



UNIVERSITA' DEGLI STUDI
DELL'AQUILA

Dipartimento di Ingegneria e Scienze
dell'Informazione e Matematica

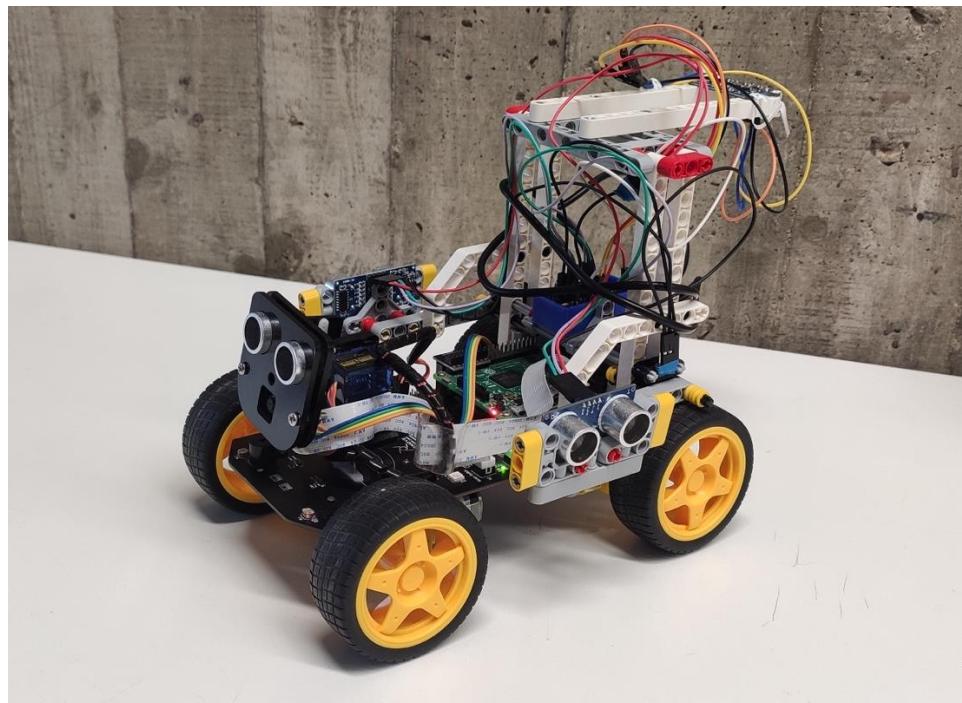


Corso di Laurea Magistrale in Ingegneria Informatica

Intelligent Systems and Robotics Laboratory

Prof. Giovanni De Gasperis

RELAZIONE DEL PROGETTO ROVER-MAZE



Composizione Gruppo:

Alfani Giacomo, Di Stefano Roberto, Paciotti Marta, Pastorelli Matteo

Sommario

Membri del gruppo e rispettivi ruoli	4
Introduzione generale alla relazione	5
Parte Virtuale.....	5
Tecnologie.....	5
CoppeliaSim Edu, Version 4.6.0 (rev.8)	5
Python e sue librerie	5
ROS 2	6
Behaviour Tree.....	6
Architettura Logica	7
Architettura fisica	8
Struttura del codice.....	9
Setup di ROS2.....	11
Behaviour Tree	13
Simulazione virtuale.....	17
Agent	17
Scena simulazione	22
Simulazione step by step	25
Considerazioni finali.....	27
Parte Fisica.....	28
Tecnologie.....	28
• Python e sue librerie	28
• ROS 2	28
• Behaviour Tree	28
• 1x Freenove 4WD Smart Car Kit for Raspberry Pi	28
• 1x Arduino nano	31
• 2x HC-SR04 Ultrasonic Module	32
• 1x Adafruit 9-DOF IMU Breakout	32
Architettura Logica	34
Architettura fisica: Comunicazione.....	35
Setup ROS2	35
Discretizzazione.....	37
Behaviour Tree	37
Architettura fisica: Hardware.....	39
Arduino	39
Struttura del codice	41

Codice Arduino.....	42
Taratura.....	43
Movimento rettilineo.....	43
Rotazione.....	44
Conclusione.....	45
IMU 9 assi	45
Sensori ultrasonici	46
Camera	47
GUI.....	49
Simulazione Fisica	51
Settaggio ambiente di simulazione	51
Taratura	51
Avvio senza GUI:	51
Simulazione step by step	53
Conclusioni finali.....	55

Membri del gruppo e rispettivi ruoli

Giacomo Alfani

Implementazione Arduino

Documentazione

Hardware e setting

Implementazione Python

Tarature

Testing

Roberto Di Stefano

ROS2

Docker

Implementazione Python

Debugging

Raspberry Pi

Testing

Marta Paciotti

Implementazione Arduino

Documentazione

Hardware e setting

Implementazione Python

Tarature

Testing

Matteo Pastorelli

ROS2

Docker

Implementazione Python

Debugging

Esperto Linux

Testing

Introduzione generale alla relazione

Il presente progetto ha lo scopo di realizzare un agente intelligente in grado di muoversi autonomamente all'interno di un labirinto, prima virtuale e poi reale, alla ricerca di due QR Code: il primo, le cui coordinate sono fornite in partenza, è l'obiettivo iniziale da raggiungere da parte dell'agente e contiene le coordinate del secondo QR Code e una chiave per decodificarne il contenuto; il secondo costituisce invece l'obiettivo finale, al raggiungimento del quale l'agente avrà raggiunto il suo scopo ultimo e si fermerà.

Parte Virtuale

In questa sezione verrà illustrata la parte virtuale del progetto Rover-Maze e di come si è cercato di riprodurre il più fedelmente possibile, all'interno della simulazione virtuale, quelle che sarebbero state le condizioni di simulazione fisiche, al fine di poter passare dalla prima alla seconda il più facilmente possibile, riutilizzando allo stesso tempo quanti più componenti già utilizzati e progettati.

Nello specifico vedremo le tecnologie utilizzate nel progetto, l'architettura logica e fisica adottata e tutti i dati relativi alla simulazione vera e propria; infine, si troveranno delle considerazioni a conclusione della sezione.

Tecnologie

Nel progetto sono state utilizzate le seguenti tecnologie:

CoppeliaSim Edu, Version 4.6.0 (rev.8)



CoppeliaSim è un software di simulazione multiuso e multipiattaforma sviluppato da Coppelia Robotics, è utilizzato per simulare e sviluppare sistemi robotici complessi in diversi settori. Permette di progettare, testare e ottimizzare algoritmi di controllo per robot, creare ambienti di simulazione realistici e analizzare il comportamento dei robot in una vasta gamma di scenari.

Nella parte virtuale di questo progetto, CoppeliaSim ha consentito di poter ricreare un ambiente simulato in cui testare il funzionamento del sistema attraverso il supporto dell'ambiente 3D messo a disposizione dal software, che ha consentito l'inserimento di elementi di diverse tipologie a modificarli e settarli a seconda delle necessità progettuali.

Python e sue librerie



Per la realizzazione del progetto si è scelto di utilizzare come linguaggio di programmazione Python, per la sua semplicità e versatilità. È stato inoltre necessario l'utilizzo delle seguenti librerie:

- **remoteAPIClient**: utilizzata per la comunicazione tra Python e CoppeliaSim;
- **py_trees**: utile al fine dell'implementazione dei behaviour tree;

- **Io**: fornisce le interfacce Python per la gestione degli stream, come file di input/output e altre operazioni di streaming, è stato utilizzato per l'elaborazione delle immagini acquisite dall'agente;
- **pillow**: consente l'elaborazione delle immagini in Python; offre una vasta gamma di funzionalità per la manipolazione e la modifica delle immagini, in particolare è stata utilizzata per l'identificazione dei QR code presenti nell'ambiente simulato;
- **numpy**: permette l'elaborazione efficiente di array multidimensionali e operazioni matematiche ad alte prestazioni, utilizzata nelle porzioni di codice in cui è stata necessaria una computazione come, ad esempio, nella gestione del grafo e dei suoi nodi;
- **pyzbar**: utile per leggere QR code o bar code, in questo progetto utilizzata per leggere i due QR code presenti nell'ambiente virtuale;
- **time**: permette la gestione del tempo in Python e fornisce funzioni per gestire temporizzazioni, misurare il tempo e convertirlo tra diverse rappresentazioni;
- **json**: per la codifica e la decodifica di dati JSON (JavaScript Object Notation) in Python, che consente di scambiare dati tra applicazioni in un formato leggibile sia per umani che per macchine; in particolare è stato utile nella manipolazione di dati forniti come quelli, ad esempio, provenienti dai sensori di prossimità;
- **base64**: per la codifica e la decodifica di dati binari in base64, utile per la rappresentazione di dati binari in formato testuale, ad esempio per l'invio di dati attraverso protocolli di comunicazione che richiedono testo; in particolare è stato utilizzato per la codifica della chiave di cifratura del QR code finale.

ROS 2



ROS2, o Robot Operating System 2, è una piattaforma open-source per lo sviluppo di software per robotica. È la versione successiva di ROS (Robot Operating System) e offre miglioramenti significativi rispetto alla sua controparte precedente. ROS2 è progettato per essere più robusto, flessibile e adatto a una più ampia gamma di applicazioni robotiche, inclusi sistemi in tempo reale e distribuiti. Nel progetto presentato è stato utilizzato come base di tutta l'architettura, permettendo di creare la struttura a nodi, adatta alle necessità progettuali, e gestendo la comunicazione tra questi ultimi, svolgendo quindi anche la funzione di message broker.

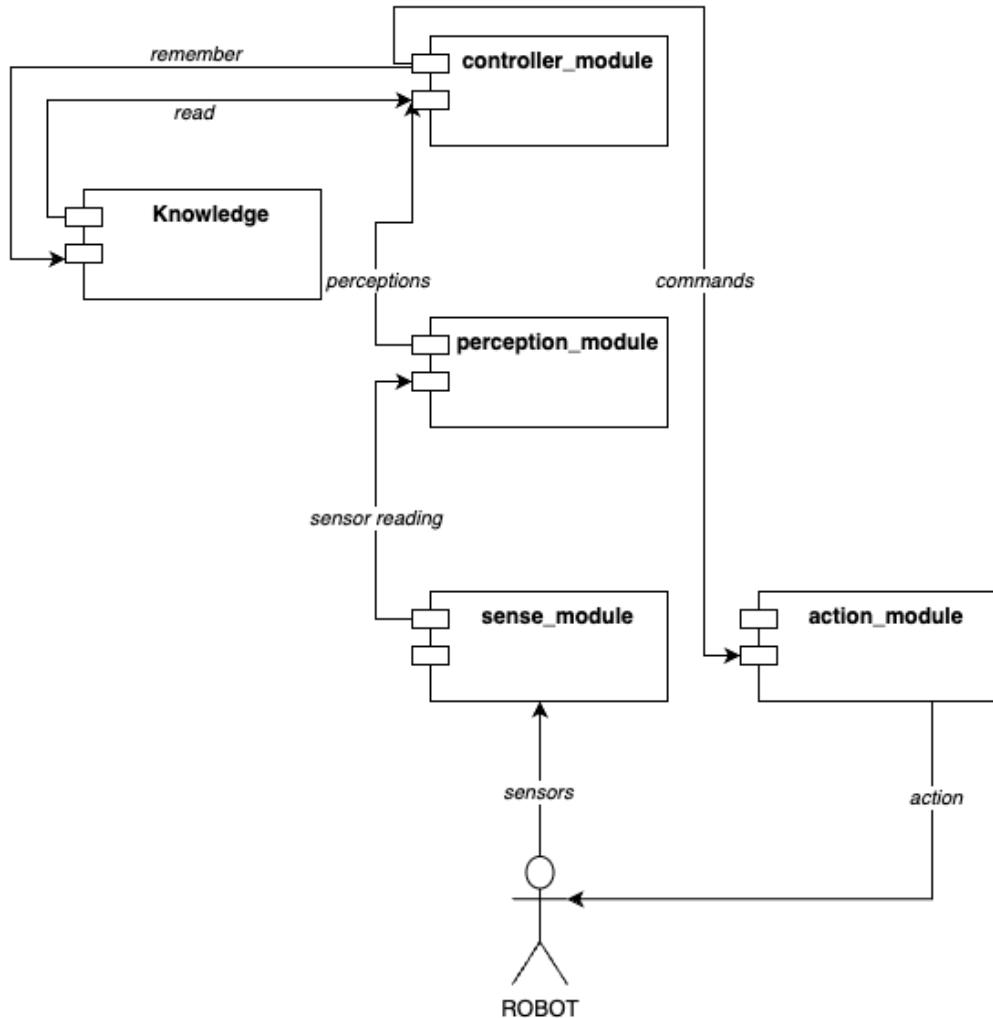
Behaviour Tree

Un albero comportamentale, noto anche come albero decisionale, costituito da una struttura gerarchica composta da nodi, che rappresentano comportamenti elementari o complessi, ed archi che indicano le relazioni di dipendenza tra i nodi. Questo è spesso utilizzato nell'intelligenza artificiale e nella robotica per definire e organizzare il comportamento di un agente o di un sistema autonomo. I vantaggi nel suo utilizzo sono sicuramente la modularità, la riusabilità, la flessibilità e la facilità nel fare debugging.

In questo lavoro è stato utilizzato per la realizzazione della logica del componente Controller, che consiste nel prendere decisioni circa il percorso che l'agente ha già svolto o che deve svolgere all'interno dell'ambiente simulato, la direzione che esso deve percorrere di conseguenza e l'analisi dei goal dell'agente stesso.

Architettura Logica

Come illustrato nella figura seguente, l'architettura logica è organizzata in 4 moduli principali:



- **action_module:** è il modulo che rappresenta il corpo simulato del robot ed esegue le azioni impartite dal controller_module, che gli comunica una tra le seguenti azioni da eseguire: *go_forward, turn_left, turn_right, stop*;
- **sense_module:** è il modulo che riceve le letture dalla camera e dai sensori di prossimità e orientamento e li fornisce al perception_module;
- **perception_module:** è il modulo intermedio tra il sense_module ed il controller_module che svolge la funzione di elaborare i dati provenienti dal sense_module e restituire informazioni ad un livello più alto al controller_module, in particolare i nodi liberi o occupati, espressi relativamente alla posizione dell'agent, l'orientamento di quest'ultimo, e i dati letti dai QR code, nel caso in cui ne venga identificato uno;
- **controller_module:** è il modulo che funge da cervello del robot e si occupa dell'elaborazione in più fasi delle informazioni provenienti dal perception_module e del loro salvataggio nel sotto-modulo Knowledge, così da poter essere riutilizzate per prendere decisioni in futuro;
 - **knowledge:** sotto-modulo destinato all'immagazzinamento di informazioni, selezionate dal controller_module, per poter essere accedute in futuro allo scopo di

prendere decisioni. Tra i dati che contiene c'è la rappresentazione della mappa, realizzata attraverso un grafo non orientato e non pesato il cui utilizzo verrà spiegato in dettaglio nella sezione Behaviour Tree.

Architettura fisica



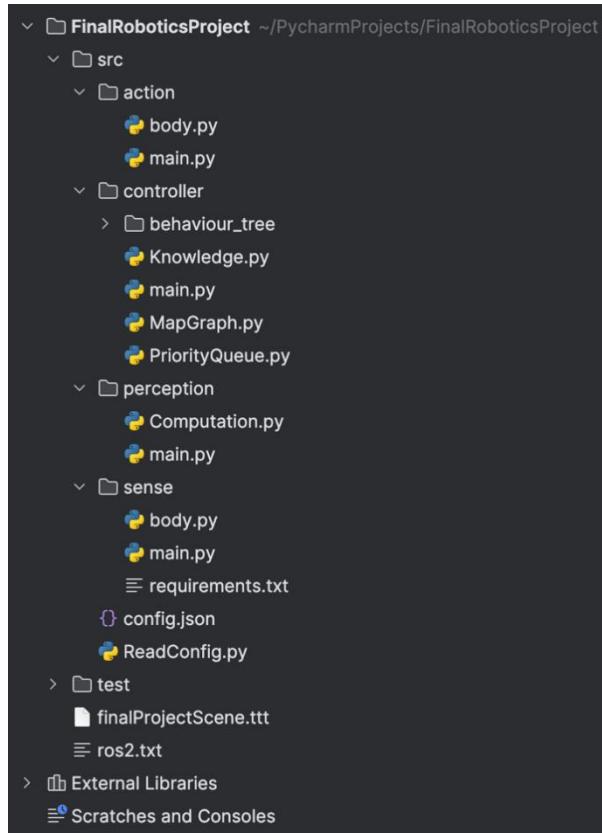
Per la realizzazione dell'architettura fisica, come già accennato in precedenza, si è deciso di utilizzare ROS2, creando un nodo per ciascun modulo logico visto in precedenza (**sense_node**, **perception_node**, **controller_node**, **action_node**) ed un diverso topic per ciascun tipo di dato scambiato come segue:

- **camera_sensor**: topic che si occupa di ricevere e fornire le informazioni riguardanti il visual sensor;
- **proximity_sensors**: topic che si occupa di ricevere e fornire le informazioni riguardanti i sensori di prossimità;
- **orientation_sensor**: topic che si occupa di ricevere e fornire le informazioni riguardanti l'orientamento;
- **arrived**: topic che si occupa di ricevere e fornire le informazioni circa il raggiungimento dell'obiettivo;

- **free_side**: topic che si occupa di ricevere e fornire le informazioni circa i nodi liberi intorno al robot;
- **orientation**: topic che si occupa di ricevere e fornire le informazioni circa la direzione in cui il robot è orientato;
- **action_topic**: topic che si occupa di ricevere e fornire le informazioni circa la successiva azione da compiere da parte del robot.

