



POLITECNICO DI BARI

**DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE (DEI)
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA DELL'AUTOMAZIONE -
ROBOTICS**

**Progetto in
Robotics – Mobile and field Robotics**

TRACTOR 3D SLAM

Prof.:

Luca DE CICCIO

Tutor:

Carlo CROCE

Studenti:

Vincenzo BONASIA 577190

Gianluca CANNITO 577138

Michela FALCONE 577423

Anno Accademico 2019-2020

Sommario

1. COSTRUZIONE DEL TRATTORE.....	3
1.1 Presentazione del problema.....	3
1.2 Modello del trattore	3
2. RVIZ E GAZEBO	9
2.1 Custom world	11
3. CONTROLLO	13
4. SENSORI.....	18
5. ELABORAZIONE DATI SENSORE LASER	25
6. COSTRUZIONE DELLA MAPPA.....	30
6.1 Mappa in 2D.....	30
6.2 Mappa in 3D.....	36
7. CONCLUSIONI E LAVORO FUTURO	41
7.1 Sviluppi futuri.....	41
8. RIFERIMENTI.....	43

1. COSTRUZIONE DEL TRATTORE

1.1 Presentazione del problema

L'obiettivo di questo progetto è quello di costruire e controllare un trattore automatico capace, attraverso una depth camera e un LIDAR, di studiare l'ambiente circostante e gli ostacoli in esso contenuti, realizzando così uno SLAM (Simultaneous Localization and Mapping).

A tal fine, è stata creata inizialmente una mappa in 2D, usando esclusivamente il laser sensor Hokuyo, seguita dalla mappa in 3D, che è stata realizzata utilizzando il pacchetto *Octomap* attraverso la depth camera.

Con un altro passaggio eseguito all'interno del progetto si è effettuata la lettura dei dati estrapolati dal sensore laser. Attraverso i codici in Python, è stato possibile visualizzare su terminale la distanza frontale e laterale del trattore rispetto agli ostacoli e, di conseguenza, è stata imposta una distanza minima di avvicinamento agli ostacoli, oltre la quale il trattore si ferma.

1.2 Modello del trattore

Per il progetto è stato usato il modello innok heros tractor con 4 ruote

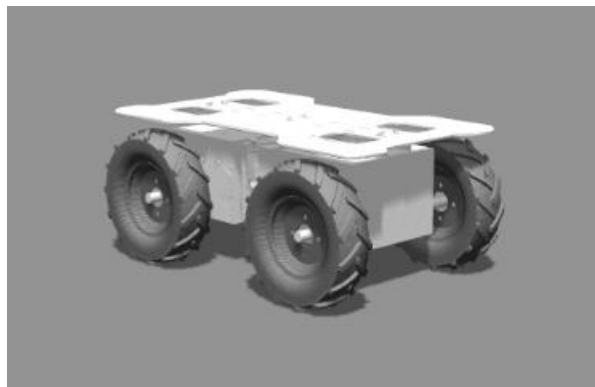


Figura 1: Trattore

Esso è dotato di due ruote anteriori sterzanti e due ruote posteriori fisse. Esempio di configurazioni con 4 ruote sono quelle che vanno sotto il nome di Ackermann steering, di cui si possono avere in generale quattro casi:

1. Due ruote posteriori standard passive non sterzanti con due ruote anteriori collegate tra di loro, attuate e sterzanti;
2. Due ruote posteriori collegate tra loro, standard, attuate e non sterzanti con due ruote anteriori collegate tra di loro, non attuate e sterzanti;
3. 4 ruote omnidirezionali svedesi attuate (1 motore per ruota);
4. 4 ruote sterzanti e attuate.

Nella prima configurazione (come anche nella seconda) l'angolo di sterzata è limitato ed è progettato per evitare slittamento in curva, quindi le curve devono avere un diametro più grande di quello dell'auto. La caratteristica fondamentale di questa configurazione è che ha ottima stabilità laterale in curva, ma manovrabilità limitata, infatti se si ha bisogno di ottima manovrabilità non si usa questo tipo di configurazione.

Per il progetto è stato usato il programma ROS (Robot Operating System). Esso adopera il modello URDF (Universal Robot Description Format), che è una collezione di file atti a fare una descrizione fisica del robot per dire al computer come il robot appare nella vita reale.

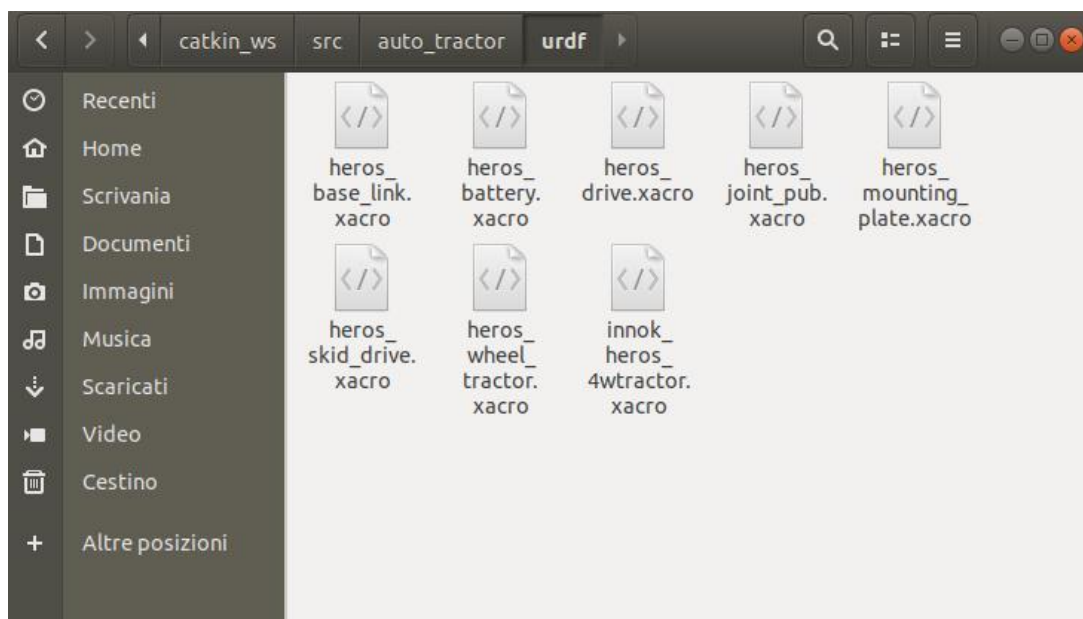


Figura 2: File URDF

I file URDF sono necessari per ROS per poter simulare situazioni in cui il robot potrebbe trovarsi.

Il file principale per la costruzione del trattore è *innok_heros_4wtractor.xacro*, nel quale vengono richiamati i restanti file che rappresentano le componenti del trattore.

Nella cartella sono presenti altri due file:

- *heros_joint_pub.xacro*, che contiene uno strumento per impostare e pubblicare valori del giunto per un dato URDF. Il package legge i parametri *robot_description*, trova tutti i giunti non fissi e pubblica il messaggio *JointState* con tutti i giunti definiti, in cui si esplica la posizione e la velocità del giunto e la forza o la coppia applicata nel giunto;
- *heros_skid_drive.xacro*, che è un controller per sistemi di ruote con meccanismo di sterzo. Il controllo è sotto forma di un comando di velocità, che viene sdoppiato e poi inviato alla singola ruota posteriore e alla singola ruota anteriore sterzante di un interasse sterzante. L'odometria viene calcolata in base al feedback dell'hardware e pubblicata. I topic presenti

sono `cmd_vel` (`geometry_msgs/Twist`), per la velocità lineare e angolare, e `odom` (`nav_msgs/Odometry`), che contiene la posa (divisa in posizione e orientamento) e la velocità.

Il codice usato è il seguente:

```
innok_heros_4wtractor.xacro M X
innok_heros_4wtractor.xacro
1 <?xml version="1.0"?>
2
3 <robot name="innok_heros_4w" xmlns:xacro="http://ros.org/wiki/xacro">
4
5   <xacro:property name="PI" value="3.1415926535897931"/>
6   <xacro:property name="wheel_radius" value="0.200" />
7   <xacro:property name="wheel_width" value="0.100" />
8   <xacro:property name="wheel_mass" value="5.4" />
9
10  <xacro:include filename="$(find auto_tractor)/urdf/heros_base_link.xacro"/>
11  <xacro:include filename="$(find auto_tractor)/urdf/heros_wheel_tractor.xacro"/>
12  <xacro:include filename="$(find auto_tractor)/urdf/heros_drive.xacro"/>
13  <xacro:include filename="$(find auto_tractor)/urdf/heros_battery.xacro"/>
14  <xacro:include filename="$(find auto_tractor)/urdf/heros_mounting_plate.xacro"/>
15
16  <xacro:include filename="$(find auto_tractor)/urdf/heros_joint_pub.xacro" />
17  <xacro:include filename="$(find auto_tractor)/urdf/heros_skid_drive.xacro" />
18
19  <heros_base_link />
20  <heros_drive front_rear='front' dx='0.782' flip_z='-1' />
21  <heros_drive front_rear='rear' dx='0.818' flip_z='1' />
22  <heros_battery name='battery_box' dx='0.336' />
23  <heros_mounting_plate wheels='4' dx='-0.02' />
24
25  <heros_joint_pub joints='joint_base_wheel_rear_left, joint_base_wheel_rear_right, joint_base_wheel_front_left, joint_base_wheel_front_right' />
26  <heros_skid_drive/>
```

Figura 3: Codice Visual Studio

Si può notare come nel codice sia stato usato *xacro*, un linguaggio XML fornito da ROS che consente di costruire file XML più corti e semplici rispetto ad URDF. Esso mette infatti a disposizione la possibilità di dichiarare delle macro. Nel caso in analisi sono state create macro che definiscono le strutture di link e giunti. Queste macro sono state poi richiamate all'interno del codice alla definizione di ogni link e giunto del robot. In questo modo, anziché andare a definire ad ogni richiamo la loro struttura, questa viene definita una volta soltanto. Il codice diventa meno prolisso e più semplice.

