre l'Editor di Unity. Nella finestra Project dell'Editor navigare in Assets/Scenes e fare doppio click sul file Labs.

- Aprire un nuovo terminale ed eseguire il source del file di setup di ROS 2 e dello workspace ros_ws e lanciare endpoint.py:

```
$ . ~/ros2_foxy/ros2-linux/setup.bash
$ . ~/ros_unity_sim/ros_ws/install/setup.bash
$ ros2 launch ros_tcp_endpoint endpoint.py
```

- Nell'Editor avviare la simulazione premendo il pulsante Play (nella parte alta della finestra).

- Aprire un nuovo terminale ed eseguire il source del file di setup di ROS 2 e dello workspace ros_ws ed eseguire cmd_vel_publisher:

```
$ . ~/ros2_foxy/ros2-linux/setup.bash
$ . ~/ros_unity_sim/ros_ws/install/setup.bash
$ ros2 run unity_pub_sub cmd_vel_publisher
```

  Nella finestra dell'Editor il robot inizierà a muoversi.

- Aprire un nuovo terminale ed avviare l'eseguibile tf_subscriber:

```
$ cd ~/ros_unity_sim/dds_ws/dds_tf/build
$ ./tf_subscriber
```

  Nel terminale verranno stampare le informazioni riguardanti la posizione del robot e la sua odometria.

Appendice B

Cartella ros_unity_sim

Di seguito vengono descritti gli elementi fondamentali contenuti nella cartella ros_unity_sim.

La cartella ros_ws è uno workspace ROS 2 contenente tre package: ROS-TCP-Endpoint, unity_slam_example e unity_pub_sub. Il primo contiene i file relativi al server endpoint descritto nella sezione 3.1; il secondo contiene lo workspace fornito da Unity Technologies e relativo all'utilizzo di Unity per la simulazione di TurtleBot3 con SLAM. il package unity_pub_sub contiene tre programmi:

- cmd_vel_publisher.cpp: implementa un nodo ROS 2 che pubblica messaggi sul topic /cmd_vel con lo scopo di comandare il movimento del robot. Il nome dell'eseguibile è cmd_vel_publisher;

- scan_subscriber.cpp: implementa un nodo ROS 2 che riceve i messaggi pubblicati sul topic /scan e stampa a terminale una parte del loro contenuto. Il nome dell'eseguibile è scan_subscriber;

- image_subscriber.cpp: implementa un nodo ROS 2 che riceve i messaggi pubblicati sul topic /image/compressed e salva in locale le immagini ricevute. Il nome dell'eseguibile è image_subscriber.

La cartella dds_ws è uno workspace eProsima contenente due cartelle: dds_unity e dds_tf. La prima contiene i file necessari a replicare il comportamento dei tre programmi contenuti nel package unity_pub_sub di ros_ws,

ma attraverso eProsima fast DDS anziché ROS 2. I nomi dei file e i relativi eseguibili sono uguali a quelli in ros_ws. Il package dds_tf contiene tf_subscriber.cpp che riceve messaggi sul topic /tf e stampa a terminale una parte del loro contenuto. Il nome dell'eseguibile è tf_subscriber. I file IDL relativi ai messaggi scambiati da questi nodi (e quelli da essi generati da eProsima Fast DDS-Gen) si trovano nelle rispettive cartelle src. Gli eseguibili si trovano nelle cartelle build.

La cartella unity_project è un progetto Unity. È possibile navigare i contenuti di questa cartella direttamente dall'Editor di Unity. Nella cartella Packages sono contenuti i riferimenti a tutti i package importati nel progetto, come ROS TCP Connector (si veda la sezione 3.2). All'interno di Assets si trovano file di varia natura utilizzati all'interno della simulazione, sia creati all'interno dell'Editor, che importati, ad esempio dall'Unity Asset Store. In Assets ci sono file quali i materials, le scene e gli script (si veda la sezione 2.1). Le scene sono tre: SimpleWarehouseScene viene fornita da UnityTechnologies, SlipScene permette di verificare il funzionamento di zone scivolose e Labs è un esempio di simulazione di un ambiente reale, ovvero alcuni laboratori del dipartimento di Ingegneria dell'Università degli Studi di Bergamo. Il modello del robot in SimpleWarehouseScene è quello originale fornito da Unity Technologies, mentre i modelli nelle altre due scene sono stati ampiamente modificati, come descritto nei Capitoli 6, 7 e 8. I file contenuti direttamente nella cartella Scripts sono forniti da Unity Technologies, mentre quelli nelle sottocartelle myControllers e myRGBCamera sono stati implementati, rispettivamente, per modificare la modalità con cui si fa muovere il robot (si veda il Capitolo 7) e per implementare un nuovo sensore, ovvero una camera (si veda il Capitolo 6).

Nella pagina seguente vengono illustrati, all'interno dell'albero della cartella ros_unity_sim, gli elementi appena descritti. L'albero non è completo di tutti i file e cartelle ma mostra la struttura generale e gli elementi di maggiore interesse.

```
ros_unity_sim
├── ros_ws
│   ├── build
│   ├── install
│   ├── log
│   └── src
│       ├── ROS-TCP-Endpoint
│       ├── unity_pub_sub
│       │   └── src
│       │       ├── image_subscriber.cpp
│       │       ├── scan_subscriber.cpp
│       │       └── cmd_vel_publisher.cpp
│       └── unity_slam_example
├── dds_ws
│   ├── dds_tf
│   │   ├── build
│   │   ├── src
│   │   │   └── tf_subscriber.cpp
│   │   └── CMakeLists.txt
│   └── dds_unity
│       ├── build
│       ├── src
│       │   ├── image_subscriber.cpp
│       │   ├── scan_subscriber.cpp
│       │   └── cmd_vel_publisher.cpp
│       └── CMakeLists.txt
└── unity_project
    ├── Assets
    │   ├── Scenes
    │   │   ├── SimpleWarehouseScene.unity
    │   │   ├── SlipScene.unity
    │   │   └── Labs.unity
    │   ├── Scripts
    │   │   ├── myControllers
    │   │   ├── myRGBCamera
    │   │   └── ...
    │   └── ...
    └── Packages
```