Struttura del codice

Nell'immagine seguente è mostrata la struttura che è stata scelta nell'organizzazione del codice:



- **action**:
 - **main.py**: definisce la classe Subscriber sottoclasse di Node della libreria ufficiale di ROS (rclpy) che si iscrive a un topic di azione (action_topic) e esegue azioni specifiche in risposta ai messaggi ricevuti su quel topic. Le azioni che può eseguire sono: *go_forward, turn_left, turn_right, stop*;
 - **body.py**: definisce una classe SimulatedPioneerBody che rappresenta il corpo del robot simulato in CoppeliaSim. Questa classe fornisce metodi per eseguire azioni sul corpo del robot simulato, come impostare la velocità dei motori e avviare o fermare la simulazione;
- **controller**:
 - **behaviour_tree**: mantiene una classe (BehaviourTree) dove viene inizializzato il behavior tree e tramite la quale si avvia il tick , e una classe per ciascun nodo del

behavior tree. Il ruolo e funzionamento di ciascun nodo verrà meglio spiegato nella sezione Behavior Tree di questa documentazione;

- **Knowledge.py:** definisce la classe Singleton omonima che funge da contenitore per le informazioni scambiate tra i diversi nodi del behavior tree. Le informazioni gestite includono la posizione attuale del robot, il grafo della mappa, il nodo obiettivo, l'orientamento, il tempo di inizio dell'azione, il percorso pianificato e altre variabili utili per portare a termine i compiti del controller. ciascun nodo legge i dati a lui necessari da tale classe, li elabora e infine salva il dato elaborato. In questo modo partendo dai dati provenienti dal modulo Perception si determina l'azione da eseguire e tutto ciò viene fatto in una serie di step mantenendo in questo modo la logica del singolo nodo semplice ed evitando allo stesso tempo un accoppiamento diretto tra i diversi nodi;
 - **main.py:** definisce la classe Controller sottoclasse di Node della libreria ufficiale di ROS (rclpy). Tale classe è responsabile della comunicazione con i moduli Perception e Action. Una volta che riceve i dati, provenienti dal modulo sottostante, li salva nella classe Knowledge e avvia il tick del behavior tree. Quando uno dei nodi del behavior tree ha determinato l'azione da eseguire chiama il metodo “perform_action” della classe Controller che la pubblica su “action_topic”;
 - **MapGraph.py:** definisce la classe omonima che fornisce una struttura per gestire la rappresentazione della mappa e delle operazioni associate ad essa come, ad esempio, path_to_next_node calcola il percorso dal nodo corrente al prossimo attraverso una BFS nel grafo o add_node e remove_node che aggiungono ed eliminano rispettivamente nodi dal grafo;
 - **PriorityQueue.py:** definisce la classe omonima che implementa una coda con priorità si è deciso di implementare tale struttura dati invece di utilizzare l'omonima classe della libreria queue in quanto quest'ultima non forniva un metodo per l'aggiornamento delle priorità;
- **perception:**
 - **computation.py:** modulo responsabile dell'elaborazione dei dati provenienti dal modulo Sense. In particolare, elabora le immagini per individuare eventuali QR-code e utilizza i dati di prossimità per determinare la posizione relativa dei nodi vicini liberi o occupati;
 - **main.py:** definisce la classe Perception sottoclasse di Node della libreria ufficiale di ROS (rclpy). Tale classe ha la responsabilità di gestire la comunicazione con i moduli Sense e Controller. Una volta che riceve i dati provenienti al modulo sottostante li elabora grazie alla classe Computation e li pubblica in modo da renderli accessibili al modulo superiore;
 - **sense:**
 - **body.py:** definisce la classe MyNode sottoclasse di Node della libreria ufficiale di ROS (rclpy). Tale classe è responsabile della comunicazione con il modulo Perception. Ogni 0.1 secondi legge i dati dei sensori grazie alla classe SimulatedPioneerBody e li pubblica per renderli accessibili al modulo sovrastante;
 - **main.py:** definisce una classe SimulatedPioneerBody che rappresenta il corpo del robot simulato in CoppeliaSim. Questa classe fornisce metodi leggere i sensori del robot simulato, come il vision sensor la bussola;

- **ReadConfig**: classe che legge i dati da un file di configurazione JSON chiamato "config.json". La classe ha un attributo `_file_content` che memorizza il contenuto del file JSON come un dizionario, semplificando la lettura dei dati;
- **config.json**: file che contiene una serie di parametri di configurazione in formato JSON.

```
{
    "PRIORITY_WEIGHT": 0.87,      Valore compreso tra 0-1 rappresenta il peso della vicinanza all'obiettivo per il calcolo priorità rispetto alla lontananza dagli ostacoli
    "SPEED": 0.3,                Rappresenta la velocità con la quale il robot si muove quando va dritto
    "SPACE": 0.5,                Rappresenta la distanza tra la posizione del robot e i nuovi nodi aggiunti alla mappa
    "FREE_SIDE_THRESHOLD": 0.5,   Rappresenta la distanza minima a cui si deve trovare un ostacolo per definire il lato libero
    "STRING_TARGET_ARRIVE": "finish",   Rappresenta la parola mantenuta nel QR-code finale
    "NODE_DISTANCE_THRESHOLD": 0.4,  Distanza minima tra il centro di due nodi perché si possano dire diversi (raggio del nodo)
    "AM_I_IN_NEXT_NODE_THRESHOLD": 0.2,  Distanza massima tra il robot e il centro di un nodo tale per cui si possa dire che il robot si trovi nel nodo
    "START_GOAL": [2, -6.5]     Posizione del primo QR code da raggiungere
}
```

Setup di ROS2

Per utilizzare ROS2 si è proceduto creando un container Docker partendo dall'immagine con tag latest della repository ufficiale di ROS, reperibile al seguente link: https://hub.docker.com/_/ros/tags, e configurandola come riportato nei passi sottostanti; alternativamente si può scaricare un'immagine già configurata a questo link: <https://hub.docker.com/r/robbobby24/ros2/tags>.

Passi e comandi da eseguire per la configurazione di ROS2:

1. Scaricare l'immagine sopracitata dalla repository ufficiale ROS2:

```
docker pull ros:latest
```

2. Creare ed eseguire il run del container Docker:

- Se si utilizza Linux:

```
docker run -it -p 6080:80 --security-opt seccomp:unconfined --add-host=host.docker.internal:host-gateway --name container_name ros:latest
```

- Altrimenti:

```
docker run -it -p 6080:80 --security-opt seccomp:unconfined --name container_name ros:latest
```

3. Creare il workspace di ROS2:

```
cd
mkdir -p ~/workspace_name/src
cd ~/workspace_name/src
colcon build
```

5. Attivare il workspace:

```
source workspace_name/install/setup.bash
```

6. Inserire attivazione workspace all'apertura del terminale (opzionale):

```
cd  
nano .bashrc
```

Inserire come ultima riga all'interno del file .bashrc il comando del punto 4;

7. Creare il package:

```
ros2 pkg create pkg_name --build-type ament_python --dependencies rclpy std_msg
```

8. Fare il build del package:

```
cd ~/workspace_name  
colcon build --symlink -install
```

In caso di errori eseguire il seguente:

```
pip3 install setuptools==58.2.0
```

9. Creazione nodo:

- nel seguente path creare un file Python e scrivere il codice del nodo:

```
~/workspace_name/src/pkg_name/ pkg_name
```

- Modificare il file:

```
~/workspace_name/src/pkg_name/setup.py
```

Modificare il dizionario entry_points:

```
entry_points={  
    'console_scripts': [  
        'sense_module = isrlab_project.sense.main:main',  
        'action_module = isrlab_project.action.main:main',  
        'perception_module = isrlab_project.perception.main:main',  
        'controller_module = isrlab_project.controller.main:main'  
    ],  
},
```

Inserendo al posto di sense_module il nome del nodo desiderato, al posto isrlab_project.sense.main il path del file creato in precedenza e al posto di :main il nome della funzione che si vuole eseguire (lo stesso vale per i restanti moduli).

10. Lanciare il nodo:

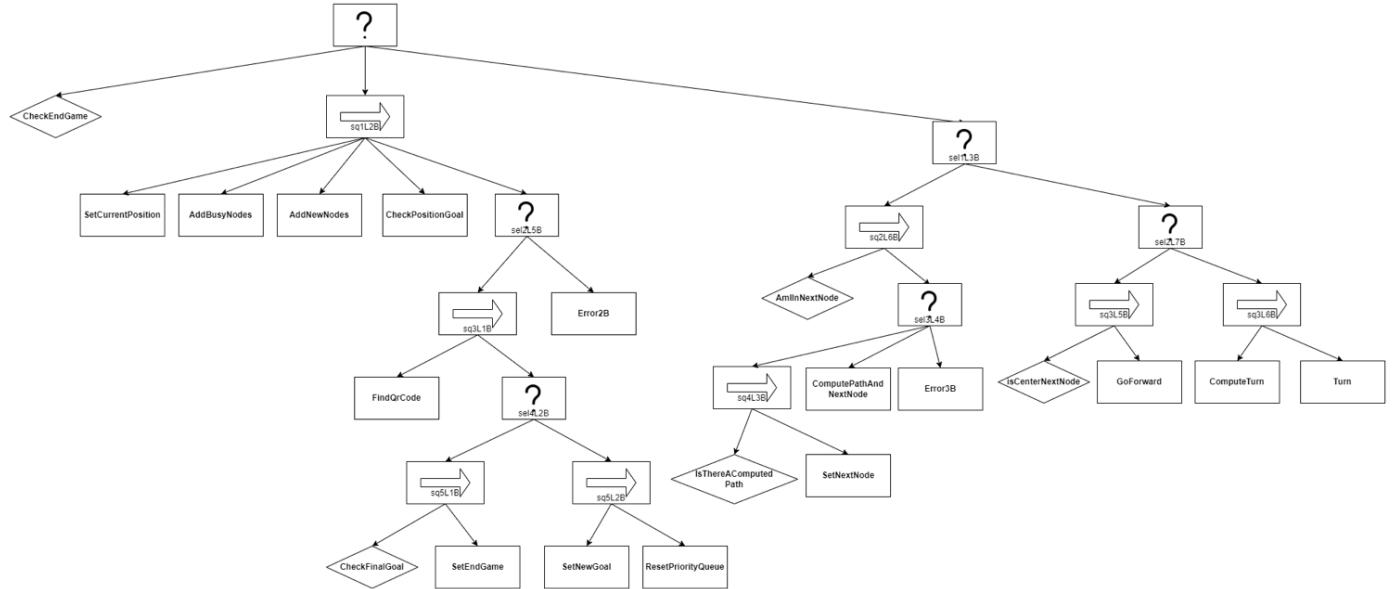
```
ros2 run name_pkg name_node
```

11. Il codice dei moduli lo si può trovare alla seguente repository github:

<https://github.com/Matteopast01/FinalRoboticsProject>

Behaviour Tree

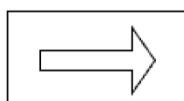
Come già introdotto in precedenza, nel progetto è stato utilizzato il Behaviour Tree per gestire tutti i comportamenti dell'agente all'interno dell'ambiente simulato.



Nella figura soprastante vediamo quello che è stato utilizzato all'interno del controller, in cui possiamo notare i seguenti elementi di base che lo costituiscono:



Selector o Fallback: un nodo di controllo che esegue i suoi figli uno alla volta fino a quando uno di essi restituisce *Success*. Se nessuno restituisce *Success* si restituisce *Failure*. È utilizzato per definire un insieme di azioni alternative, dove una qualsiasi delle azioni può essere eseguita.



Sequence: un nodo di controllo che esegue i suoi figli uno alla volta in ordine. Se uno di essi restituisce *Failure*, la sequenza si interrompe restituendo anch'essa *Failure*. Se tutti i figli restituiscono *Success*, la sequenza restituisce *Success*. È utile per definire una sequenza di azioni che devono essere eseguite in successione.



Action: un nodo di esecuzione che rappresenta compiti o azioni specifiche che l'agente deve eseguire. Svolgono una funzione specifica e restituiscono *Success*, *Failure* o *Running*.



Condition: un nodo di esecuzione che permette di valutare una condizione o uno stato dell'ambiente e restituisce un risultato booleano *True* o *False*. Vengono utilizzati per prendere decisioni basate sullo stato corrente del sistema.

Infine le frecce rappresentano gli **archi** dell'albero ed indicano il flusso da un nodo all'altro.

Di seguito, invece, si possono trovare tutti i nodi dell'albero più in dettaglio e comprendere come agiscono:

CheckEndGame: controlla nella knowledge la variabile booleana end_game.

Se *True* restituisce *Success* altrimenti *Failure*.

SetCurrentPosition:

- **input**: orientamento, azione in esecuzione, posizione precedente, tempo inizio azione attuale, tempo attuale, costante velocità;
- **output**:
 - se vado dritto: spazio=(tempo attuale - tempo inizio azione attuale) * costante velocità; current position x = x + spazio per cos(orientamento); current position y = y + spazio per sin(orientamento);
 - altrimenti: posizione attuale = posizione precedente.

Restituisce sempre *Success*.

AddBusyNode:

- **input**: delta x, delta y (di ciascun nodo trovato occupato per ognuna delle direzioni avanti, destra e sinistra), posizione attuale;
- **output**: posizione dei nodi occupati (posizione attuale più delta x e y);
- **azione**: aggiunge i nuovi nodi alla lista dei nodi occupati.

Restituisce sempre *Success*.

AddNewNodes:

- **input**: delta x, delta y (di ciascun nodo trovato libero per ognuna delle direzioni avanti, destra e sinistra), posizione attuale;
- **output**: posizione dei nuovi nodi (posizione attuale più delta x e y);
- **azione**: aggiunge i nuovi nodi al grafo.

Restituisce sempre *Success*.

CheckPositionGoal:

- **input**: posizione attuale, posizione obiettivo;
- **azione**: ferma il robot se la distanza tra l'obiettivo e la posizione attuale è minore di una NODE_DISTANCE_THRESHOLD (parametro di configurazione).

Restituisce *Success* se la distanza tra l'obiettivo e la posizione attuale è minore di una NODE_DISTANCE_THRESHOLD (parametro di configurazione).

FindQrCode:

- **input**: primo parametro JSON da Perception (QR Code trovato o meno);
- **azione**: gira su sé stesso ad ogni tick.

Restituisce *Success* se il primo parametro del JSON è *True*.

Restituisce *Running* se il primo parametro del JSON è *False*.

CheckFinalGoal:

- **input**: terzo parametro JSON da Perception (testo presente nel QR Code);
- **output**: testo (chiave per il prossimo QR Code oppure testo da decodificare).

Restituisce *Success* se nel terzo parametro (decodificato tramite chiave precedentemente salvata nella knowledge) c'è scritto una stringa specifica fissata STRING_TARGET_ARRIVE (parametro di configurazione).

Restituisce, altrimenti, *Failure*.

SetEndGame:

- **azione**: setta la variabile end_game nella knowledge a *True*, manda all'action il segnale di stop.

Restituisce sempre *Success*.

SetNewGoal:

- **input**: secondo parametro di JSON da Perception (posizione del nuovo obiettivo);
- **output**: nuovo obiettivo uguale al secondo parametro di JSON.

Restituisce sempre *Success*.

ResetPriorityQueue:

- **input**: nuovo obiettivo;
- **output**: nuova priority queue;
- **azione**: aggiungi tutti i nodi del grafo alla priority queue secondo la nuova priorità (la priorità dipende dalla distanza in linea d'aria rispetto all'obiettivo e dalla distanza media dai nodi occupati).

Restituisce sempre *Success*.

Error:

- **azione**: setta la variabile end game nella knowledge a *True*.

Restituisce sempre *Success*.

IsThereAComputedPath:

- **input**: variabile path della knowledge.

Restituisce *Success* se la variabile path ha almeno un elemento.

Restituisce *Failure* altrimenti.