Il trattore è caratterizzato dalle seguenti componenti:

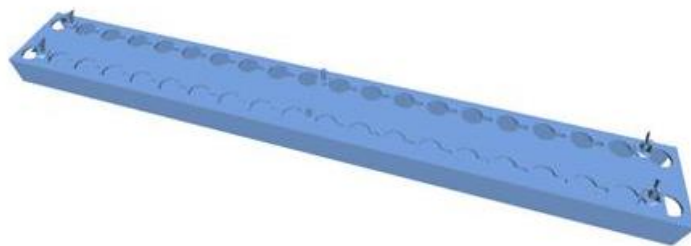


Figura 4: Beam



Figura 5: Battery Box

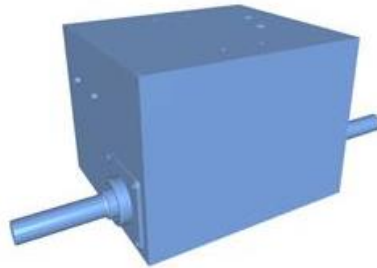


Figura 6: Drive Box

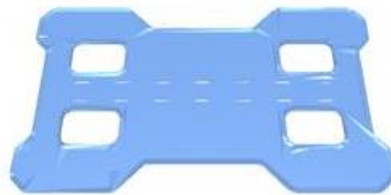


Figura 7: Mounting plate

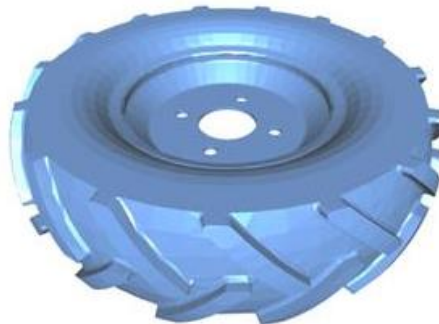


Figura 8: Wheel Tractor

Di seguito, viene riportato un esempio di mesh, importata nel file URDF e utilizzata all'interno della macro *heros_battery.xacro*.

```
<geometry>
| <mesh filename="package://auto_tractor/meshes/battery_box.STL" />
</geometry>
```

Figura 9: Esempio di mesh

Se si vogliono aggiungere parti al robot, bisogna creare altre forme e altri collegamenti nel file URDF e collegare questi al primo elemento (root) tramite joint. URDF, infatti, crea una struttura ad albero:

parte dalla root e scende alle “foglie” attraverso i joint. Per ogni file è ammessa una sola root, tutti gli altri elementi sono i cosiddetti “children”. Si usano i termini “parent link” e “child link” per stabilire la dipendenza tra i collegamenti.

Nel nostro caso di studio, si hanno i seguenti collegamenti:

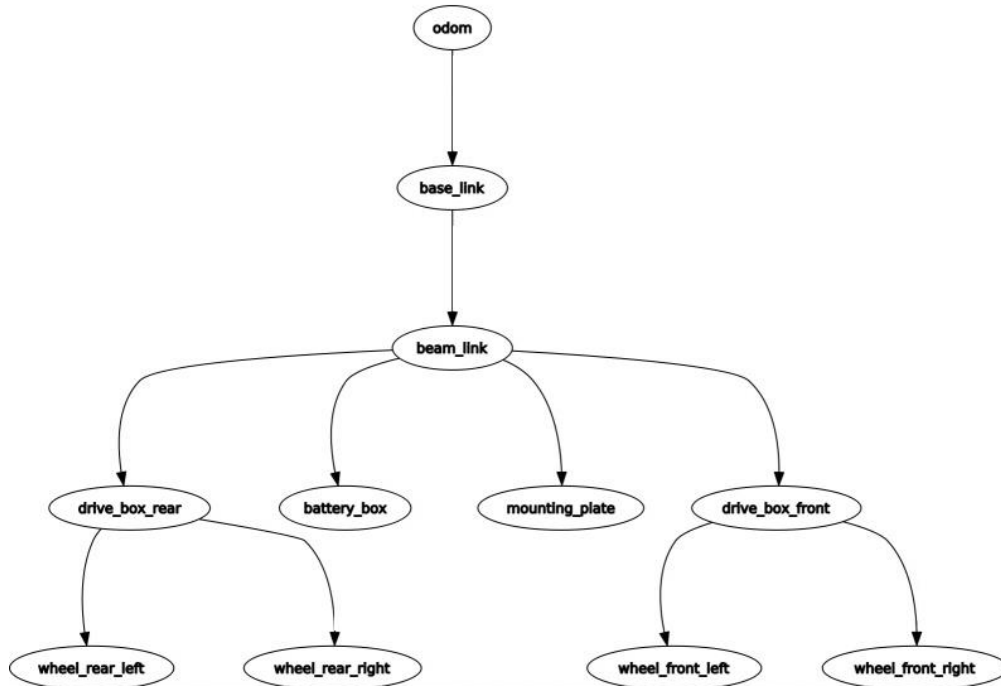


Figura 10: Struttura ad albero del trattore

Per ogni componente viene definito un link con il nome e il giunto, in cui viene indicato il parent link e il child link e la posizione in cui si deve trovare, come viene mostrato di seguito.

```
<link name="{link_name}" >
```

```
... link description ...
```

```
</link>
```

```
<joint name="{link_name}_joint" type="...">
```

```
<origin xyz="... .." rpy="... .." />
```

```
<parent link="..." />
```

```
<child link="{link_name}" />
```

```
<axis xyz="... .." />
```

```
</joint>
```

I giunti possono essere di differenti tipi:

- giunti non flessibili (fissi);

- giunti flessibili, che possono essere a loro volta
 - continui, che possono assumere qualsiasi angolo compreso nell'intervallo $[-\infty, +\infty]$ in radianti;
 - revolute, che ha limiti alto, basso, di massima velocità e di effort;
 - prismatici, si muovono lungo un asse;
 - planari, si muovono su un piano (2D);
 - floating, sono senza vincoli (3D).

All'interno del file *heros_wheel_tractor.xacro* viene inserito l'elemento di trasmissione, il quale è un'estensione del modello di descrizione del robot URDF utilizzato per descrivere la relazione tra un attuatore e un giunto. Ciò consente di modellare concetti come rapporti di trasmissione e collegamenti paralleli. Più attuatori possono essere collegati a più giunti attraverso una trasmissione complessa.

Ecco un esempio di elemento di trasmissione:

```
<transmission name="trans_wheel_${front_rear}_${left_right}">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="joint_base_wheel_${front_rear}_${left_right}">
    <hardwareInterface>hardware_interface/VelocityJointInterface</hardwareInterface>
  </joint>
  <actuator name="motor_wheel_${front_rear}_${left_right}">
    <mechanicalReduction>100</mechanicalReduction>
  </actuator>
</transmission>
```

Figura 11: Trasmissione

2. RVIZ E GAZEBO

La simulazione del robot è uno strumento essenziale nel toolbox della robotica. Un simulatore ben progettato consente di testare rapidamente algoritmi, progettare robot, eseguire test di regressione e addestrare il sistema di intelligenza artificiale utilizzando scenari realistici.

Per il progetto sono stati usati RViz e Gazebo.

RViz è uno strumento di visualizzazione 3D per applicazioni ROS. Offre una visualizzazione del modello robot, acquisisce informazioni sui sensori dai sensori dei robot e riproduce i dati acquisiti. Può visualizzare dati da videocamera, laser, dispositivi 3D e 2D, tra cui immagini e nuvole di punti. Gazebo offre la possibilità di simulare in modo accurato ed efficiente gruppi di robot in complessi ambienti interni ed esterni con grafiche di alta qualità e comode interfacce grafiche e programmatiche. All'interno del progetto è stata creata la cartella *launch*, in cui sono stati inseriti i file: *innok_heros_rviz.launch* e *innok_heros_gazebo.launch*.

Da terminale con il comando `roslaunch auto_tractor innok_heros_rviz.launch`, si apre la schermata di RViz.

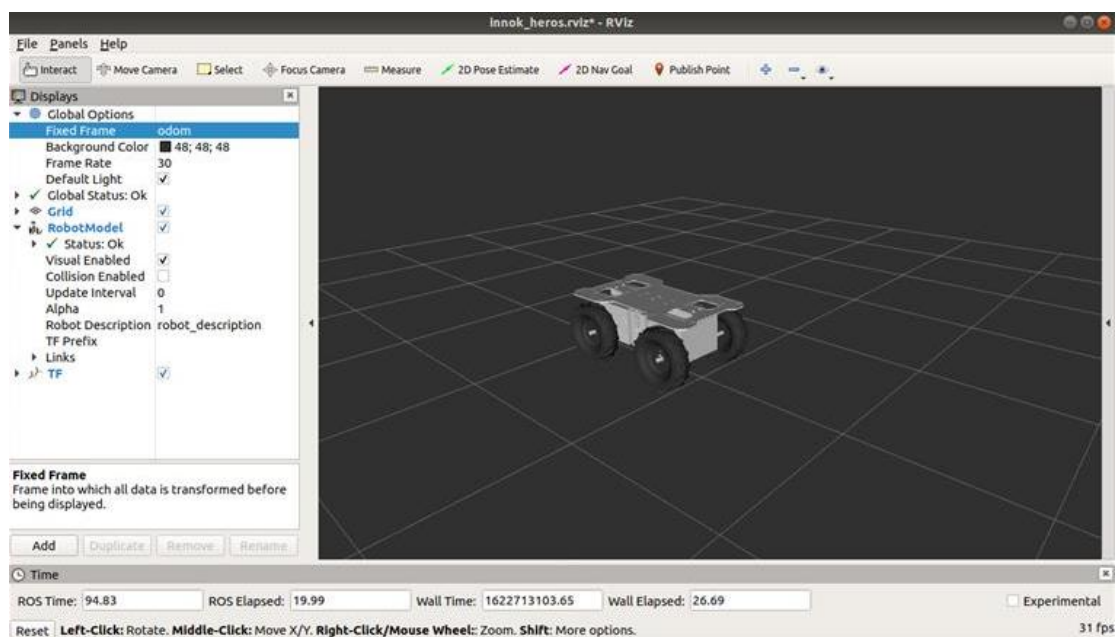


Figura 12: Trattore su RViz

Per la visualizzazione su RViz possono essere modificati i parametri che fanno capo a:

- *Global options*;
- *Grid*;
- *RobotModel*;
- *TF*.

Nella *Global Options*, i parametri chiave sono:

- *Fixed frame*: indica il nome del frame usato come riferimento per tutti gli altri frame. Si possono selezionare tutti i frame disponibili, anche se le scelte migliori sono map o odom;
- *Frame rate*: la frequenza massima utilizzata per aggiornare la vista 3D (30 o 60 FPS).

Il plugin della *Grid* permette di visualizzare una griglia normalmente associata al piano del pavimento.

Il plugin *Robot Model* permette di visualizzare il modello del robot secondo la descrizione dal modello URDF. I parametri chiave sono:

- *Visual enabled*: abilita o disabilita la visualizzazione 3D del modello;
- *Robot description*: il topic in cui è pubblicata la descrizione del robot;
- Espandendo la voce *Links*, si può vedere l'albero dell'intero modello con tutti i giunti e i link a disposizione e le relative posizioni e orientamenti nello spazio in riferimento al *fixed frame*.

Il plugin *TF* permette di visualizzare la posizione e l'orientamento di tutti i frame che compongono la *TF hierarchy*. Fondamentale per l'utilizzo di questo plugin è la capacità di abilitare/disabilitare la visualizzazione di singoli frame.

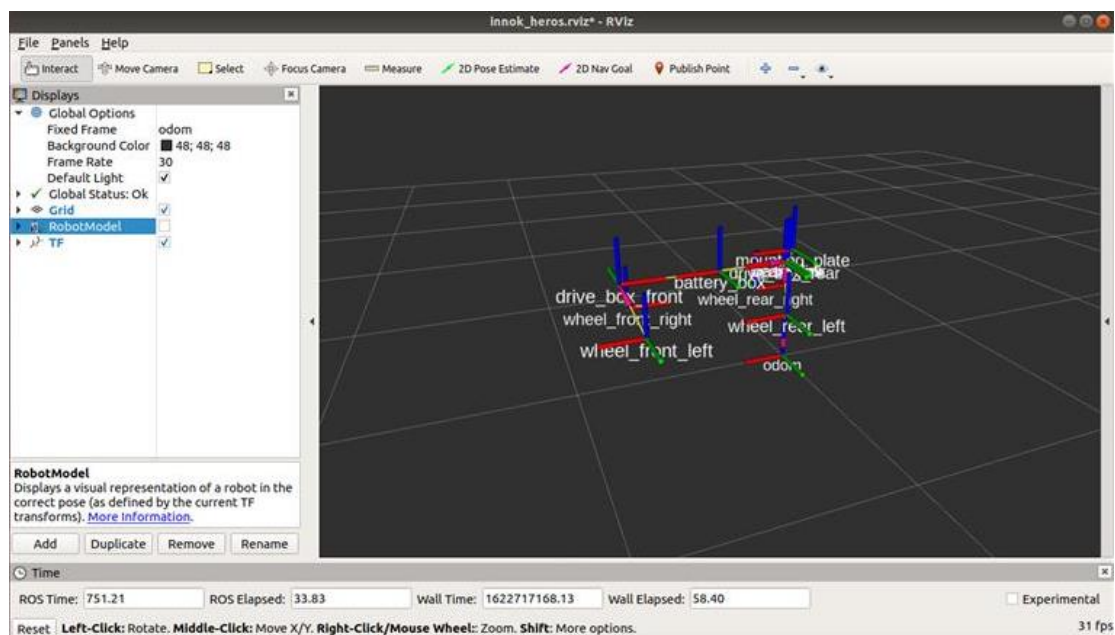


Figura 13: TF hierarchy

Da terminale con il comando `roslaunch auto_tractor innok_heros_gazebo.launch`, si apre la schermata di Gazebo. Questo launch file:

- Carica l'urdf nel ROS Parameter Server;
- Lancia un gazebo world vuoto;
- Esegue uno script python per chiedere a gazebo_ros di generare un robot URDF;
- Pubblica lo stato del robot su tf usando *joint_states* e *robot_description*.

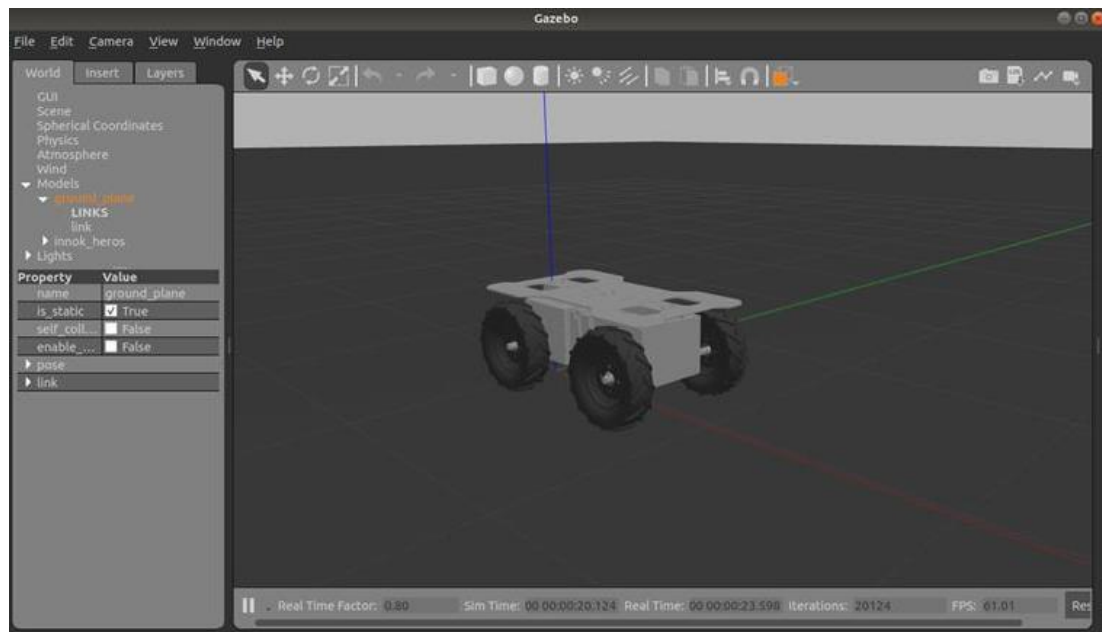


Figura 14: Trattore su Gazebo

2.1 Custom world

Si è passati a creare un file *world* personalizzato specifico per il robot e per il caso di studio. A tal scopo, volendo simulare una campagna, sono stati inseriti:

- un recinto per segnare il confine dell'ambiente;
- alberi;
- un gazebo;
- una macchina;
- due fantocci.

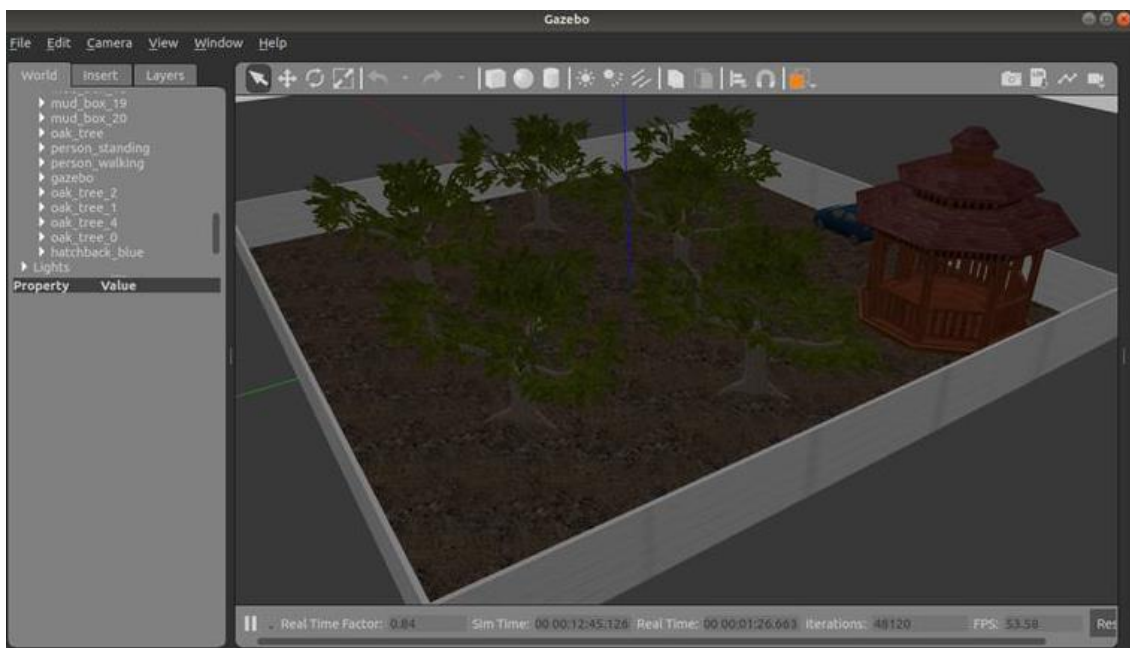


Figura 15: Custom world

Una volta terminata l'aggiunta di elementi all'ambiente, è stata creata una cartella *world* all'interno del progetto in cui è stato salvato il file come *countryside.world*.

Nel file *innok_heros_gazebo.launch* è stata inserita la riga di codice riportata nella figura seguente per richiamare il file salvato, in modo tale che, una volta aperto Gazebo, l'ambiente viene spawnato con il trattore al suo interno.