AmIInNextNode:

- **input**: posizione attuale, next node.

Restituisce *Success* se la distanza tra il next node ed il current node è minore di AM_I_IN_NEXT_NODE_THRESHOLD (variabile di configurazione).

Restituisce *Failure* altrimenti.

ComputePathAndNextNode:

- **input:** posizione attuale, grafo, coda con priorità dei nodi da visitare;
- **output:** calcola cammino per raggiungere il prossimo nodo (usa BFS da current node a next_node);
- **azione:** estrae il next_node dalla coda con priorità.

Restituisce *Success* se esiste un nodo nella pila con un cammino per raggiungerlo.

Restituisce *Failure* altrimenti.

ComputeAction:

- **input:** Orientation, next Node
- **output:** determina la prossima azione da eseguire;
- **azione:** setta action nella knowledge.

Restituisce sempre *Success*.

PerformAction:

- **input:** variabile action della knowledge;
- **azione:** pubblica l'azione al modulo action (se l'azione è go_forward ed il nodo davanti non è libero resetta il next_node e non esegue l'azione).

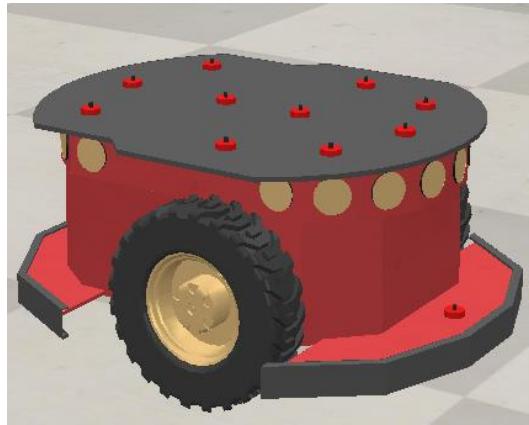
Restituisce *Success* sempre.

Simulazione virtuale

Nella simulazione il robot partirà da una posizione prefissata all'interno di un labirinto per poi dirigersi, autonomamente, alla ricerca di due QR Code: il primo, le cui coordinate sono fornite in partenza, è l'obiettivo iniziale da raggiungere da parte del robot e contiene le coordinate del secondo QR Code e una chiave per decodificarne il contenuto; il secondo costituisce invece l'obiettivo finale, al raggiungimento del quale l'agente avrà ottenuto il suo scopo ultimo e si fermerà.

Agent

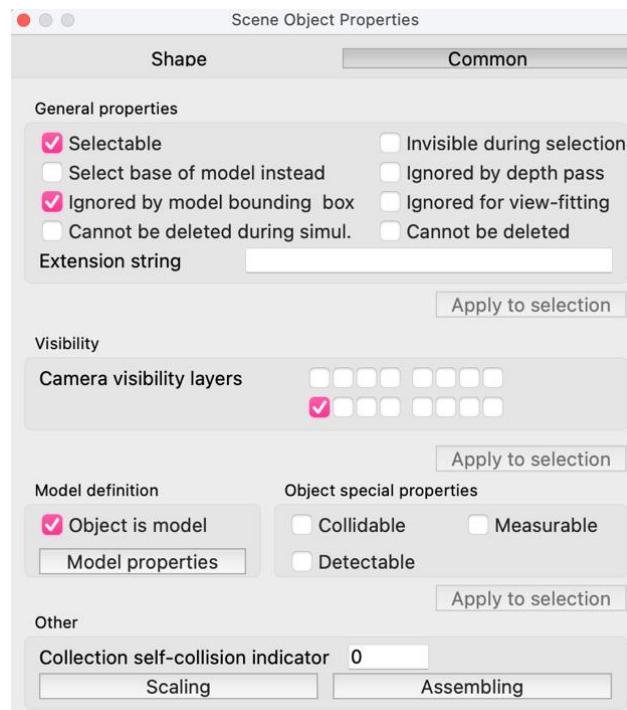
Date le esigenze progettuali, si è scelto come agent per il progetto un PioneerP3DX, visibile nell'immagine seguente:



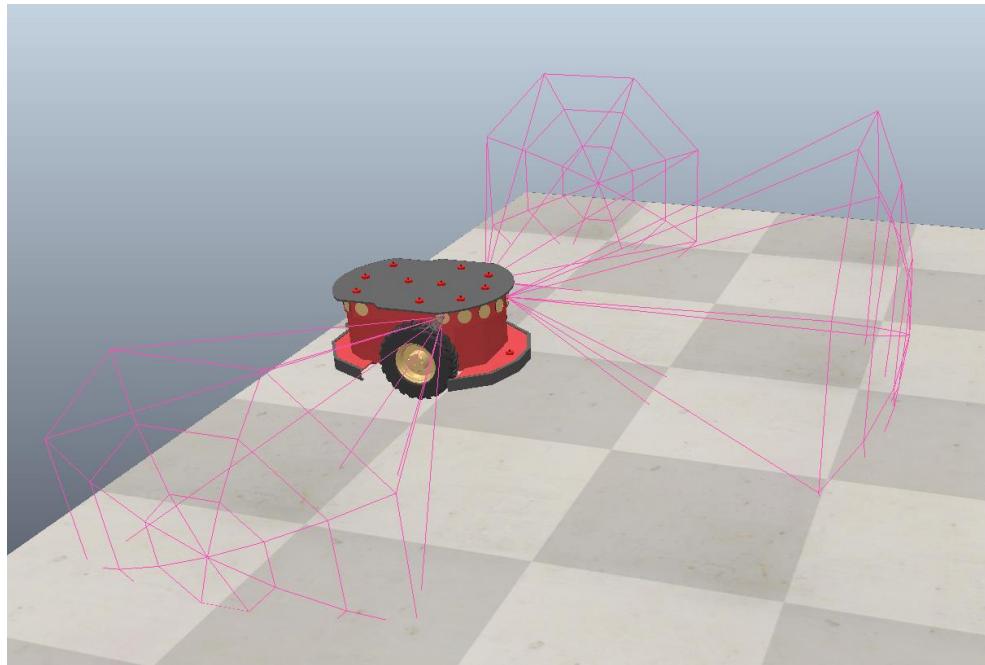
Esso è dotato di default di:

- 2 ruote;
- 16 ultrasonicSensor, numerati da 0 a 15.

Vengono riportati di seguito i settaggi scelti per il Pioneer:



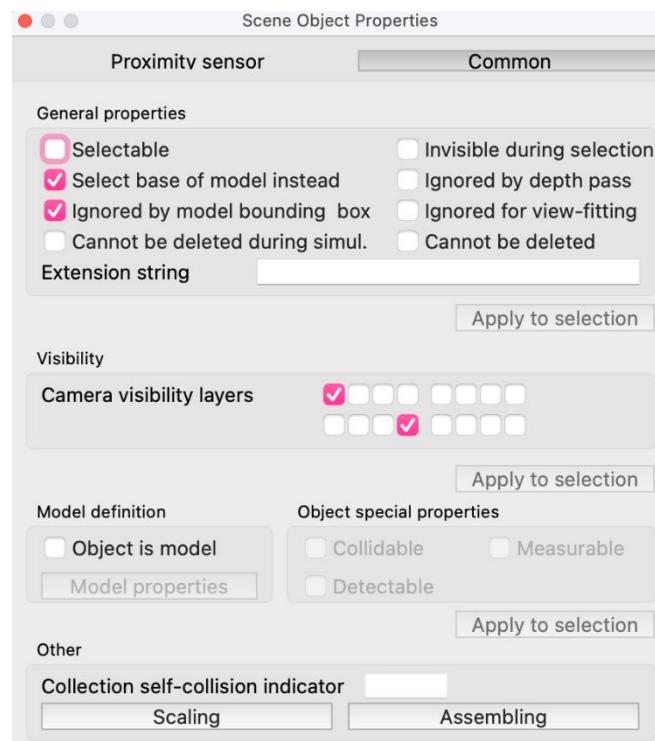
Si è quindi scelto di utilizzare solo 3 dei suoi sensori di prossimità dei 16 presenti, così come da figura:



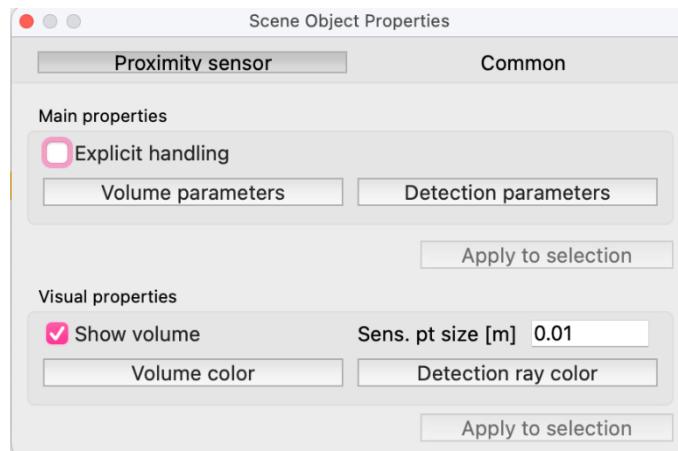
In particolare, si utilizzano i sensori **ultrasonicSensor[0]**, **ultrasonicSensor[3]**, **ultrasonicSensor[6]**, poiché era necessario riuscire ad avere informazioni solo di ciò fosse presente nell'ambiente nella parte anteriore, destra e sinistra del Pioneer.

I settaggi **dell'ultrasonicSensor[0]** e **dell'ultrasonicSensor[6]** sono riportati a seguire:

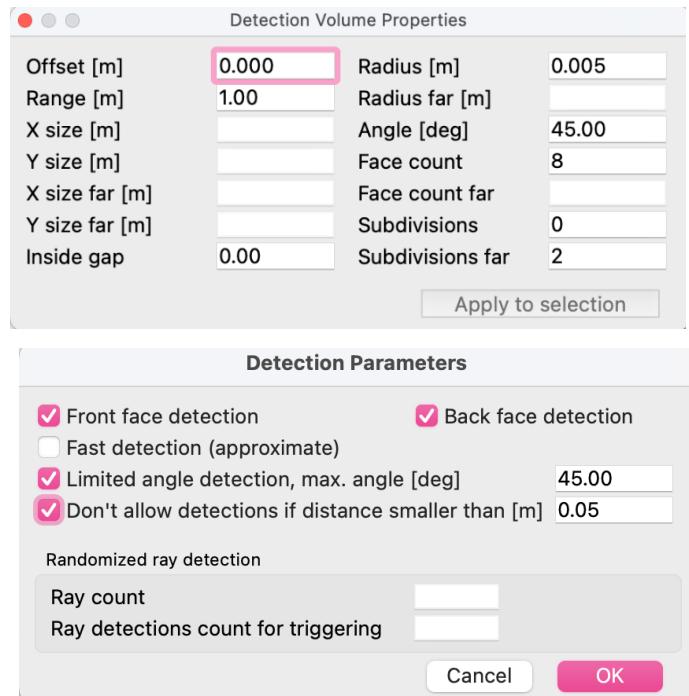
nella Scene Object Properties abbiamo, nella sezione Common:



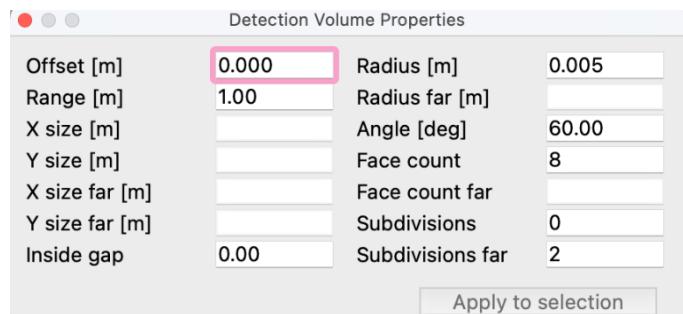
mentre nella sezione Proximity sensor i seguenti:

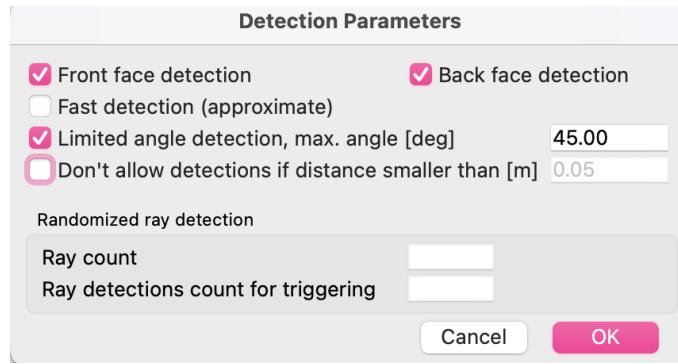


Inoltre, tra le sezioni presenti al suo interno, sono state personalizzate Detection Volume Properties e Detection Parameters:

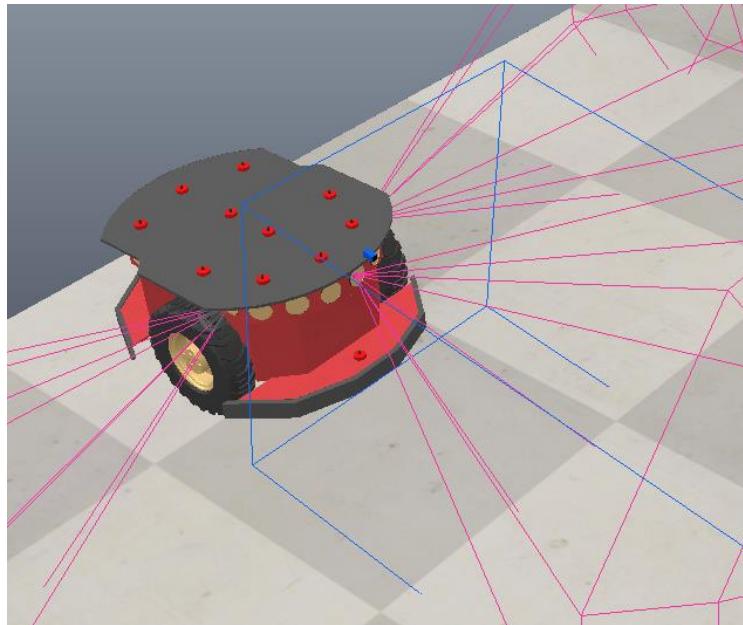


Per quanto riguarda l'**ultrasonicSensor[3]** ciò che cambia, rispetto all'**ultrasonicSensor[0]**, sono i settaggi di Detection Volume Properties e Detection Parameters come segue:

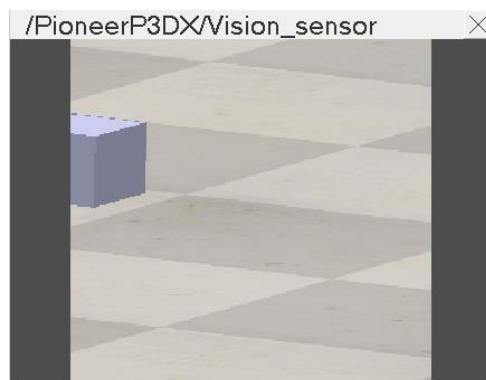




Si è, inoltre, aggiunto un **Vision_sensor** nella parte frontale del Pioneer:

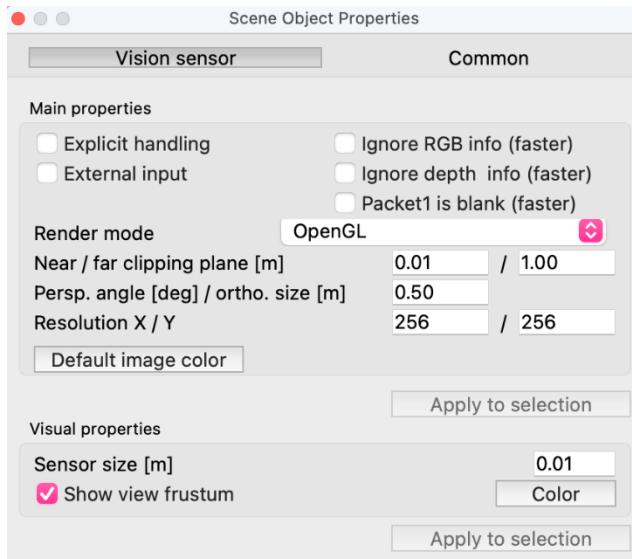
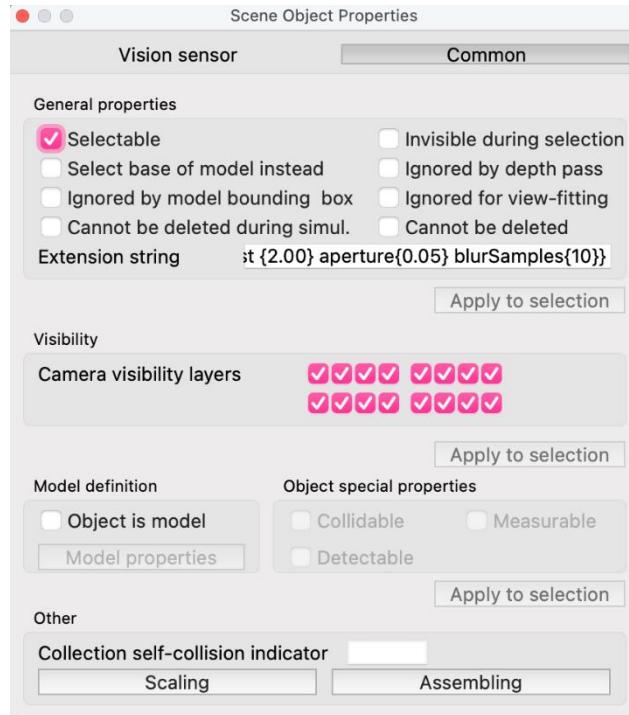