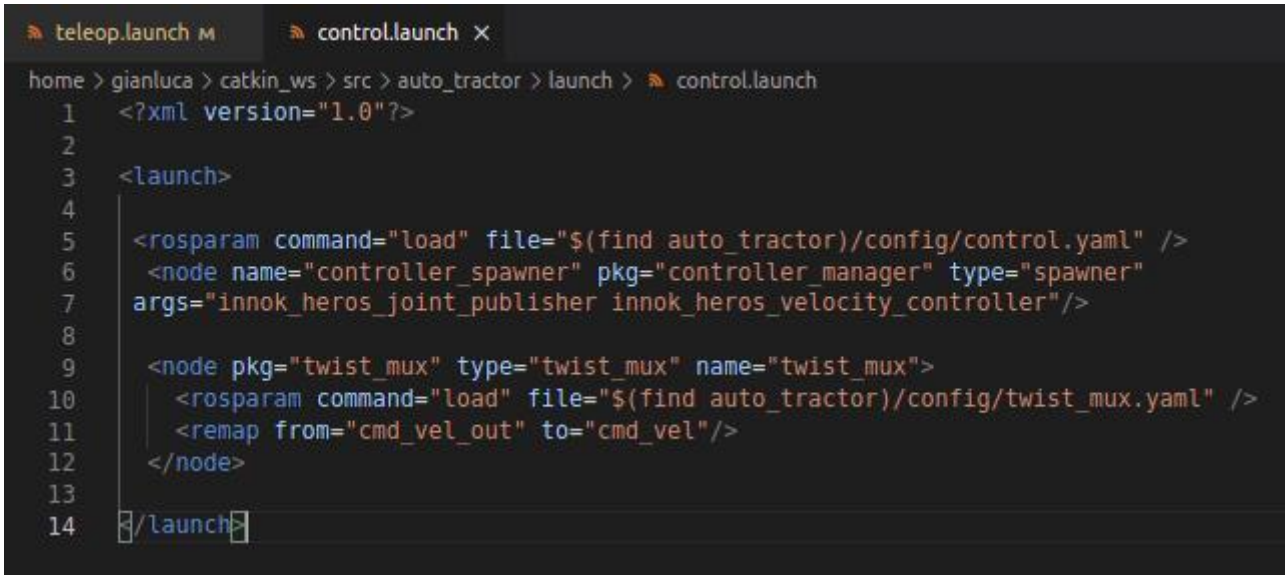
```
<include file="$(find gazebo_ros)/launch/empty_world.launch">
  <arg name="world_name" default="$(find auto_tractor)/world/countryside.world" />
  <arg name="debug" value="false" />
  <arg name="gui" value="$(arg gui)" />
  <arg name="paused" value="$(arg paused)"/>
  <arg name="use_sim_time" value="$(arg use_sim_time)"/>
  <arg name="headless" value="false"/>
</include>
```

Figura 16: Righe di inclusione countryside

3. CONTROLLO

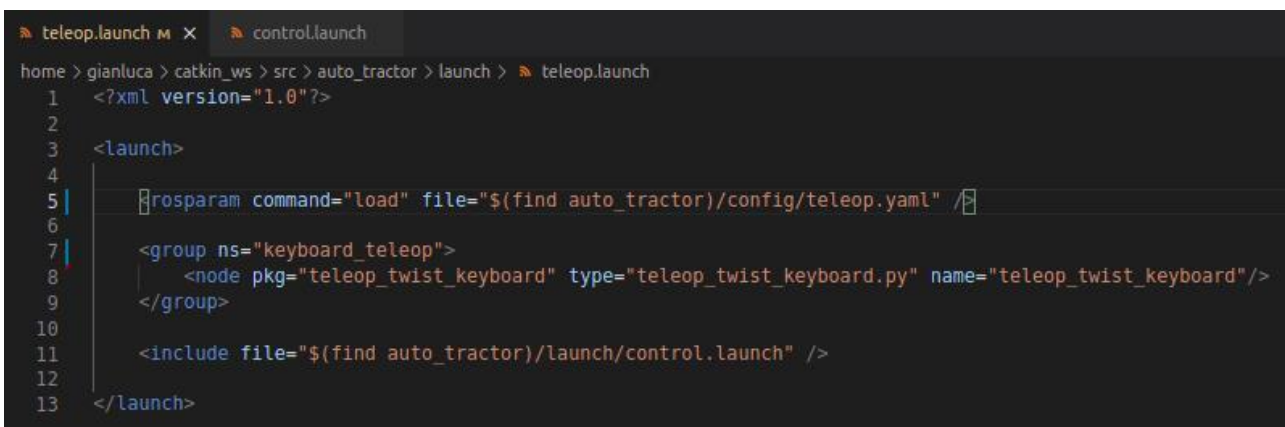
Per il controllo, sono stati aggiunti nella cartella *launch* due file:

- *control.launch*;
- *teleop.launch*.



```
home > gianluca > catkin_ws > src > auto_tractor > launch > control.launch
1  <?xml version="1.0"?>
2
3  <launch>
4
5      <rosparam command="load" file="$(find auto_tractor)/config/control.yaml" />
6      <node name="controller_spawner" pkg="controller_manager" type="spawner"
7      args="innok_heros_joint_publisher innok_heros_velocity_controller"/>
8
9      <node pkg="twist_mux" type="twist_mux" name="twist_mux">
10         <rosparam command="load" file="$(find auto_tractor)/config/twist_mux.yaml" />
11         <remap from="cmd_vel_out" to="cmd_vel"/>
12     </node>
13
14 </launch>
```

Figura 17: *control.launch*



```
home > gianluca > catkin_ws > src > auto_tractor > launch > teleop.launch
1  <?xml version="1.0"?>
2
3  <launch>
4
5      <rosparam command="load" file="$(find auto_tractor)/config/teleop.yaml" />
6
7      <group ns="keyboard_teleop">
8          <node pkg="teleop_twist_keyboard" type="teleop_twist_keyboard.py" name="teleop_twist_keyboard"/>
9      </group>
10
11      <include file="$(find auto_tractor)/launch/control.launch" />
12
13 </launch>
```

Figura 18: *teleop.launch*

In questi file vengono utilizzati tre pacchetti: *twist_mux*, *teleop_twist_keyboard* e *controller_manager*.

Per quanto riguarda il *twist_mux*, quando c'è più di una singola fonte per spostare un robot con un messaggio *geometry_msgs::Twist*, è importante fare il multiplex di tutte quelle fonti di input per ottenerne uno solo che vada al controller (es. *diff_drive_controller*). Questo pacchetto fornisce un nodo che fa parte di una lista di topic che pubblicano messaggi di tipo *geometry_msgs::Twist* e ne fa il multiplex utilizzando uno schema basato sulla priorità.

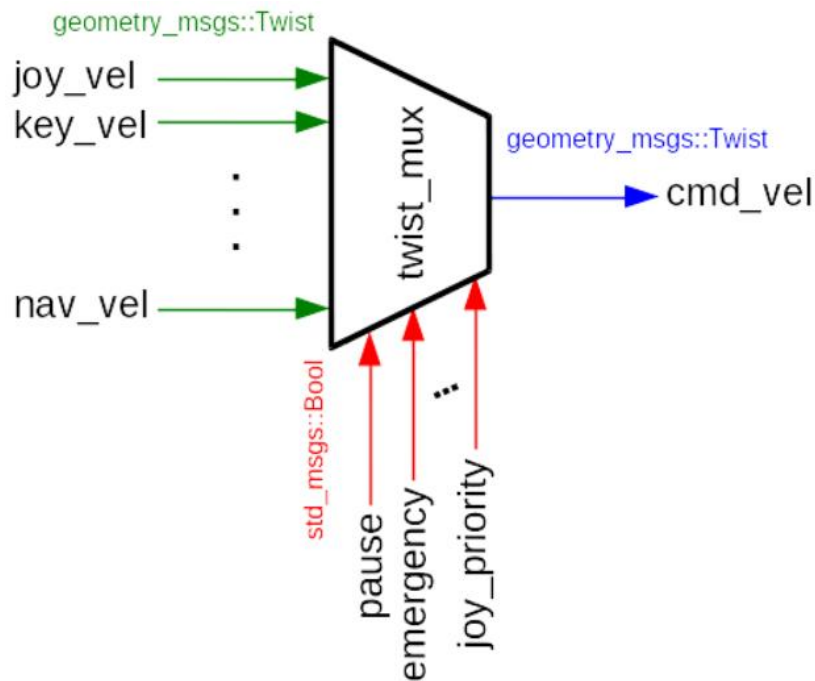


Figura 19: Diagramma di un nodo *twist_mux*

Nel nostro caso i messaggi sono pubblicati in *cmd_vel*, che è l'output del topic *geometry_msgs::Twist* del twist multiplexer.

Da terminale scrivendo il comando *rostopic echo /cmd_vel*, sono mostrate le velocità lineari e angolari del trattore rispetto ai tre assi x,y,z.

```
gianluca@gianluca-HP-Notebook:~$ rostopic echo /cmd_vel
linear:
  x: 0.5
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
linear:
  x: 0.5
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: -1.0
---
```

Figura 20: Velocità lineari e angolari del trattore

Con *teleop_twist_keyboard* è possibile controllare il trattore da tastiera (in anello aperto) con:

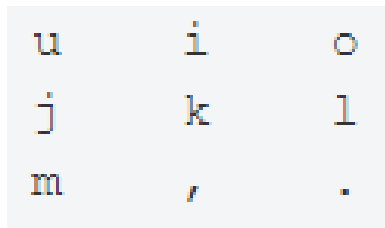


Figura 21: Comandi per controllare il trattore

Altri comandi utili per il controllo possono essere dati nel seguente modo:

- q/z per aumentare/diminuire le velocità massime del 10%;
- w/x per aumentare/diminuire solo la velocità lineare del 10%;
- e/c per aumentare/diminuire solo la velocità angolare del 10%;
- qualsiasi altro tasto per lo stop.

Dopo aver avviato il launch file per Gazebo, in un'altra finestra del terminale si utilizza il comando `roslaunch auto_tractor teleop.launch` per attivare il controllo da tastiera.