Il suo utilizzo è stato necessario perché esso fornisce la possibilità all'agente di poter acquisire immagini dalla fotocamera messa a disposizione, capacità fondamentale per poter identificare i QR



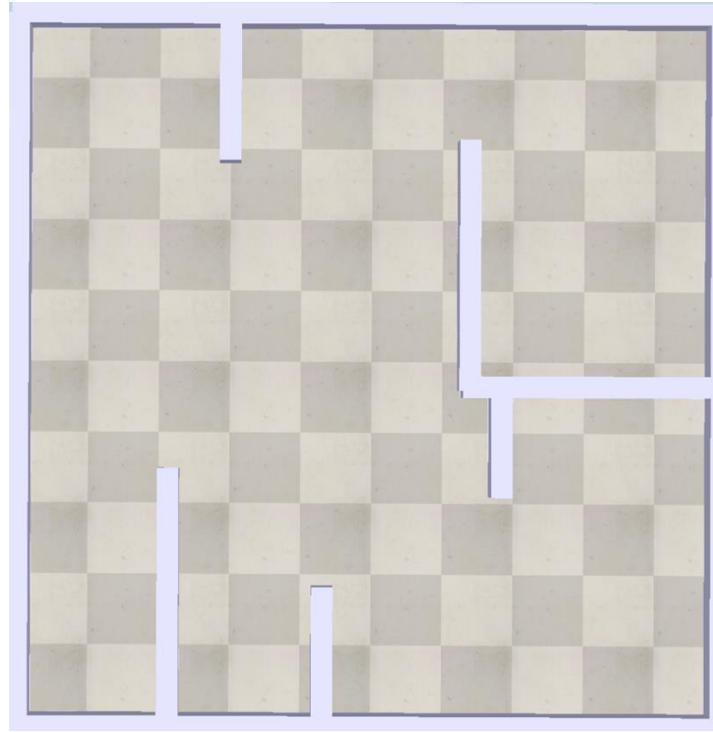
code presenti nell'ambiente. Nell'ambiente di simulazione viene, inoltre, messa a disposizione una finestra che mostra ciò che il Vision_sensor riprende in quel momento:

I settaggi del Vision_sensor sono i seguenti:

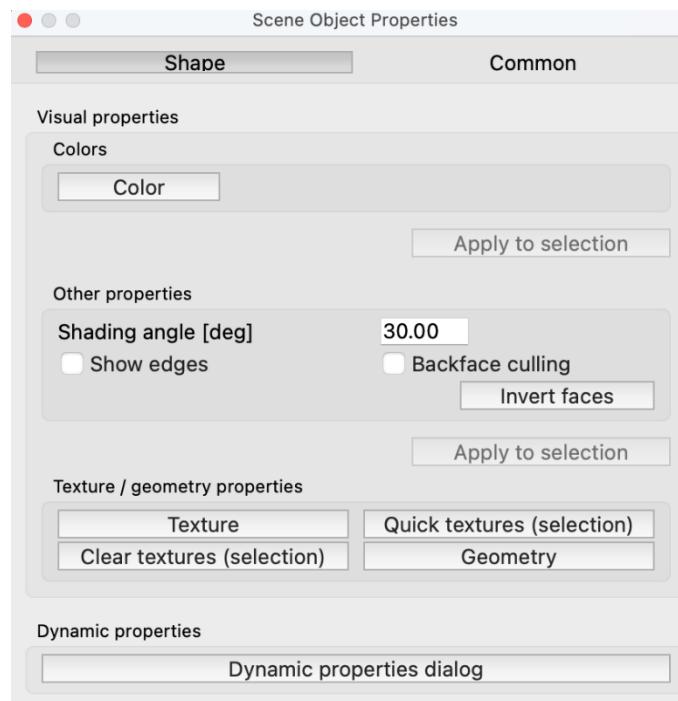


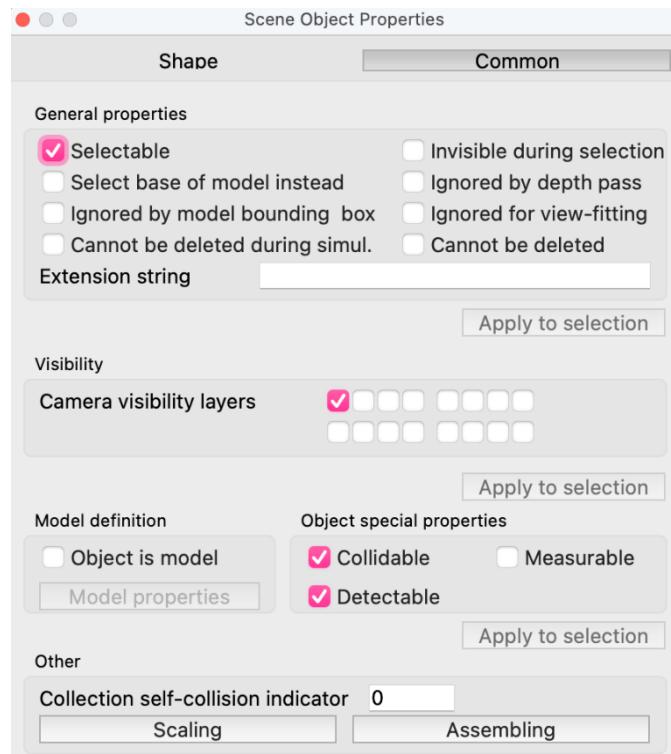
Scena simulazione

Per creare l'ambiente di simulazione virtuale, si è sfruttato l'editor di CoppeliaSim per realizzare un labirinto, ottenendo il seguente risultato:

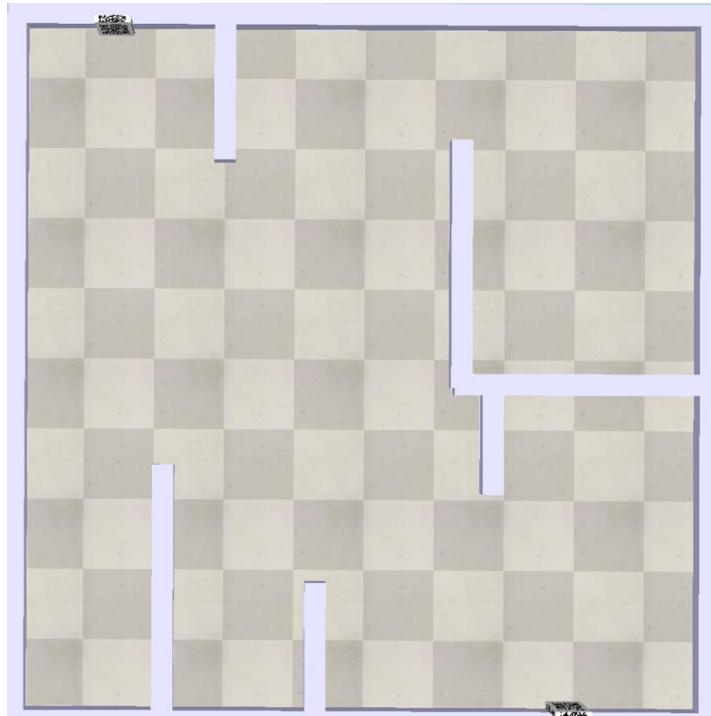


Il labirinto è stato creato componendo tra loro elementi messi a disposizione dal simulatore, in particolare 9 Cuboid, con i seguenti settaggi:

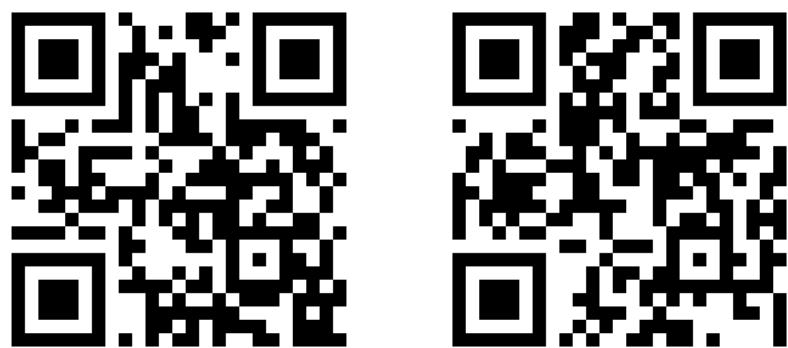




In seguito, si sono inseriti 2 Cuboid aggiuntivi, ai quali è stata modificata la texture standard sostituendola con quella dei 2 QR code che il Pioneer dovrà leggere:



Di seguito ci sono i 2 QR code che sono stati inseriti, dove quello sinistro è il primo che viene identificato, e il destro quello finale:

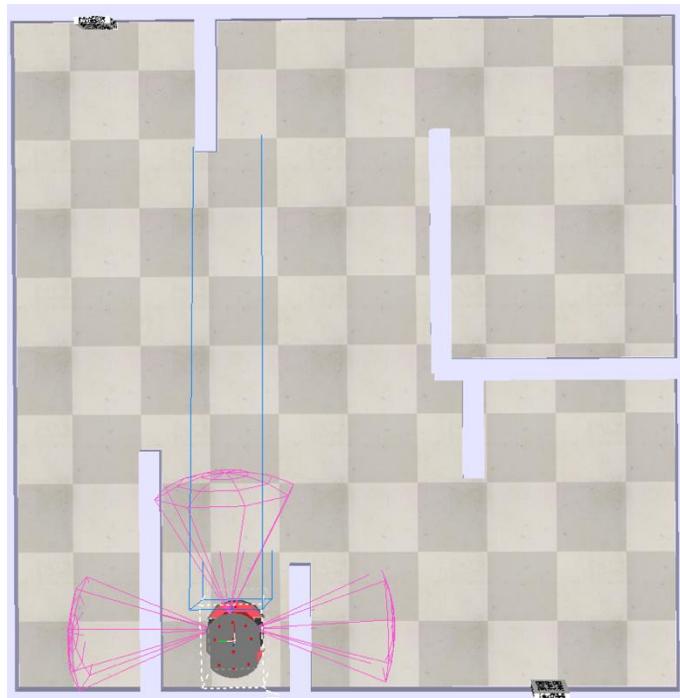


Il loro contenuto è il seguente:

QR Code iniziale: **10.#2.8#DQwXAhYR**

QR Code finale: **10.#2.8#key.png**

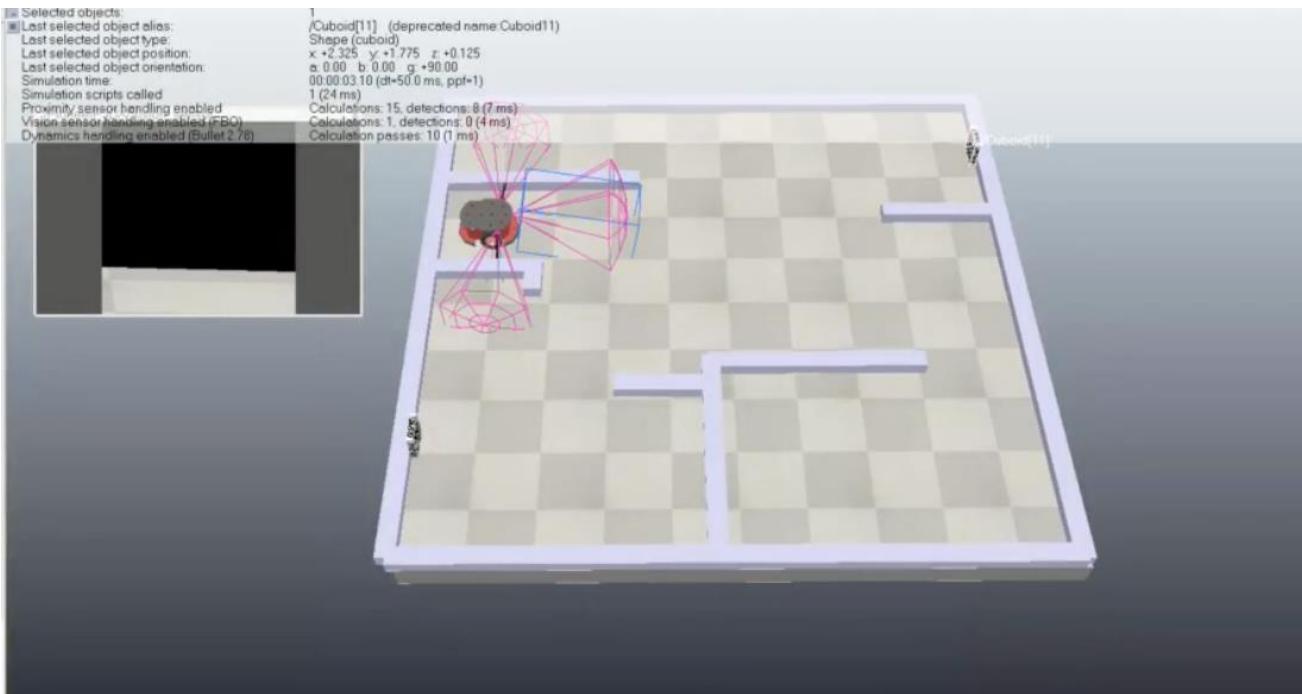
Il risultato finale dell'ambiente di simulazione è il seguente:



Simulazione step by step

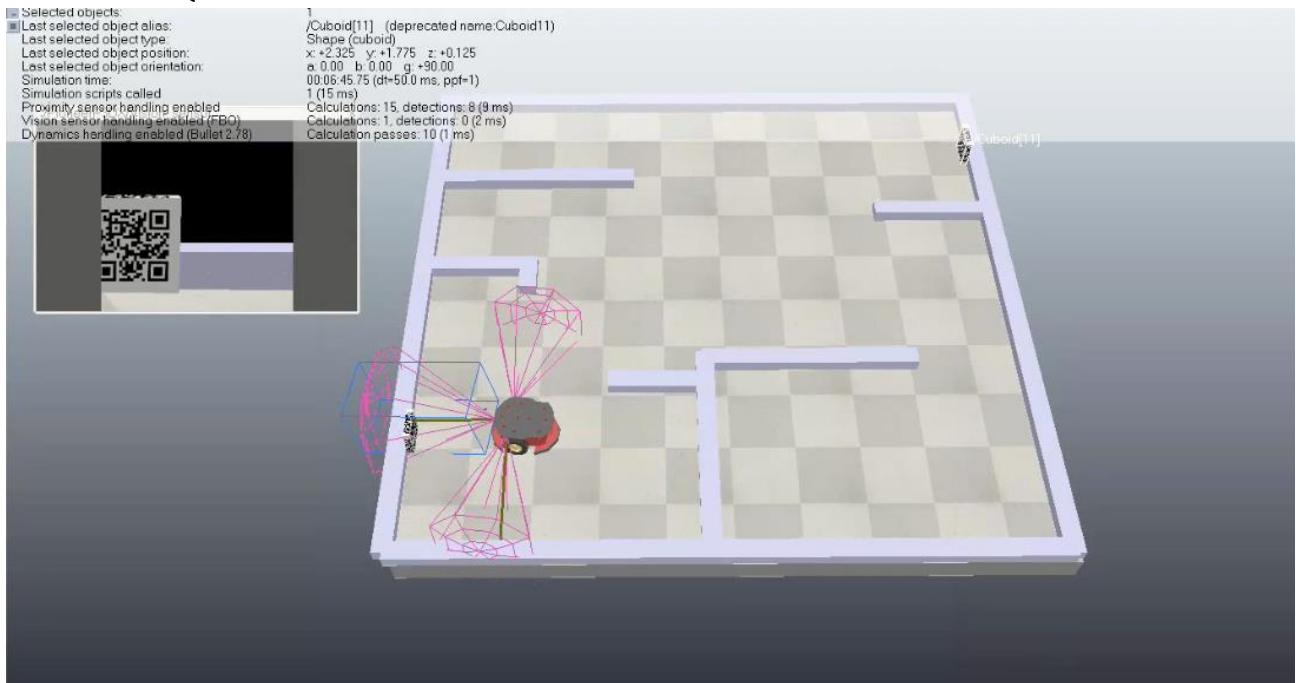
Una volta terminata la fase di test, si è riuscita ad ottenere una simulazione dell'attività del robot all'interno simulato con successo. Si riportano le fasi principali di essa:

- Fase iniziale:



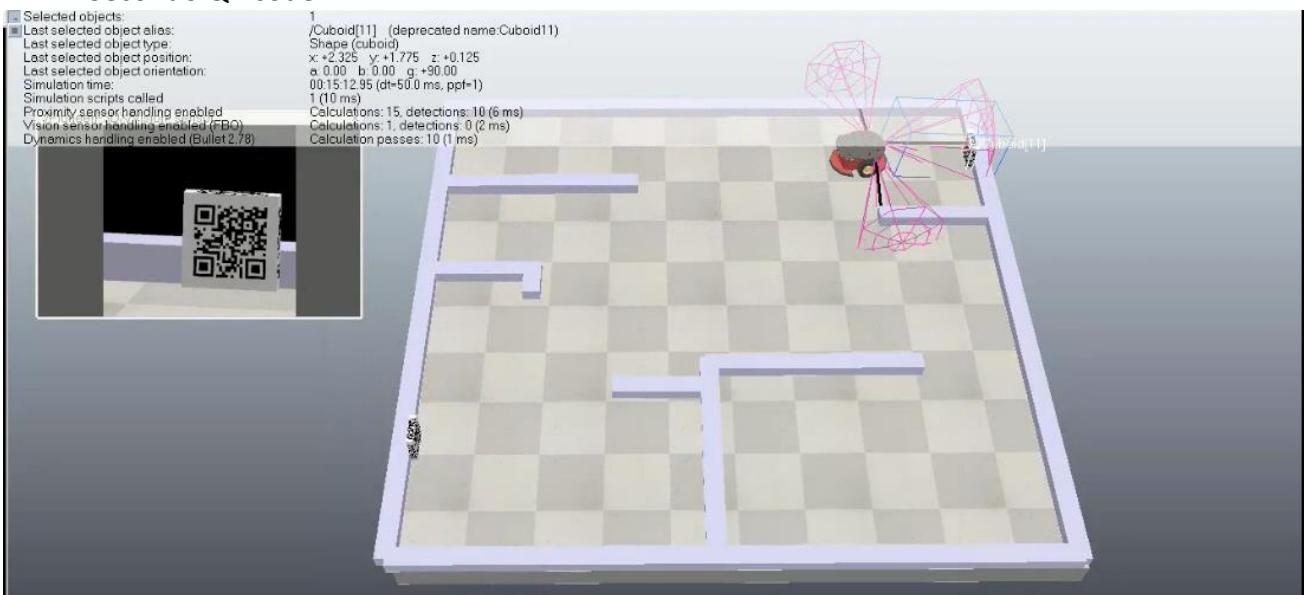
Il Pioneer parte da una posizione iniziale stabilita, e inizia ad esplorare per raggiungere la posizione del primo QR code che conosce già;

- Primo QR code:



L'agente raggiunge il primo QR code all'interno del labirinto, lo identifica ed avendo ora un nuovo obiettivo, ossia una nuova posizione da raggiungere, riparte verso di essa;

- Secondo QR code:



L'agente raggiunge il secondo QR code e ne decifra il contenuto attraverso la chiave che ha trovato all'interno del QR code precedente.

Al seguente link è possibile reperire la simulazione completa: [link al video](#).

Considerazioni finali

Con la simulazione si è raggiunto un risultato soddisfacente ma si è anche notato che negli istanti iniziali il robot tendeva ad oscillare velocemente da destra a sinistra, anche se questo non influiva sul corretto raggiungimento dell'obiettivo; si è di conseguenza pensato di implementare, nella successiva parte fisica, la seguente formula che migliora questo aspetto rendendo più fluido il movimento del robot stesso:

$$r = a*s + (1-a)*r$$

Dove:

s : è il valore di velocità impartito dal controllore;

r : è il valore di velocità dato effettivamente alle ruote;

a : fattore di ponderazione del nuovo valore rispetto al vecchio.

Parte Fisica

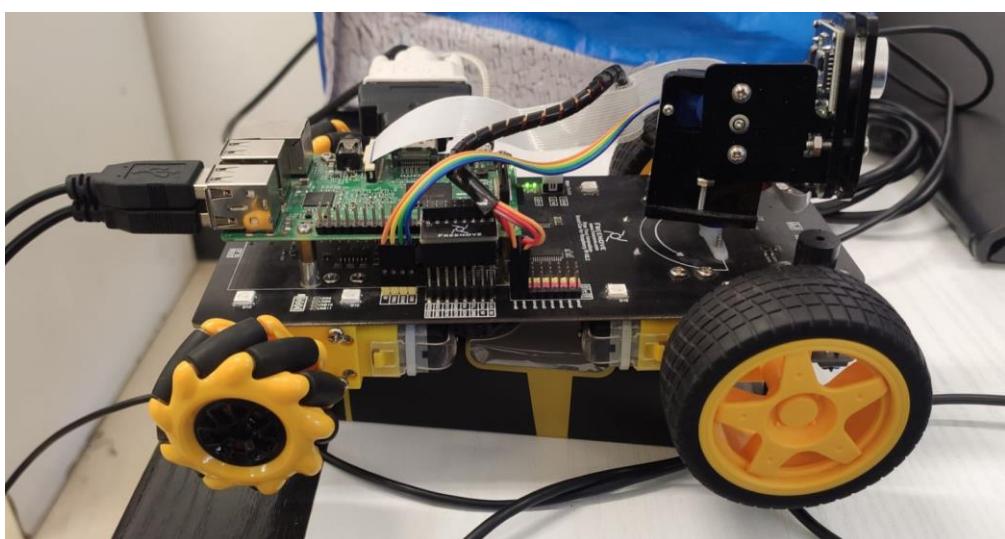
In questa sezione viene illustrata la parte fisica del progetto Rover-Maze: si riutilizzeranno le componenti logiche usate per la parte virtuale e saranno integrate alle componenti elettroniche più avanti illustrate.

Nello specifico elencheremo tutte le tecnologie già utilizzate nel virtuale ed in dettaglio quelle nuove, l'architettura logica e fisica adottata, le tarature effettuate e tutti i dati relativi alla simulazione vera e propria; infine, si troveranno delle considerazioni a conclusione della sezione.

Tecnologie

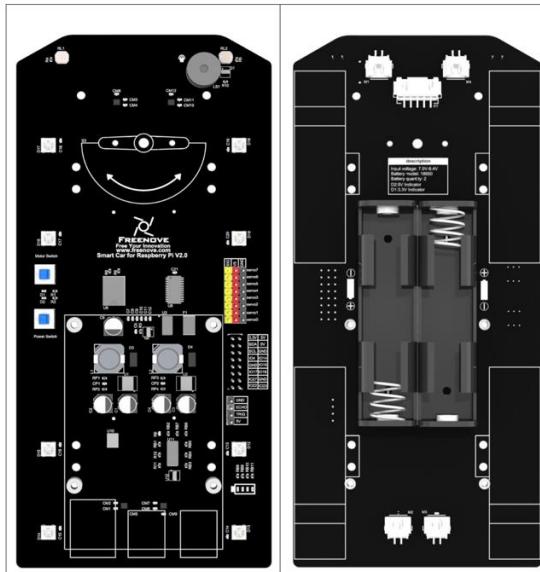
Nel progetto sono state utilizzate le seguenti tecnologie, alcune già usate nella parte virtuale (fare riferimento a questa sezione per maggiori informazioni) ed altre specifiche di questa sezione:

- Python e sue librerie;
 - **py_trees;**
 - **Io;**
 - **pillow;**
 - **numpy;**
 - **pyzbar;**
 - **time;**
 - **json;**
 - **base64;**
 - **serial:** libreria che permette di leggere da porta seriale per acquisire le letture provenienti da un Arduino Nano a cui sono collegati parte dei sensori;
- ROS 2;
- Behaviour Tree;
- 1x Freenove 4WD Smart Car Kit for Raspberry Pi:



Di seguito gli elementi che lo costituiscono:

4WD Smart Car Board per Raspberry Pi:



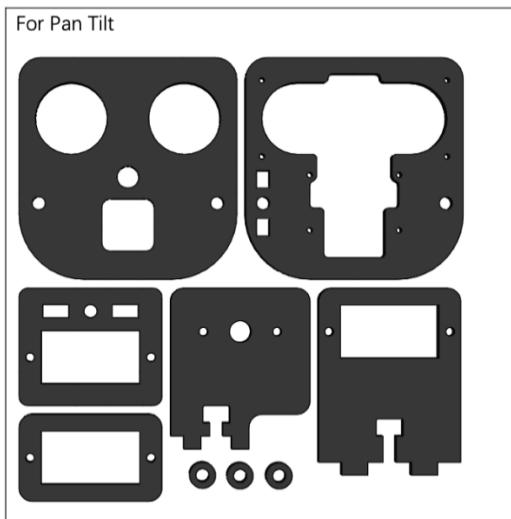
Parti della macchina:

M1.4*4 self-tapping Screw  x12 Freenove	M2.5*4 Screw  x5 Freenove	M3*6 Screw  x5 Freenove	M2.5*8+6 Standoff  x5 Freenove	M3*30 Standoff  x3 Freenove
M2*10 Screw  x5 Freenove	M3*14 Screw  x4 Freenove	M2 Nut  x5 Freenove	M3 Nut  x4 Freenove	

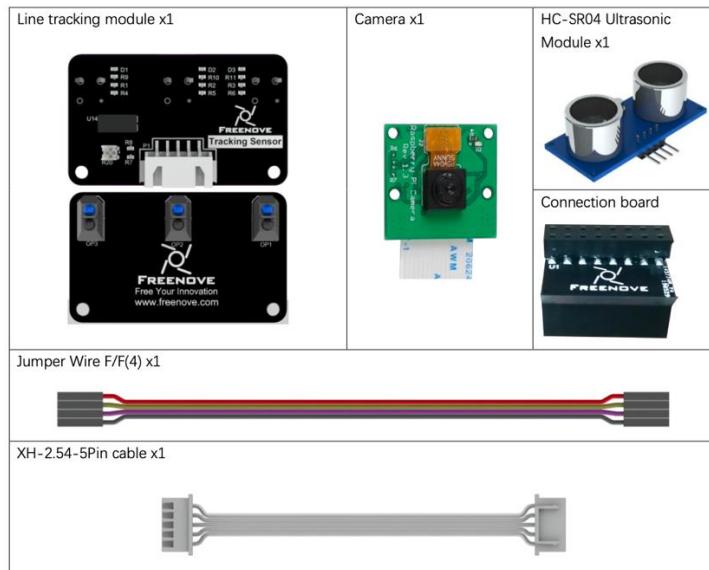
Parti di trasmissione:

Servo package x2 	Driven wheel x4 
DC speed reduction motor x4 	Motor bracket package x4 

Parti acriliche:



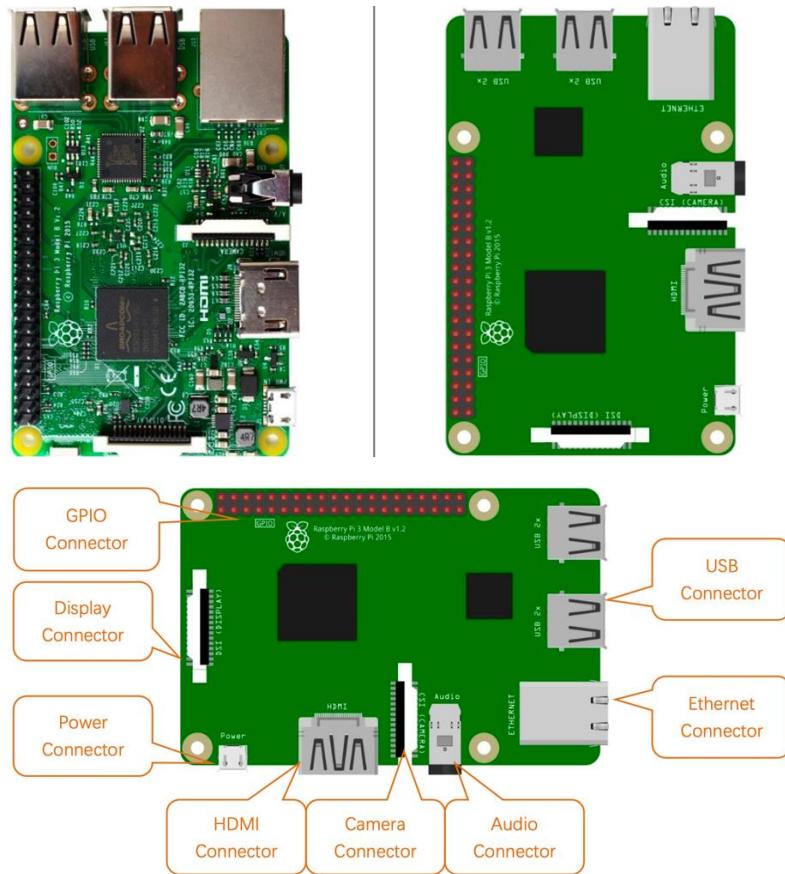
Parti elettroniche:



Parti già pronte:

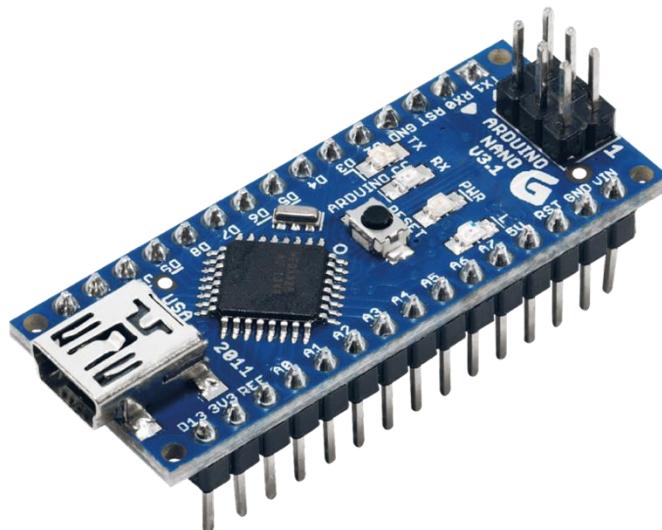


Raspberry Pi 3:



Gli elementi che sono stati integrati successivamente, per le esigenze che si sono poste sono:

- 1x Arduino nano:



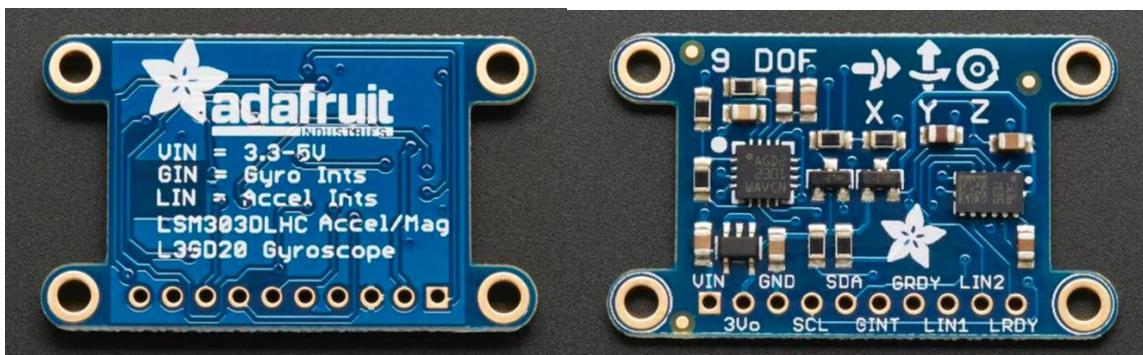
- 2x HC-SR04 Ultrasonic Module:



- 1x Adafruit 9-DOF IMU Breakout:

il chip presenta diversi sensori:

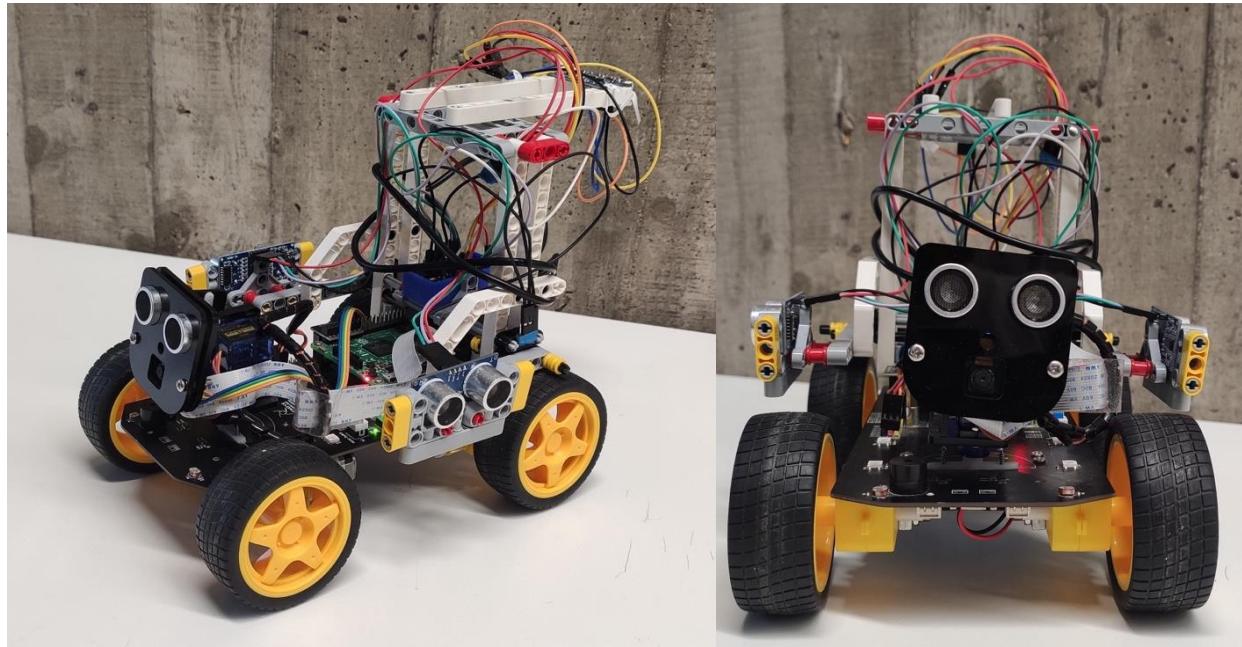
- un Giroscopio a 3 assi;
- un Magnetometro a 3 assi;
- un Accelerometro a 3 assi.