L'ultimo pacchetto è il *controller_manager*, che fornisce l'infrastruttura per interagire con i controller. A seconda che si stia eseguendo i controller da un file di avvio, dalla riga di comando o da un nodo ROS, il controller manager fornisce strumenti diversi per eseguire i controller.

Nei due file *launch* introdotti per il controllo vengono richiamati tre file YAML presenti nella cartella *config*:

- *control.yaml*;
- *teleop.yaml*;
- *twist_mux.yaml*.

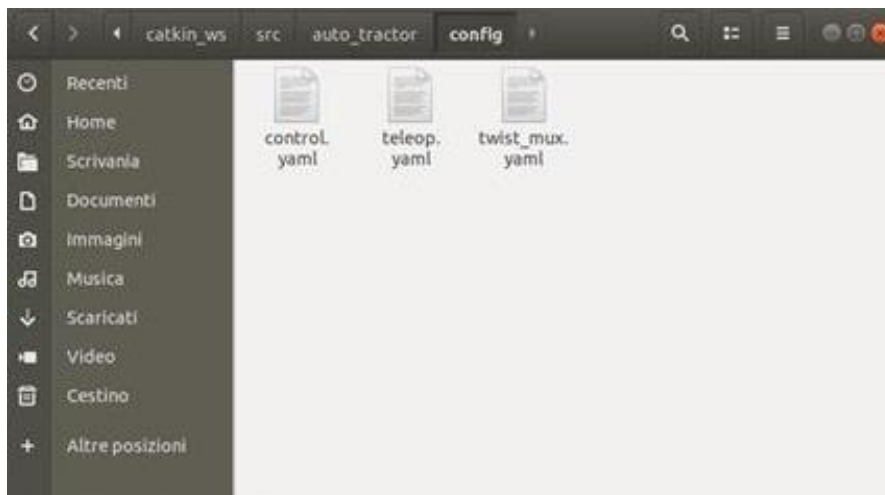


Figura 22: Cartella config

YAML è uno standard di serializzazione dei dati leggibile dall'uomo che può essere utilizzato insieme a tutti i linguaggi di programmazione e viene spesso utilizzato per scrivere file di configurazione.

L'acronimo YAML sta per “*YAML Ain't Markup Language*”, indicandolo come flessibile e orientato ai dati. Infatti, può essere utilizzato con quasi tutte le applicazioni che necessitano di archiviare o trasmettere dati. La sua flessibilità è in parte dovuta al fatto che YAML è composto da frammenti di altri linguaggi.

Le impostazioni del controller devono essere salvate in un file *.yaml*, nel nostro caso *control.yaml*, che viene caricato sul param server tramite il file *roslaunch*. I controllori presenti sono: *joint_state_controller* e *diff_drive_controller*.

Il *joint_state_controller* è una classe in uno dei pacchetti di *ros_control*. Quest'ultimo framework fornisce la capacità di implementare e gestire i controller del robot, che consiste principalmente in un meccanismo di feedback (solitamente è usato un loop PID), che può ricevere un setpoint e controllare l'uscita utilizzando il feedback degli attuatori.

Il *joint_state_controller* non è un nodo autonomo, ma si interfaccia con l'hardware. Le interfacce hardware sono utilizzate dal controllo ROS in combinazione con uno dei controller ROS per inviare e ricevere comandi all'hardware. Il *joint_state_controller* trasforma i dati da una rappresentazione *ros_control* interna a messaggi di tipo *JointState* e li pubblica. Le interfacce *JointState* sono un tipo di interfaccia hardware utile per supportare la lettura dello stato di un array di giunti, ognuno dei quali ha una posizione, velocità e forza o coppia.

Il *diff_drive_controller* è un controller per sistemi di ruote motrici differenziali. Il controller funziona con i giunti delle ruote attraverso un'interfaccia di velocità. Nel nostro caso, i giunti sono *joint_base_wheel_rear_left* e *joint_base_wheel_rear_right*, cioè quelli delle ruote posteriori.

Sono stati poi impostati i limiti di velocità e accelerazione lineare e angolare come mostrato nella seguente figura.

```
# Velocity and acceleration limits
linear:
  x:
    has_velocity_limits      : true
    max_velocity             : 1.0   # m/s
    has_acceleration_limits : true
    max_acceleration         : 3.0   # m/s^2
angular:
  z:
    has_velocity_limits      : true
    max_velocity             : 2.0   # rad/s
    has_acceleration_limits : true
    max_acceleration         : 6.0   # rad/s^2
```

Figura 23: Limiti di velocità e accelerazione

Negli altri due file (*teleop.yaml* e *twist_mux.yaml*) vengono impostati parametri utili per i due pacchetti *teleop_twist_keyboard* e *twist_mux*, anche loro necessari per il controllo del trattore.

4. SENSORI

In ogni sistema di controllo c'è bisogno di attuazione, controllo e sensori. In particolare, nella robotica mobile c'è bisogno di tutta una serie di sensori per ovviare a un problema noto come “*dead reckoning*”.

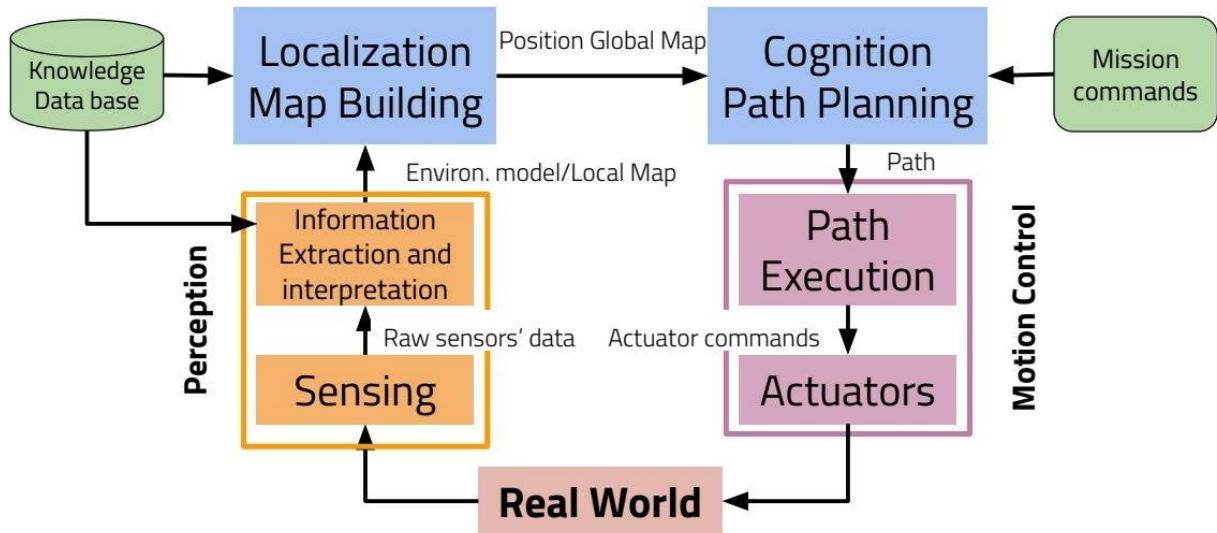


Figura 24: Loop di controllo

Un sensore può essere visto come un sistema che ha come ingresso lo stato e come uscita la misura.

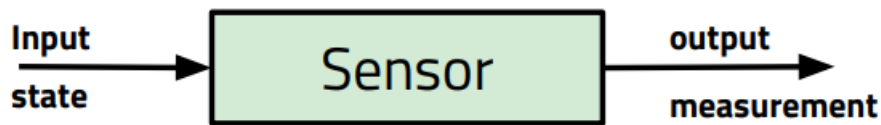


Figura 25: Sistema sensore

I sensori possono essere attivi, cioè emettono energia e misurano la reazione dell'ambiente a questa emissione, che passivi, cioè misurano l'energia proveniente dall'ambiente.

Le caratteristiche di un sensore sono le seguenti:

- Range, si esplicano limite superiore e inferiore degli ingressi al sensore, cioè l'insieme di tutti gli stati che possono essere misurati. Esso deve essere appropriato per l'applicazione;
- Range dinamico, definito dallo spread tra input più basso e più alto, spesso misurato in decibel:

$$DR = 20 \log \left(\frac{x_{max}}{x_{min}} \right)$$

- Full scale, indica i limiti superiore e inferiore dei valori di output del sensore, in altre parole è la differenza tra il minimo e il massimo output;
- Linearità, dice quanto la caratteristica del sensore è vicina alla caratteristica lineare. Una caratteristica lineare tra ingresso e uscita è auspicabile per i sensori;

- Larghezza di banda/Frequenza, abilità di inseguire input sinusoidali continui, frequenza con cui si aggiornano le misure del sensore. Frequenze più alte sono necessarie per gli anelli di controllo interni, frequenze più basse per la pianificazione e il mapping;
- Risoluzione, minima differenza tra i valori di due misure. Per i sensori analogici, di solito è la risoluzione A/D;
- Accuratezza, differenza tra il valore misurato dal sensore e il valore reale;
- Precisione, riproducibilità dei risultati;
- Errore sistematico, deterministico e causato da fattori che possono essere modellati (per esempio le distorsioni di una camera);
- Errore randomico, non deterministico e può essere modellato come variabili random con distribuzioni di probabilità note, spesso gaussiane.

I sensori utilizzati all'interno del progetto sono stati installati sullo chassis del trattore.

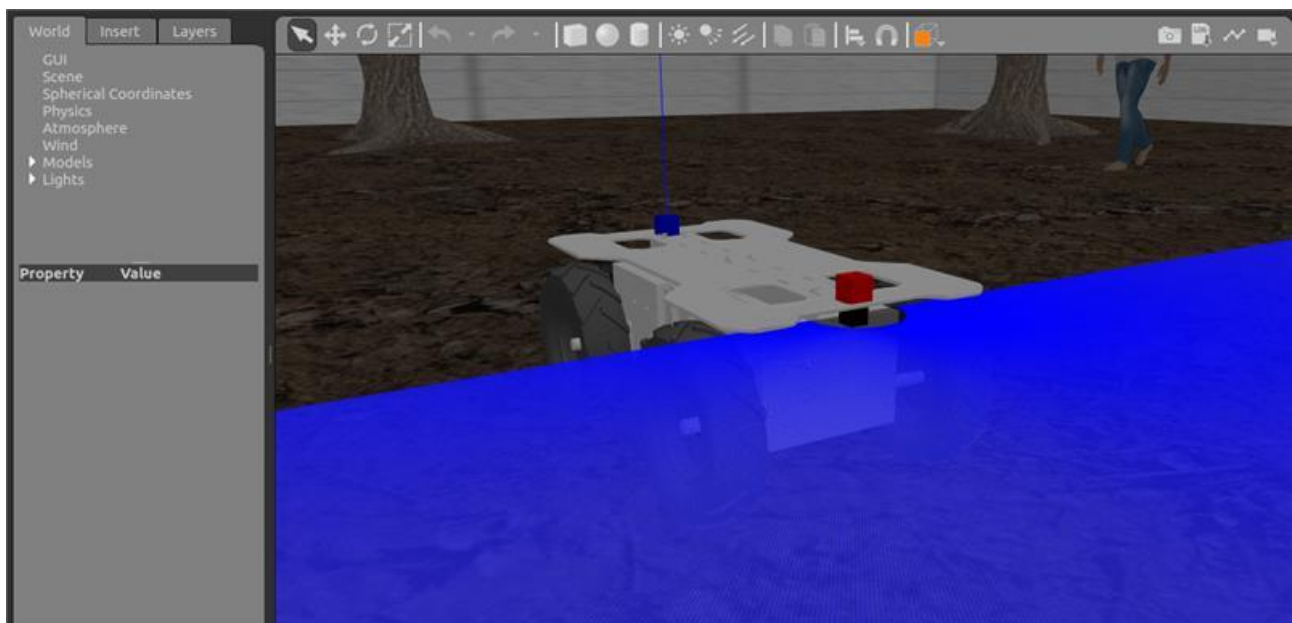


Figura 26: Trattore con i tre sensori su Gazebo

Come si può vedere dalla figura sopra, sono presenti tre sensori rappresentati dal cubo nero, dal cubo rosso e cubo blu, rispettivamente il laser (di cui il fascio blu indica il range di scansione che va da -90° a 90°), la depth camera e la camera. In particolare, nella parte anteriore sono installati la depth camera e il laser e nella parte posteriore la camera.

I sensori LIDAR (acronimo di *Light Detection and Ranging* o *Laser Imaging Detection and Ranging*) hanno una lunghezza d'onda molto stretta, da cui la capacità di rilevare facilmente i falsi positivi. Essi vengono sovente realizzati nella guida autonoma perché possono restituire una nuvola di punti.

Difatti, all'interno di questi sensori vi è un laser che, ruotando su sé stesso, è in grado di acquisire le distanze a 360°, a meno di occlusioni.

Inoltre, possono disporre di più livelli, in numero pari a quello dei fasci laser.

Ogni punto della nuvola è un punto nello spazio di cui è nota la distanza.

Con particolari tecniche di filtraggio e di elaborazione della nuvola di punti, si possono riconoscere caratteristiche di oggetti o effettuare il riconoscimento di oggetti.

Questi sensori sono i più costosi nella robotica.

Si basano su uno specchio rotante che riflette la luce in una direzione, che dipende dall'angolo di orientazione rispetto all'emettitore, e la luce riflessa viene captata da un rilevatore.

Si può sfruttare l'effetto Doppler per misurare anche la velocità dell'oggetto in virtù dell'angolo di sfasamento.

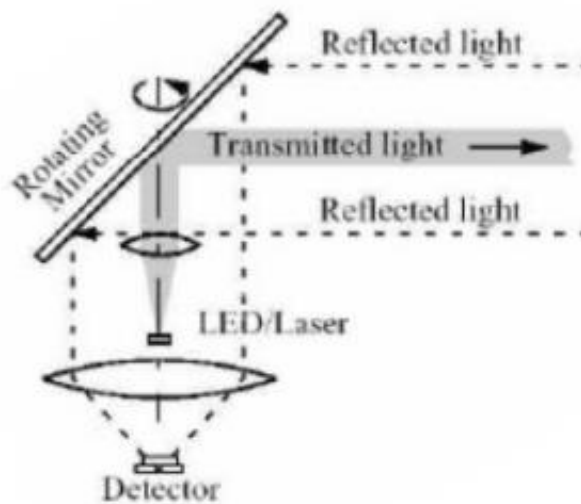


Figura 27: Sensore LIDAR

Le telecamere forniscono dati di immagine al robot che possono essere utilizzati per l'identificazione di oggetti, il monitoraggio e le attività di manipolazione.

La depth camera è una telecamera a tempo di volo (*time of flight camera*, TOF-camera). Questo tipo di camera è uno strumento che permette di stimare in tempo reale la distanza tra la telecamera e gli oggetti o la scena inquadrati, misurando il tempo che occorre ad un impulso luminoso per percorrere il tragitto telecamera-oggetto-telecamera (tempo di volo).

La scena è quindi acquisita in modo completo come per una foto, ma la misura della distanza è effettuata indipendentemente su ciascun pixel, consentendo così la ricostruzione 3D dell'oggetto o della scena misurata. Questa tecnica si pone quindi in alternativa a quella dei Laser scanner 3D che invece analizzano la scena una linea per volta.

Nella parte posteriore del trattore è stata posizionata una camera RGB, che è una telecamera dotata di un sensore CMOS standard attraverso il quale vengono acquisite le immagini a colori di persone e oggetti. Il CMOS (MOS complementare) è un sensore fotografico nato come alternativa ai CCD (dispositivi ad accoppiamento di carica). Un sensore fotografico è un sistema complesso, caratterizzato da più elementi che interagiscono tra loro. I colori della scena inquadrata vengono percepiti in questa maniera: quando i fotoni, che trasportano le informazioni della luce, si scontrano e interagiscono con gli elementi silicei del sensore, vengono rilasciati degli elettroni e si forma una piccola scarica elettrica (fenomeno noto come effetto fotoelettrico). Gli elettroni che si liberano vengono raccolti dai singoli pixel, in cui ognuno di essi ha una capacità massima di elettroni che può raccogliere. Un pixel può visualizzare solo un singolo colore ed esso è determinato dalla quantità e dal tipo di informazioni che raccoglie dalla luce. Più elettroni si accumulano su un pixel, più chiaro è il corrispondente valore tonale nell'immagine. Il sensore fotografico non è comunque in grado da solo di determinare il colore tramite il conteggio degli elettroni e ha bisogno di un filtro di colore. All'interno del filtro, combinando il valore tonale del grigio ricevuto e le informazioni di colore intrinseche del filtro, viene determinato il colore finale per ciascun pixel. La differenza tra i sensori CCD e CMOS risiede nell'acquisizione, dovuta ai sistemi di lettura degli elettroni. Quando finisce il tempo di esposizione, nel sensore CCD gli elettroni eccitati dai fotoni, migrano verso la base del sensore. Il trasferimento avviene per variazione dei voltaggi tra i vari pixel, le informazioni vengono amplificate e convertite in segnale analogico. Nei sensori CMOS, invece, la codifica delle informazioni generate avviene direttamente sul pixel. Successivamente il segnale dei singoli fotositi viene raccolto per trasformare i dati in immagine, usando un segnale digitale.

I tre sensori sono stati inseriti all'interno della cartella URDF nel file *innok_heros_4wtractor.xacro*. Attraverso il codice sottostante è stato creato un collegamento (*link*) che viene caratterizzato da alcune proprietà come la geometria (è stato usato un cubo per rappresentare tutti e tre i sensori), la massa, l'inerzia e il materiale. Oltre al link, nel file URDF viene aggiunto un giunto per collegare i sensori al trattore e per scegliere la posizione (*origin*) dei sensori stessi. Nell'ultimo blocco del codice è richiamato il Gazebo plugin, che consente il funzionamento dei sensori.

```
<link name="{link_name}">
  ... link description ...
</link>
<joint name="{link_name}_joint" type="fixed">
  <origin xyz="... .." rpy="... .." />
  <parent link="base_link" />
```

```

<child link="${link_name}" />
<axis xyz="... .." />
</joint>

<gazebo reference="${link_name}">
  <sensor type="..." name="...">
    ... sensor parameters ...
    <plugin name="..." filename="...">
      ... plugin parameters ...
    </plugin>
  </sensor>
</gazebo>

```

Per visualizzare il trattore con i sensori installati, bisogna utilizzare il comando *roslaunch auto_tractor innok_heros_gazebo.launch*. Aprendo un'altra scheda all'interno del terminale, si possono vedere elencati i topic con il comando *rostopic list*: tra questi ci sono quelli che fanno riferimento alla camera e alla depth camera.