- Componenti lego provenienti dal kit Lego Mindstorms educational eve3:

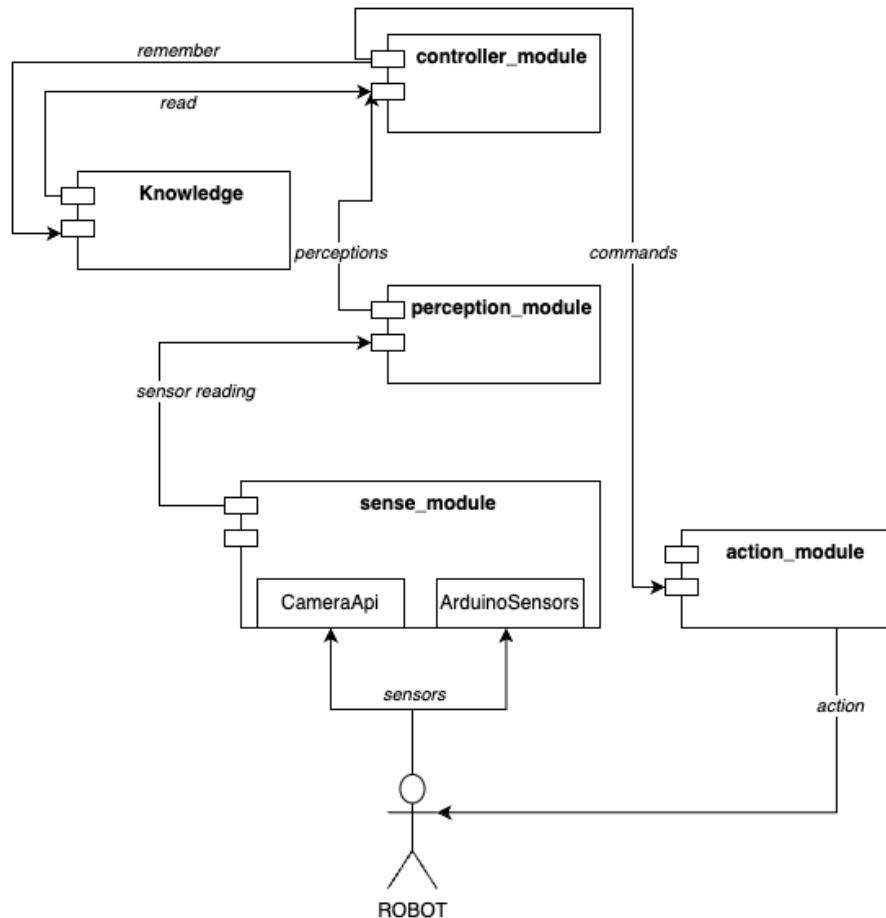


Il modello completamente assemblato risulta il seguente:



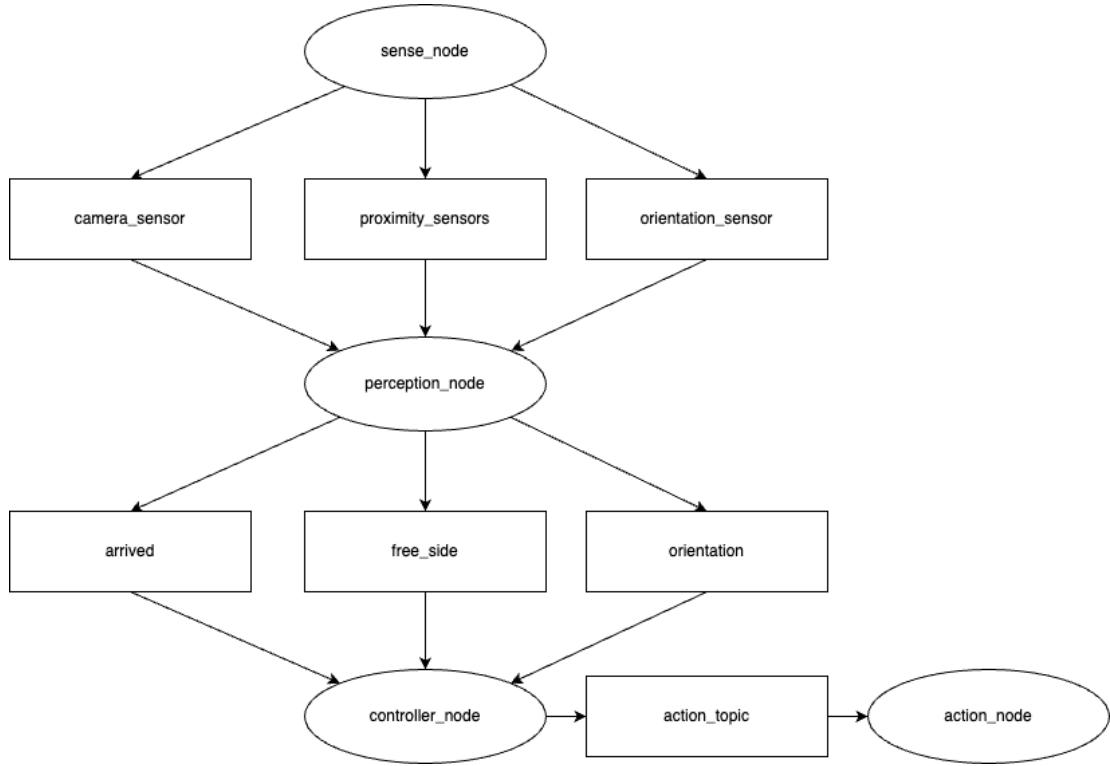
Architettura Logica

Questa è rimasta invariata rispetto a quella della parte virtuale, tranne per il fatto che nel sense_module sono stati aggiunti due sottomoduli, CameraApi e ArduinoSensors, per permettere la lettura dei dati provenienti dalla camera e dall'Arduino nano rispettivamente. In figura si può vedere lo schema:



Architettura fisica: Comunicazione

L'intercomunicazione tra i nodi ROS è rimasta invariata rispetto alla parte virtuale, come di nuovo riportato in figura sottostante:



Fisicamente, però, l'architettura risulta distribuita come segue:

- **Raspberry**: contiene il **sense_node** e l'**action_node**;
- **PC**: contiene **perception_node** e **controller_node**.

Questa configurazione distribuita è stata scelta per alleggerire il più possibile, da un punto di vista computazionale, il Raspberry Pi 3.

Invece per la comunicazione tra il **sense_node** e i sensori si è proceduto in questo modo:

- **API**: per la lettura dei dati dalla camera;
- **Seriale**: per la lettura dei sensori ultrasonici e dell'IMU a 9 assi.

Setup ROS2

Per utilizzare ROS2 si è proceduto creando due diversi container Docker:

- il primo utilizzato sul PC partendo dall'immagine con tag latest della repository ufficiale di ROS, reperibile al seguente link: https://hub.docker.com/_/ros/tags, configurandola come

riportato nei passi già riportati nell'omonima sezione virtuale, tranne per il passo 2 da sostituire col seguente:

```
docker run -it --privileged --network host --security-opt seccomp:unconfined --name myros  
ros
```

alternativamente si può scaricare un'immagine già configurata a questo link: https://hub.docker.com/repository/docker/matteop2001/logic_robotics_project/general per far partire il container, dopo aver scaricato l'immagine, sarà comunque necessario eseguire il comando:

```
docker run -it --privileged --network host --security-opt seccomp:unconfined --name myros  
matteop2001/logic_robotics_project:latest
```

- il secondo utilizzato sul Raspberry partendo dall'immagine con tag latest della repository ufficiale di ROS, reperibile al seguente link: <https://hub.docker.com/r/arm64v8/ros/tags>, configurandola come riportato nei passi già riportati nell'omonima sezione virtuale, tranne per il passo 2 da sostituire col seguente:

```
docker run -it --device /dev/gpiomem --device /dev/i2c-0 --device /dev/i2c-1 --device  
/dev/ttyUSB0 --privileged --network host --security-opt seccomp:unconfined --name myros2  
arm64v8/ros
```

alternativamente si può scaricare un'immagine già configurata a questo link: https://hub.docker.com/r/matteop2001/robotics_project/tags; per far partire il container, dopo aver scaricato l'immagine, sarà comunque necessario eseguire il comando:

```
docker run -it --device /dev/gpiomem --device /dev/i2c-0 --device /dev/i2c-1 --device  
/dev/ttyUSB0 --privileged --network host --security-opt seccomp:unconfined --name myros2  
matteop2001/robotics_project
```

L'utilizzo di network host risulta necessario per l'intercomunicazione tra due container ROS eseguiti su macchine diverse.

Per concludere, affinché tutti i nodi possano comunicare effettivamente tra loro, su entrambi i container bisogna modificare il file `~/.bashrc` aggiungendo il seguente comando:

```
ROS_DOMAIN_ID=NUM
```

dove NUM deve essere sostituito da un numero qualsiasi compreso tra 0 e 232 ma uguale per entrambi i container (preferibilmente più basso possibile); infine riavviare il terminale o eseguire `source .bashrc` per apportare le modifiche. Per maggiori dettagli si può consultare la documentazione ufficiale al seguente [link](#).

Discretizzazione

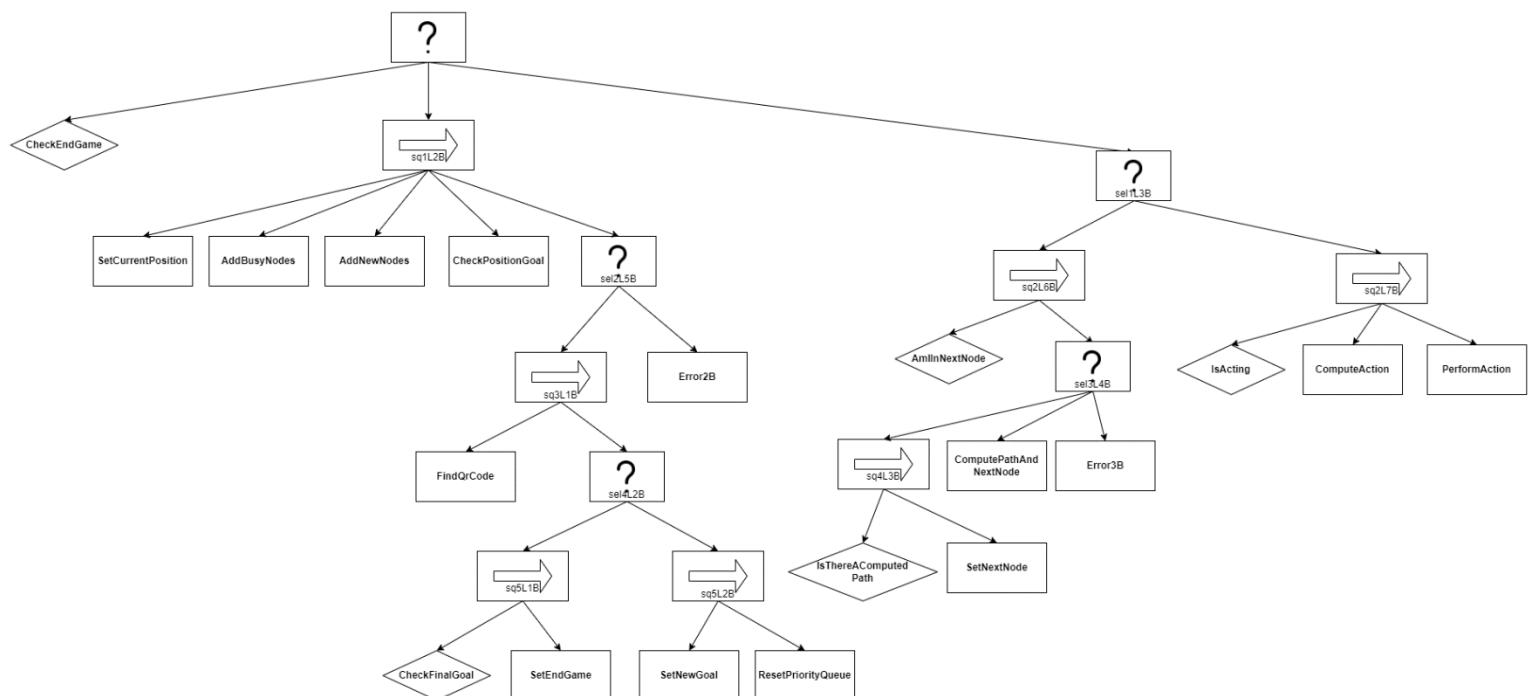
Nel virtuale, disponendo di un “hardware” più potente, il nodo sense riusciva a pubblicare dati nell’ordine di millisecondi e quindi i dati erano aggiornati pressoché in tempo reale; nel fisico, invece, sense pubblica circa ogni 2s e quindi è stato necessario ricorrere ad una discretizzazione. Quanto detto ha portato a dover effettuare una modifica ai nodi di controller, ossia aggiungere un tempo di azione che deve poi essere condiviso anche al nodo action, il quale ha responsabilità di fermare il robot alla fine di ogni azione.

```

action action_time
[INFO] [1708855717.461191055] [action_node]: {"action": "go_forward", "time": 0.8620689655172413}
action action_time
[INFO] [1708855719.756046801] [action_node]: {"action": "stop", "time": 0.8620689655172413}
action action_time
[INFO] [1708855721.576261445] [action_node]: {"action": "go_forward", "time": 0.8620689655172413}
action action_time
[INFO] [1708855723.764766040] [action_node]: {"action": "stop", "time": 0.8620689655172413}
action action_time
[INFO] [1708855726.923238412] [action_node]: {"action": "go_forward", "time": 0.8620689655172413}
action action_time
[INFO] [1708855729.224024195] [action_node]: {"action": "stop", "time": 0.8620689655172413}
action action_time
[INFO] [1708855730.978435352] [action_node]: {"action": "turn_right", "time": 0.5128514555836653}
action action_time
[INFO] [1708855733.174639251] [action_node]: {"action": "turn_right", "time": 0.512280377479828}
action action_time
[INFO] [1708855735.530827089] [action_node]: {"action": "turn_right", "time": 0.3490943862993355}
action action_time
[INFO] [1708855737.809320009] [action_node]: {"action": "go_forward", "time": 0.8620689655172413}
action action_time
[INFO] [1708855740.308288807] [action_node]: {"action": "stop", "time": 0.8620689655172413}
action action_time
[INFO] [1708855741.540615279] [action_node]: {"action": "turn_left", "time": 0.510436632397042}
action action_time
[INFO] [1708855744.970321636] [action_node]: {"action": "turn_left", "time": 0.42019122529254976}
action action_time
[INFO] [1708855747.357273410] [action_node]: {"action": "turn_left", "time": 0.4169859812282187}
action action_time
[INFO] [1708855749.587387645] [action_node]: {"action": "turn_left", "time": 0.3278229012511832}

```

Behaviour Tree



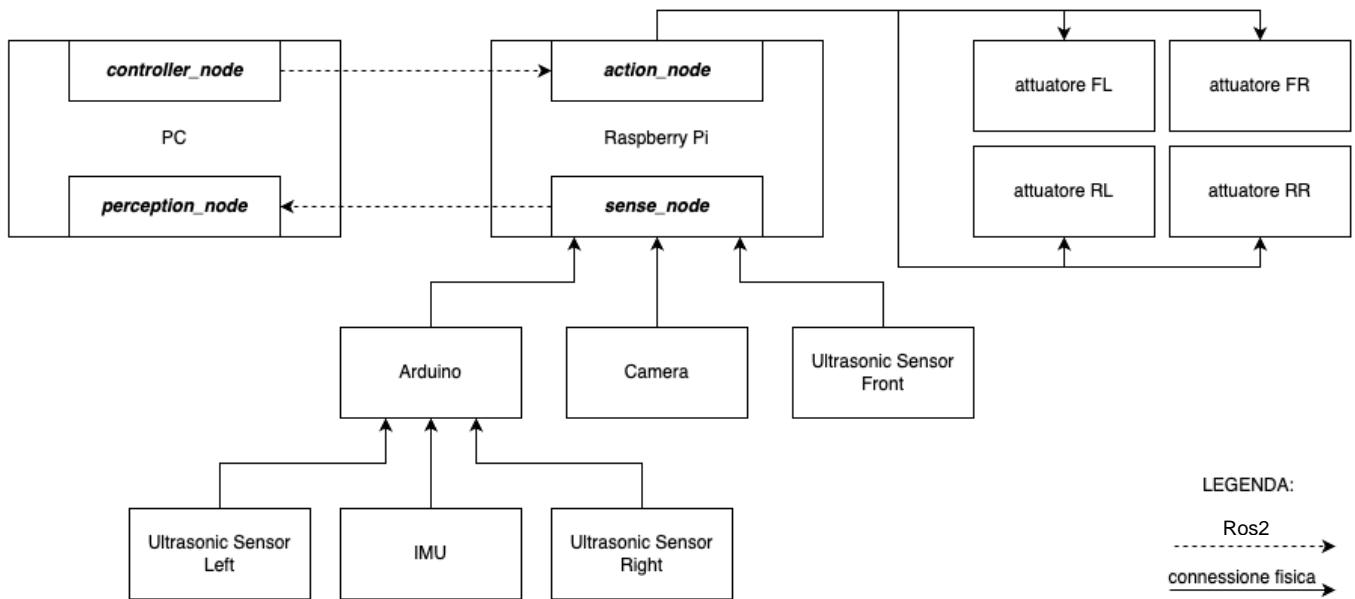
Rispetto al Behaviour Tree costruito per la simulazione effettuata su CoppeliaSim, è stato aggiunto un nodo in più:

IsActing:

- **input:** end action time (il tempo di durata dell'azione).

Che restituisce Success se l'azione è finita (end action time =0).

Architettura fisica: Hardware



In figura è evidenziata la gerarchia delle connessioni hardware e come le singole componenti sono collegate tra loro, tramite connessione fisica cablata o tramite ssh.

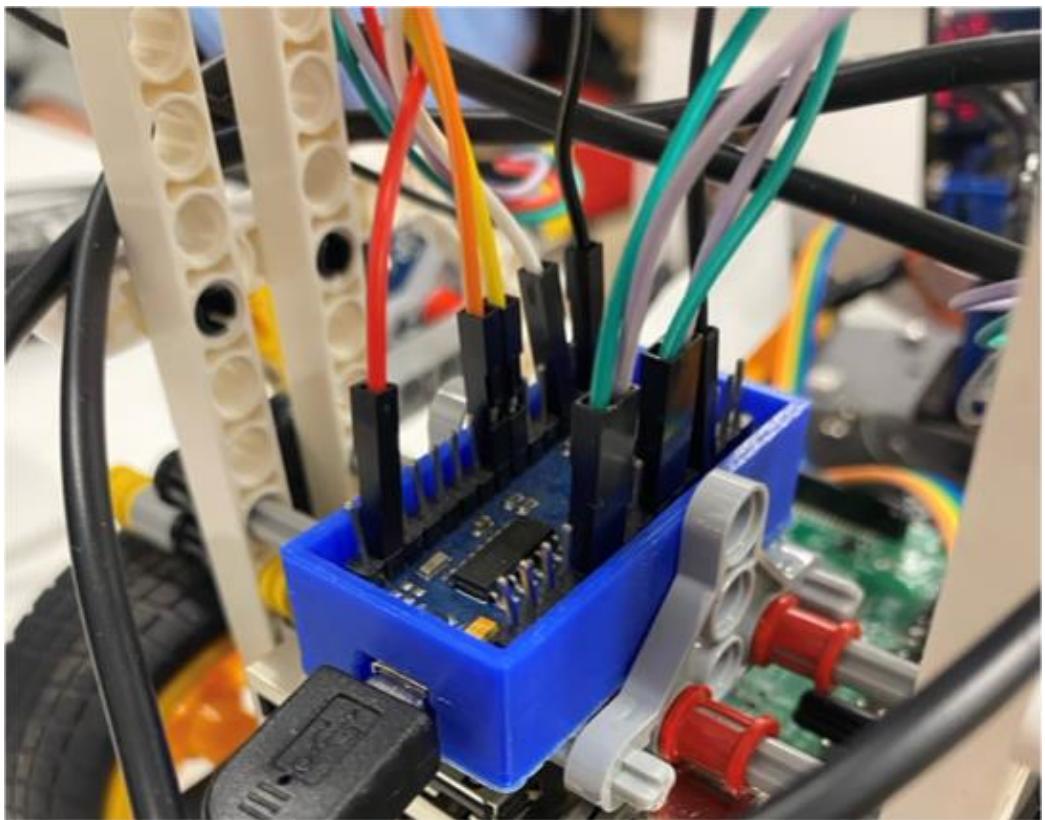
Arduino

Come visibile già dallo schema precedente, ad Arduino sono collegati i sensori sottoelencati tramite i seguenti collegamenti:

IMU 9 axis	Arduino
GND	GND
VIN	V3
SCL	A5 [SCL]
SDA	A4 [SDA]

Ultrasonic left	Arduino
GND	GND
VIN	V5
Trig	3
Echo	4

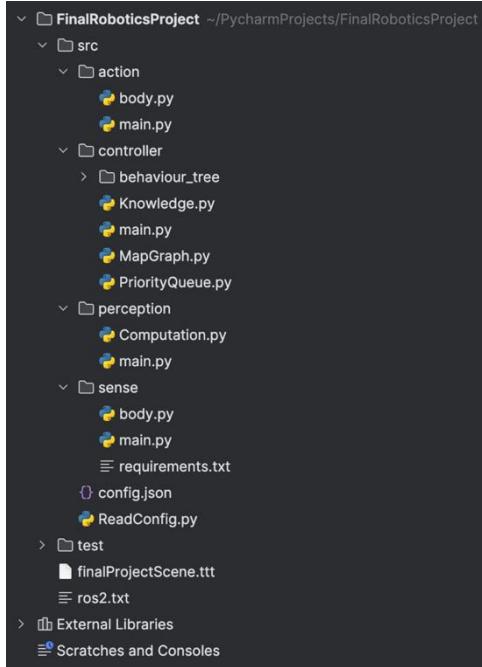
Ultrasonic right	Arduino
GND	GND
VIN	V5
Trig	7
Echo	8



Collegamenti pin di Arduino

Struttura del codice

La struttura del codice è rimasta pressoché invariata rispetto al virtuale.



Da notare che ora le classi body.py non rappresentano più il corpo simulato del robot su CoppeliaSim, come accadeva in precedenza, ma bensì il corpo del robot **Freenove 4WD** per interagire con gli attuatori.

Altre differenze sono nel fatto che nel virtuale avevamo un solo file **config.json**, ora invece ne abbiamo due; infatti, prima avevamo lo stesso file per tutti e 4 i container che si trovavano sulla stessa macchina, a differenza di ora, che abbiamo su Raspberry sense ed action, con questo file di configurazione:

```
{  
    "TURN_SPEED": 1500,  
    "SPEED": [740, 750, 690, 700],  
    "ABS_SPEED": [340, 350, 230, 240]  
}
```

Mentre, i nodi di perception e controller si trovano su un'altra macchina ed hanno bisogno di un ulteriore file di configurazione:

```
{  
    "PRIORITY_WEIGHT": 1,  
    "SPEED": 29,  
    "SPACE": 25,  
    "FREE_SIDE_THRESHOLD": 40,  
    "STRING_TARGET_ARRIVE": "finish",  
    "NODE_DISTANCE_THRESHOLD": 18,  
    "SLOPE": 0.477,  
    "OFFSET": 0.125,  
    "AM_I_IN_NEXT_NODE_THRESHOLD": 15,  
    "START_GOAL": [110, -90]  
}
```

Codice Arduino

Per l'utilizzo dei sensori collegati in seriale ad Arduino è stato necessario il caricamento su quest'ultimo di uno script con lo scopo di leggere i due senso di prossimità, destro e sinistro, e i dati in ingresso dalla IMU. Il file è stato chiamato **sensor_and_imu_reading imu**, riportiamo di seguito le due funzioni principali:

- Funzione per la lettura dei sensori di prossimità:

```
// Funzione per il sensore ultrasonico
long sonarsensor(int triggerPin, int echoPin) {
    long distance = 0;
    digitalWrite(triggerPin, LOW);
    digitalWrite(triggerPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(triggerPin, LOW);
    long times = pulseIn(echoPin, HIGH);
    if (times < 38000) {
        distance = 0.034 * times / 2;
        return distance;
    }
    return distance;
}
```

- Funzione per la lettura della IMU:

```
float imu_orientation_reading() {
    sensors_event_t accel_event;
    sensors_event_t mag_event;
    sensors_vec_t orientation;

    /* Calculate pitch and roll from the raw accelerometer data */
    accel.getEvent(&accel_event);

    /* Calculate the heading using the magnetometer */
    mag.getEvent(&mag_event);
    if (dof.magGetOrientation(SENSOR_AXIS_Z, &mag_event, &orientation)) {
        /* 'orientation' should have valid .heading data now */
        float angle = orientation.heading - initialAngle;
        if (angle < 0) {
            angle += 360; // Aggiungi 360 gradi per ottenere un valore positivo
        }
        return angle;
    }
}
```

Taratura

Movimento rettilineo

Il passaggio dalla simulazione virtuale a quella fisica del progetto ha necessariamente richiesto una fase di taratura dei movimenti del robot.

In particolare, è stato necessario effettuare diverse misure: si è iniziato andando a cronometrare il tempo che il robot impiegava a percorrere un metro, iniziando con valori di velocità per i 2 motori destri di 1500 e dei 2 motori sinistri di 1270 (scelte diverse tra loro per correggere una traiettoria tendente leggermente verso sinistra).

Dopo 5 prove sono stati ottenuti i seguenti risultati:

1. 2,64 s
2. 2,57 s
3. 2,78 s
4. 2,57 s
5. 2,73 s

Con una media di 2,66 s.

Si è deciso allora di settare il valore medio appena attenuto come tempo per effettuare la controprova, ossia verificare che lo spazio percorso fosse effettivamente di 1 metro, ottenendo il seguente risultato:

1. 2,66 s → 96 cm

Poiché lo spazio percorso risultava inferiore a 1m, si è deciso di procedere incrementando il tempo progressivamente, ottenendo i seguenti risultati su 5 prove:

1. 2,70 s → 98 cm
2. 2,74 s → 100 cm
3. 2,74 s → 98 cm
4. 2,74 s → 99 cm
5. 2,75 s → 99 cm

Arrivando al settaggio definitivo di 2,75 s che corrisponde ad una velocità di 36 cm/s; nel codice è stato settato il valore k, rapporto tra le velocità dei motori sinistri rispetto a quelli destri, a 0.84666. In un secondo momento, è stato però necessario diminuire la velocità del robot, procedendo in modo simile a quanto illustrato in precedenza, arrivando ad una velocità per i motori destri di 900 e per i motori sinistri di 765 (scelte diverse tra loro per il motivo illustrato in precedenza), ossia una velocità pari a 22 cm/s con k=0.85.

Successivamente si è svolto anche un test per calcolare lo spazio percorso dal robot in 1 secondo, poiché spesso veniva impartito il comando di avanzare per quel determinato tempo, ottenendo questi risultati:

- 31 cm;
- 29.5 cm;
- 30.5 cm;
- 30 cm;
- 29 cm.

Da cui si è scelto come valore la media, ossia 30 cm, valore fondamentale per calcolare di quanto il robot si sposti nel labirinto.

Rotazione

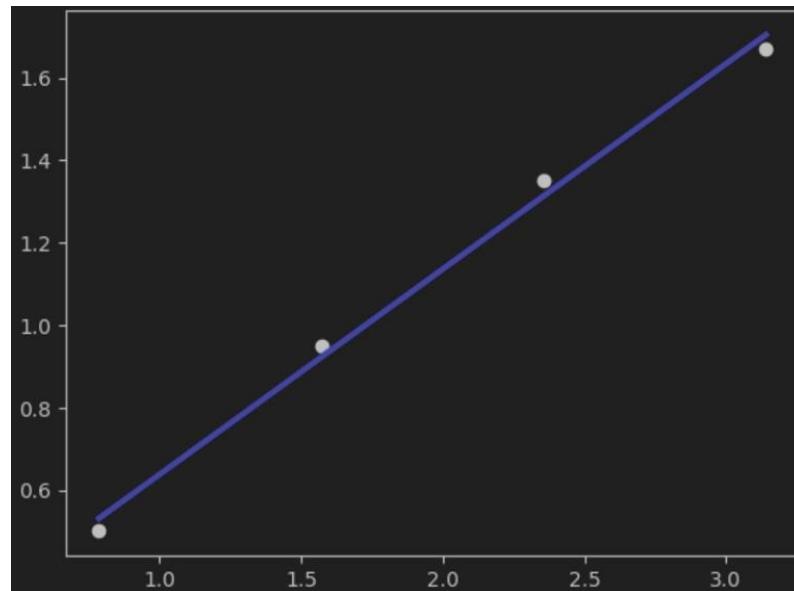
Un ulteriore test importante da eseguire è stato quello di rotazione, visto che il robot avrebbe dovuto compiere spesso anche questo movimento. Le prove hanno evidenziato come, ad influire sulle prestazioni, fosse anche il tipo di pavimento a causa del diverso coefficiente di attrito, come visibile dai dati seguenti:

Angolo di rotazione	Pavimento aula C 1.9 (motori a 1000)	Pavimento Lab (motori a 1500)	Pavimento aula A 1.7 (motori a 1500)
360°	2.3 s	3.3 s	3.35 s
180°	1.3 s	1.69 s	1.67 s
90°	0.78 s	0.9 s	0.95 s
45°	0.45 s	0.5 s	0.5 s

Dopo questi test si è scelto di considerare solo angoli inferiori a 180°, perché queste sarebbero state le rotazioni effettivamente compiute in fase di simulazione:

Angolo di rotazione	Pavimento Lab (motori a 1500)
180°	1.67 s
135°	1.35 s
90°	0.95 s
45°	0.5 s

Questa scelta è stata presa anche perché i dati sopra riportati sono stati dati in input ad un modello che calcolasse una retta di regressione lineare per poter stabilire rotazioni di un'angolazione qualsiasi, e che quindi avrebbe dato risultati più imprecisi proprio per le angolazioni piccole di rotazione, che sono di maggior interesse nella simulazione.