Scrivendo i seguenti comandi si può visualizzare a schermo l'output delle camere:

- *roslaunch image_view image_view image:=/camera/image_raw*

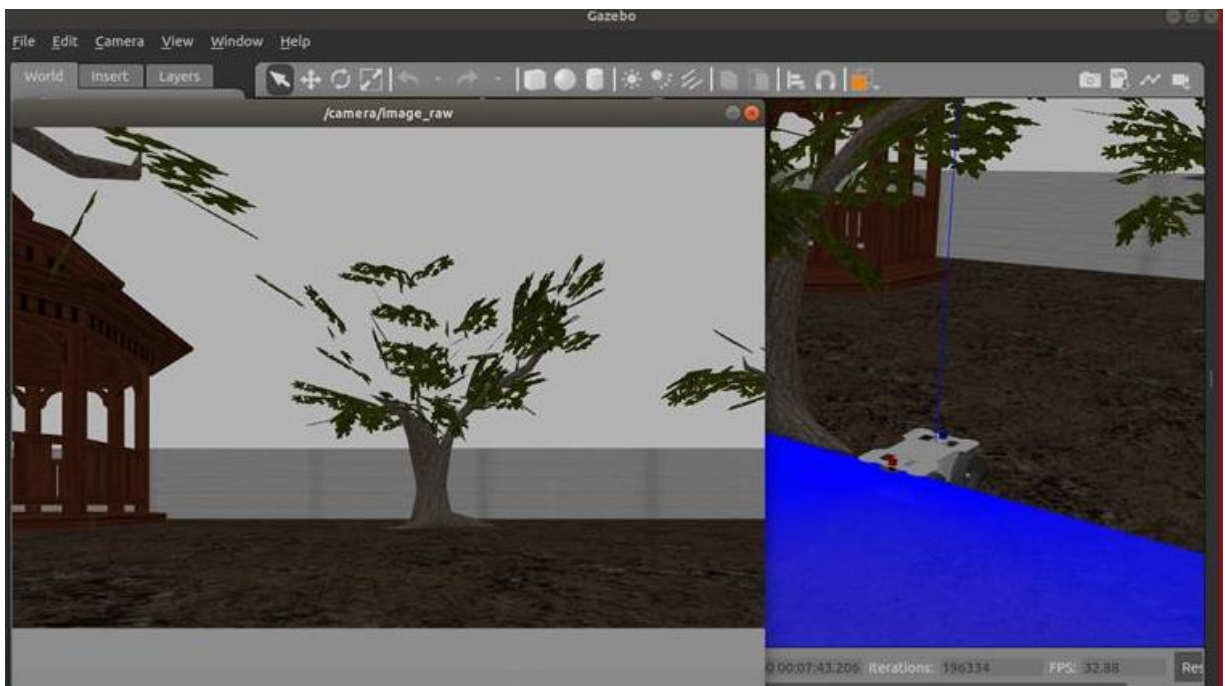


Figura 28: Trattore e ripresa dalla camera

- `roslaunch image_view image_view image:=/depth_camera/color/image_raw`

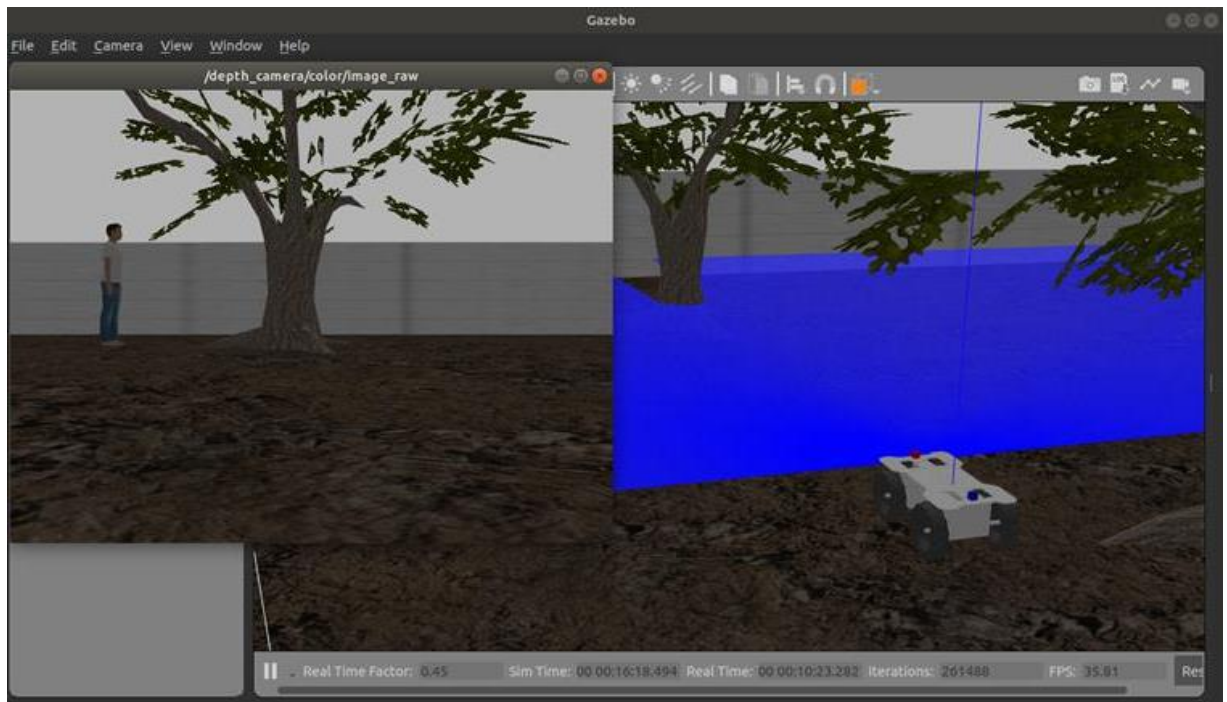


Figura 29: Trattore e ripresa dalla depth camera (color)

- `roslaunch image_view image_view image:=/depth_camera/depth/image_raw`

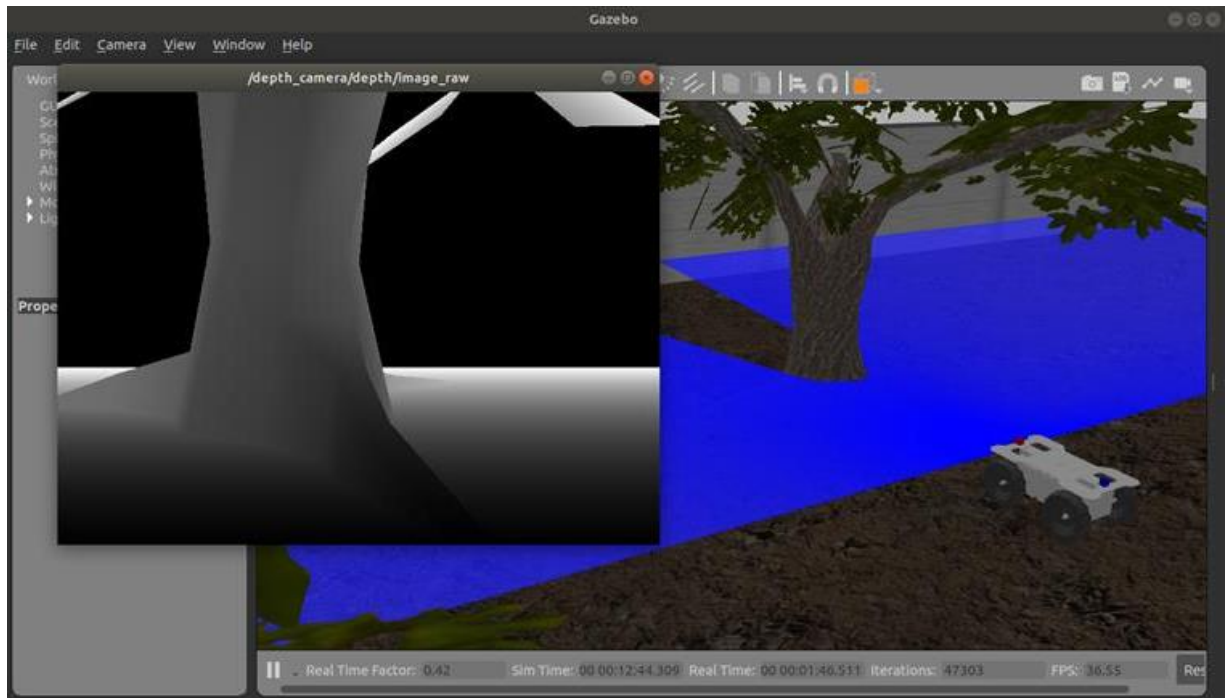


Figura 30: Trattore e ripresa dalla depth camera (depth)

Su RViz grazie all'aggiunta del topic *LaserScan* è possibile notare come sulla griglia siano presenti dei tratti rossi, che rappresentano gli ostacoli. È possibile sempre grazie al topic modificare lo stile, la dimensione e il colore con cui vengono mostrati i tratti, ad esempio, si possono usare punti, quadrati, sfere, ecc.

Su RViz si è anche in grado di visualizzare le immagini riprese dalla camera o dalla depth camera inserendo il topic di interesse.

Il comando per aprire RViz è sempre *roslaunch auto_tractor innok_heros_rviz.launch*.

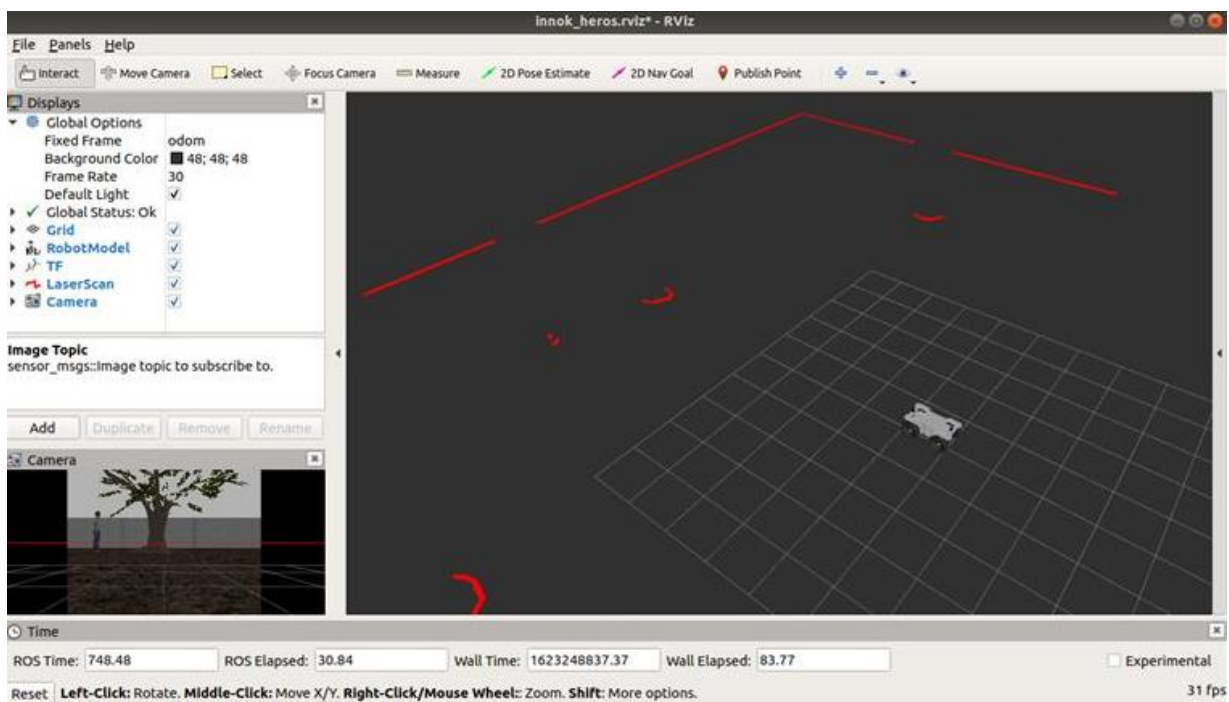


Figura 31: Trattore con ostacoli su RViz (LaserScan aggiunto)