Esempio retta di regressione lineare

Conclusione

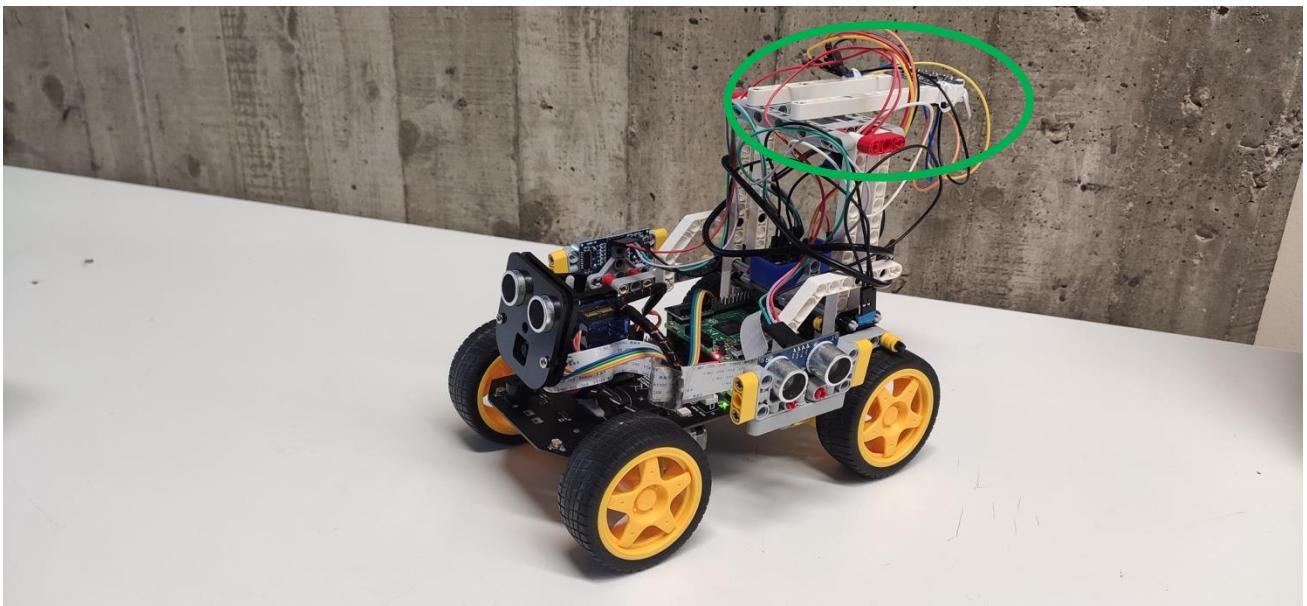
Nonostante le tarature effettuate nei due paragrafi precedenti, si è notato che lo spazio e gli angoli di rotazione non rimangono costanti anche a parità di pavimentazione, suggerendo una dipendenza anche da altri fattori, come ad esempio la percentuale di carica delle batterie di alimentazione. Visto che non si è potuto però accertare la reale causa, si è deciso che ad ogni nuovo inizio di simulazione sia necessario effettuare nuovamente le 5 prove di movimento in avanti (spazio percorso in 1 secondo) ed i 4 test di rotazione (45° , 90° , 135° , 180°) per settare correttamente tutti i parametri della stessa.

IMU 9 assi

Per la taratura della IMU si è cercato di appurare se alle rotazioni fisiche corrispondessero effettivamente letture corrette, di seguito i dati ottenuti:

Angolo di rotazione	Angolo misurato
360°	360°
180°	150°
90°	80°
45°	35°

Visto quanto sopra, si evince che il dispositivo rileva abbastanza correttamente le rotazioni e per questo non sono state necessarie ulteriori correzioni o ritarature ad inizio simulazione. In via precauzionale, però, per evitare interferenze magnetiche dovute ai motori e agli altri dispositivi, è stata alloggiata su di una struttura rialzata e arretrata rispetto al baricentro, realizzata con pezzi del Lego Mindstorm già citato in precedenza.



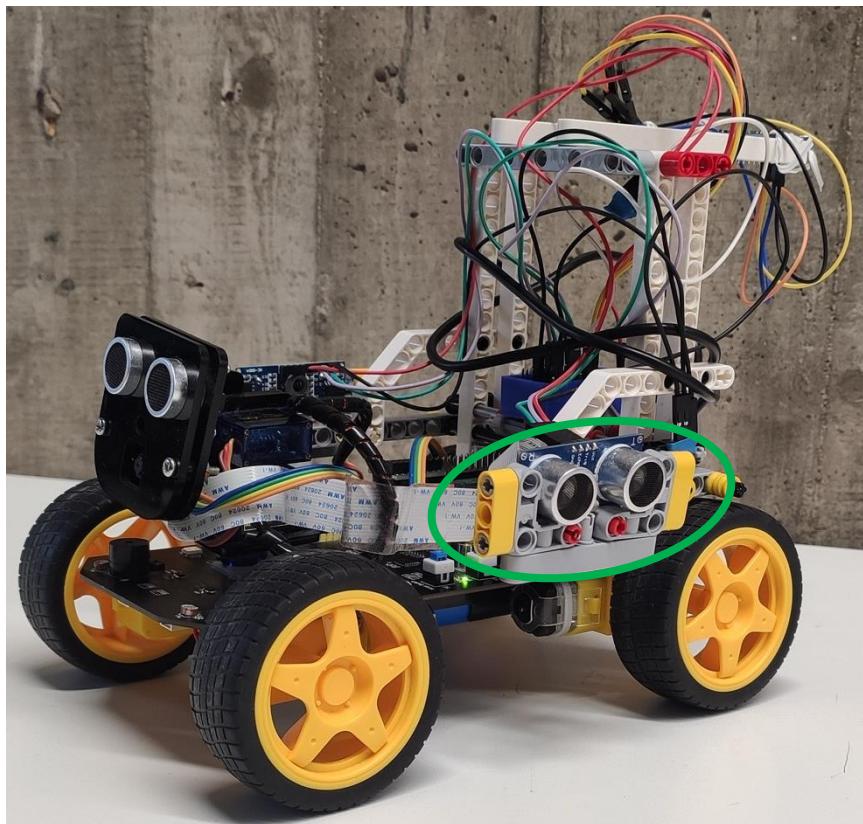
Posizione IMU a 9 assi

Sensori ultrasonici

I sensori ultrasonici sono stati tarati posizionando gli stessi a distanze differenti, di interesse progettuale, da un ostacolo e rilevando le distanze misurate con i seguenti risultati:

Distanza ostacolo	distanza misurata
10 cm	9 cm
20 cm	18 cm
30 cm	27 cm
40 cm	35 cm
50 cm	45 cm

Viste le distanze di movimento rettilineo e le dimensioni del labirinto, i precedenti risultati si ritengono accettabili ai fini progettuali. Fisicamente, i sensori sinistro e destro sono stati posizionati in un alloggiamento realizzato con pezzi Lego Mindstorm ed ancorati al ponteggio creato per sorreggere la bussola, visto la mancanza di altri punti di ancoraggio sulla board del robot; quello frontale invece è già integrato nel kit iniziale del robot.



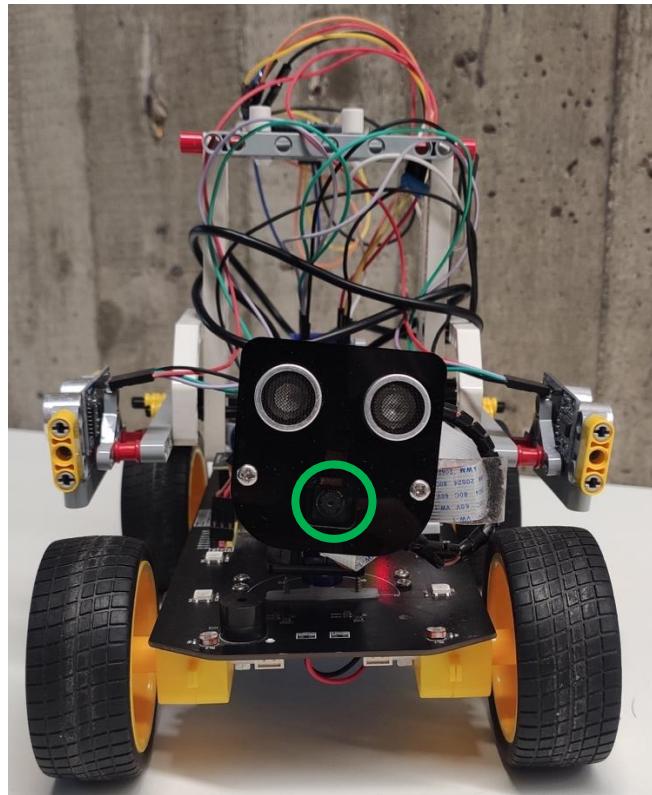
Posizione sensori ultrasonici

Camera

Questo dispositivo è stato tarato effettuando alcune foto di prova, per capire la risoluzione reale delle immagini, e successivamente test pratici di rilevazione di QR code, con risultati più che soddisfacenti anche in termini di distanza di rilevamento.



Foto scattata dalla camera



Posizione Camera

GUI

Si è deciso di creare una GUI al fine di facilitare l'avvio e lo stop della simulazione. Essa è stata creata in Python con l'utilizzo del modulo Tkinter. In particolare, sono stati inseriti due pulsanti, "Start" e "Stop", e premendo su uno di essi viene visualizzato un messagebox bloccante che fornisce un feedback all'utente sullo stato dell'operazione.

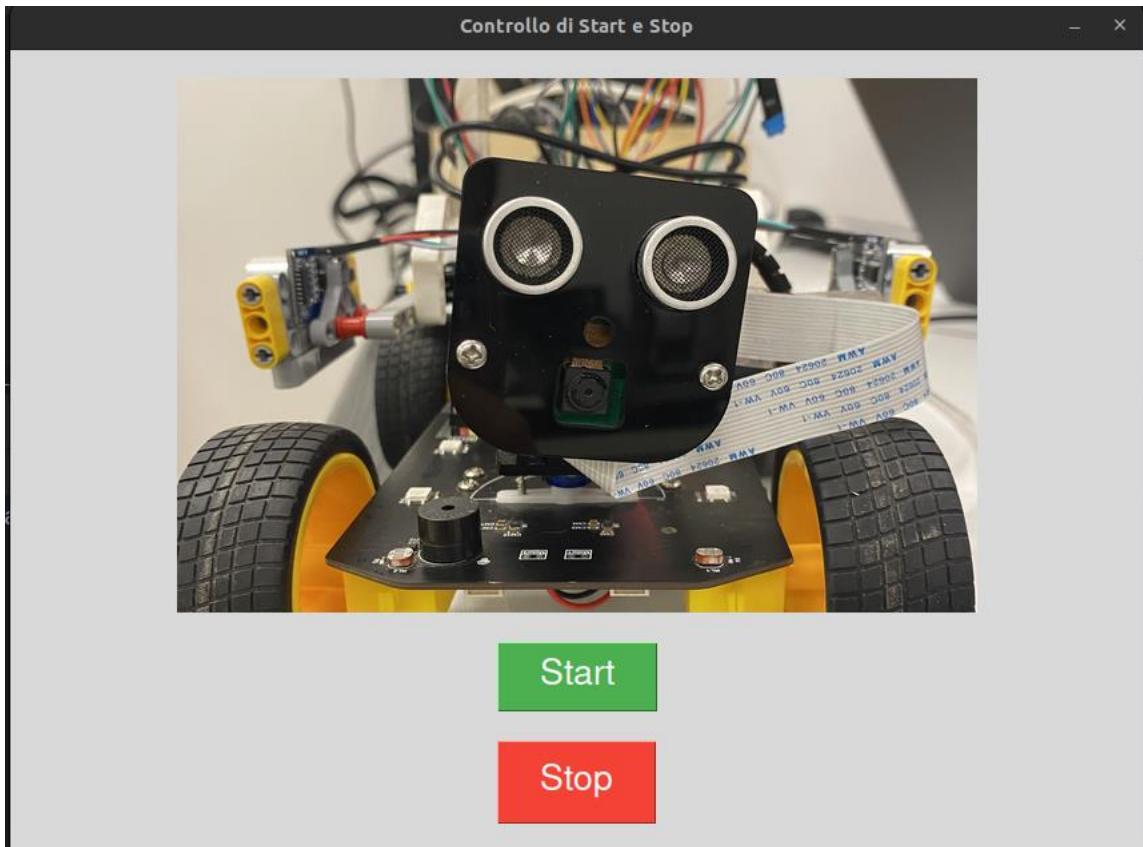


Foto della GUI

Quando il tasto di Start viene premuto viene avviato il container contenente la logica sul personal computer, e viene fatta una richiesta all'api sul Raspberry Pi che serve per avviare il container lì presente.

```
1 usage  ▾ Matteopast01 *
def start(self):
    self.start_button.config(state=tk.DISABLED)
    self.stop_button.config(state=tk.NORMAL)
    messagebox.showinfo(title: "Message", message: "Simulation will start shortly!")
    subprocess.Popen(["bash", "robot_start.sh"])
    config = ReadConfig()
    ip = config.read_data("IP_RASPBERRY")
    r = requests.get(f'http://{{ip}}:5008/start')
```

Al tasto di Stop vengono fermati i motori ed entrambi i container.

```
↳ Matteopast01
def stop(self):
    self.start_button.config(state=tk.NORMAL)
    self.stop_button.config(state=tk.NORMAL)
    messagebox.showinfo( title: "Message", message: "Simulation has been stopped!")
    subprocess.Popen(["bash", "robot_stop.sh"])
    config = ReadConfig()
    ip = config.read_data("IP_RASPBERRY")
    r = requests.get(f'http://{ip}:5008/stop')
```

Simulazione Fisica

Settaggio ambiente di simulazione

L'ambiente simulativo, costituito dal labirinto, è stato realizzato nella pratica con scatole e cartoni di vario tipo, come in foto:



I punti hanno coordinate, relativamente al robot:

- X = 150 cm, Y = 0;
- 1° QR: X = 150 cm, Y = 120 cm;
- 2° QR: X = 150 cm, Y = 250 cm.

Il robot, a partire dalla sua posizione iniziale, dovrà raggiungere dapprima la posizione del 1°QR, dove ruotando su se stesso inizierà a cercarlo e successivamente decodificarlo; una volta fatto, conoscerà le coordinate del 2°QR e quindi ripeterà nuovamente la stessa procedura con la differenza che stavolta per decodificarlo avrà bisogno della chiave contenuta nel 1°QR Code. Alla decodifica di ogni QR il robot emetterà un suono di conferma decodifica.

Taratura

come accennato anche nella sezione omonima, si effettuano le tarature di movimento rettilineo e rotazione per determinare i corretti parametri di simulazione.

Avvio senza GUI:

- Avviare l'API necessaria per ottenere i dati della telecamera con il comando:

```
pi@pi:~ $ cd dapp-isrlab-project/
pi@pi:~/dapp-isrlab-project $ python3 api.py
 * Serving Flask app docker start myros2 -i
 * Environment: produ
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://0.0.0.0:5008/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 264-375-449
```

- Nella macchina preposta, avviare il container con i moduli perception e controller con i comandi:

- Lanciare i due nodi ROS controller e perception con i comandi:

```
root@dellone:/# ros2 run isrlab_project perception_module
[INFO] [1708872023.424650973] [perception_node]: Hello from perception_module
```

```
root@dellone:/# ros2 run isrlab_project controller_module
[INFO] [1708872021.997645584] [controller_node]: Hello from controller_module
```

- Su Raspberry avviare il container dove sono contenuti sense e action, in particolare avviare prima sense e poi action

```
root@pi:/# ros2 run my_robot sense_module
init robot
ultasonic init
serial init
api init
Connecting to simulator...
SIM objects referenced
[INFO] [1708872089.469461421] [sense_node]: Hello from sense_module
[INFO] [1708872090.592878139] [sense_node]: Publish Camera sensor
[INFO] [1708872091.870966152] [sense_node]: Publish Proximity sensor: {"left": 260.0, "center": 4, "right": 14.0}
[INFO] [1708872091.879255612] [sense_node]: Publish Orientation sensor: "0.004886921905583819"
[INFO] [1708872092.078199113] [sense_node]: Publish Camera sensor
```

```
root@pi:/# ros2 run my_robot action_module
[INFO] [1708872082.046326462] [action_node]: Hello from action_module
[INFO] [1708872094.828714818] [action_node]: {"action": "stop", "time": 0}
action action_time
[INFO] [1708872096.302680226] [action_node]: {"action": "stop", "time": 0}
action action_time
```

Simulazione step by step

Una volta terminata la fase di test, si è riuscita ad ottenere una simulazione dell'attività del robot con successo. Si riportano le fasi principali di essa:

- Fase iniziale:



Il robot parte da una posizione iniziale stabilita, e inizia ad esplorare per raggiungere la posizione del primo QR code che conosce già;

- Primo QR code:



L'agent raggiunge il primo QR code all'interno del labirinto, lo identifica ed avendo ora un nuovo obiettivo, ossia una nuova posizione da raggiungere, riparte verso di essa;

- Secondo QR code:



L'agent raggiunge il secondo QR code e ne decifra il contenuto attraverso la chiave che ha trovato all'interno del QR code precedente.

Al seguente link è possibile reperire la simulazione completa: [link al video](#) .

Conclusioni finali

Nonostante i risultati ottenuti in simulazione siano stati soddisfacenti, ci sono diversi aspetti del lavoro che potrebbero essere migliorati per ottenere prestazioni migliori.

In particolare, i sensori a volte non sono precisi nei valori che restituiscono, ad esempio il sensore di prossimità centrale segnala a volte distanze non veritieri, in eccesso o in difetto, e anche la bussola spesso indica angoli scarsi con differenze che arrivano ai 45°.

Non avendo potuto stabilire le cause di tali comportamenti, si potrebbero effettuare approfondimenti a tal proposito al fine di riuscire a risolverli o compensarli adeguatamente.