5. ELABORAZIONE DATI SENSORE LASER

I dati *LaserScan* sono tra i dati più usati nel mondo della robotica. Molti robot mobili li usano per poter avere un riferimento in merito alle distanze dagli oggetti più vicini o, semplicemente, per misurare la distanza rispetto a qualche obiettivo.

Misurare la distanza tra i diversi componenti che possono trovarsi in un ambiente è molto importante per un robot, ad esempio permette di eseguire una navigazione ottimale con il suo uso principale che consiste nell'evitare ostacoli, oltre a poter eseguire una mappatura dell'ambiente in cui si sposterà il robot.

Il *LaserScan* Data contiene i dati che il LIDAR sta raccogliendo. I dati sono distribuiti nella seguente maniera:

- *Angle_minimum* e *Angle_maximum* che sono gli angoli di partenza dello scan in radianti;
- *Angle_increment* che è la distanza angolare tra ogni fascio. Il LIDAR sta mandando in pratica dei fasci e ognuno di questi si sta scontrando con gli ostacoli per poi tornare al LIDAR, in questo modo si può sapere se c'è un ostacolo lungo il percorso. Se un fascio va fino all'infinito senza tornare, vuol dire che non ci sono ostacoli sul cammino. Invece, se il fascio del laser torna, sapremo che a una distanza *x* in una certa direzione c'è un ostacolo che non sta facendo passare il fascio.
- *Time_increment*, è il tempo tra le misure dei fasci del laser;
- *Scan_time*, è il tempo tra ogni scan in secondi;
- *Range_minimum* e *Range_maximum* è la distanza in cui ogni fascio può rilevare qualcosa, ad esempio se il fascio può rilevare qualcosa fino a 100 m, ma c'è qualcosa oltre questi 100 m, il laser non è "forte" abbastanza per rilevarlo. In pratica questo è il range del laser.
- *Range_array* contiene tutti i rilevamenti;
- *Intensity_array* contiene l'intensità di questi rilevamenti.

È stato creato un codice in python che è in grado di visualizzare su terminale la distanza frontale e laterale del robot dagli ostacoli.

Si procede con l'analisi del codice.

Si inizia facendo l'import del messaggio *LaserScan* dal pacchetto *sensor_msgs.msg*.

Viene quindi definita una python class che è stata chiamata "Robot", in cui con *def_init_()* si fa il construct della classe.

```
#!/usr/bin/env python

import rospy

from sensor_msgs.msg import LaserScan

class Robot():
    def __init__(self):
```

Il *Class subscriber* sottoscrive sullo *scan topic* e usa messaggi di tipo *LaserScan*. Per avere informazioni sullo *scan topic* stesso, è possibile usare da terminale il comando *rostopic info /laser_scan/scan*, che mostra il tipo di messaggi usati dal topic. In altre parole, lo *scan topic* è il “posto” dove vengono pubblicati i messaggi provenienti dal laser.

Con il comando *rostopic echo /laser_scan/scan -n1* si può vedere l’ultimo messaggio pubblicato contenente tutti i dati elencati sopra (*Angle_minimum* e *Angle_maximum*, *Angle_increment*, ecc.), quindi si hanno tutte le informazioni di questo specifico laser, tra cui tutte le letture di tutti i fasci del laser. Questi messaggi sono pubblicati costantemente sullo *scan topic*, quindi il laser raccoglie costantemente informazioni e le pubblica.

Ogni volta che un messaggio è pubblicato nello *scan topic*, si sta per innescare *self.scan_callback*.

Vengono poi inizializzati i valori *a*, *b* e *c*, e il *ctrl_c* inizializzato come *FALSE*.

```
self.robot_sub = rospy.Subscriber("/laser_scan/scan", LaserScan, self.scan_callback)
self.a = 0.0 #right
self.b = 0.0 #front
self.c = 0.0 #left
self.ctrl_c = False
```

Si inizializza il rate a 10 Hz e viene aggiunto uno *shutdownhook* che viene innescato quando il programma è fermato con il comando *ctrl_c*.

```
self.rate = rospy.Rate(10) #10hz
rospy.on_shutdown(self.shutdownhook)
```

I valori inizializzati *a*, *b* e *c* vengono poi aggiornati con i dati reali dal laser. Dal *ranges array* si prende il valore che è nella posizione 180 e si salva in *a*. Per *b* invece dalla lunghezza totale dell’array si divide per 2. Ci sono 720 posizioni, cioè ci sono 720 fasci differenti che stanno coprendo 180°, infatti, da *Angle_minimum* e *Angle_maximum* è possibile affermare che il laser copre metà circonferenza. Questa è la ragione per cui stiamo prendendo i valori 180(=*a*), 360(=*b*), 540(=*c*).

Con la funzione *read_laser*, vogliamo fare il print di questi valori, assicurandoci grazie al ciclo *while* che il programma non è finito.

```
def scan_callback(self, msg):
    #print len(msg.ranges) #720
    self.a = msg.ranges[180]
    self.b = msg.ranges[len(msg.ranges)/2]
    self.c = msg.ranges[540]

def read_laser(self):
    while not self.ctrl_c:
        if self.b > 100:
            self.b = 5
        if self.a > 100:
            self.a = 5
        if self.c > 100:
            self.c = 5

    print "a = " + str(self.a) + " b = " + str(self.b) + " c = " + str(self.c)
```

Si può notare che, se l'ostacolo si trova a una distanza superiore di 100 m in una delle tre direzioni, il valore stampato è impostato a 5.

Il *shutdownhook* si innesca quando *ctrl_c* è TRUE.

```
def shutdownhook(self):
    self.ctrl_c = True
```

Con la funzione *main* si inizializza la classe, mentre la funzione *read_laser* è usata per stampare i valori.

```
if __name__ == '__main__':
    rospy.init_node('robot_test', anonymous=True)
    robot_object = Robot()
    try:
        robot_object.read_laser()

    except rospy.ROSInterruptException:
        pass
```

Si può modificare il codice in maniera tale da controllare il flusso dei messaggi: in questa maniera è possibile osservare in tempo reale come si modificano i valori a, b e c aggiungendo un qualunque ostacolo lateralmente o frontalmente rispetto al robot.

A tale scopo il rate è impostato a 1, così da far pubblicare un messaggio al secondo (1 Hz).

```
self.rate = rospy.Rate(1) #1hz
rospy.on_shutdown(self.shutdownhook)
```

Si inserisce `self.rate.sleep()` dopo il comando `print` che rallenterà la stampa dei valori a 1 per secondo.

```
print "a = "+ str(self.a)+ " b = "+ str(self.b)+ " c = "+ str(self.c)
self.rate.sleep()
```

Modificando il codice appena descritto, è stata impostata una distanza massima dall'ostacolo oltre la quale il robot non si può muovere ed è possibile visualizzare su terminale in tempo reale la distanza frontale. A tal fine, dopo `import LaserScan`, si inserisce un messaggio di tipo `Twist` ottenuto dal `geometry_msgs.msg`.

```
import rospy

from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist
```

Si crea un nuovo publisher per muovere il robot, che pubblicherà nel topic `cmd_vel` messaggi di tipo `Twist` e si definisce la dimensione della coda per esempio a 1.

```
self.robot_sub = rospy.Subscriber("/laser_scan/scan", LaserScan, self.scan_callback)
self.robot_pub = rospy.Publisher("/cmd_vel", Twist, queue_size=1)
```

Si inserisce una funzione `main`, in cui si definisce una variabile `out_of_maze` inizializzata come `FALSE`. Si definisce, inoltre, un messaggio `velocity` di tipo `Twist`. Dopo quest'ultimo si crea un loop `while`, in cui, mentre il valore `out_of_maze` è false, si stampa il `self.b`, inoltre, se `self.b` è maggiore di 2, la velocità lineare in x è 0,5 m/s, altrimenti la velocità lineare in x è 0.

```
def main(self):
    self.out_of_maze = False
    self.vel = Twist()
    while not self.out_of_maze:
        print self.b
        if self.b > 2:
            self.vel.linear.x = 0.5
        else:
            self.vel.linear.x = 0

        self.robot_pub.publish(self.vel)
        self.rate.sleep()
```

Infine si pubblica il *vel_message* e si definisce il *rate.sleep*. In pratica, se non si ha davanti un ostacolo per 2 metri, si dice al robot di muoversi in avanti, altrimenti, se abbiamo un oggetto più vicino di due metri il robot si fermerà.

Per concludere bisogna modificare la riga *robot_object.read_laser()*:

```
if __name__ == '__main__':
    rospy.init_node('robot_test', anonymous=True)
    robot_object = Robot()
    try:
        robot_object.read_laser()

    except rospy.ROSInterruptException:
        pass
```

nella seguente maniera:

```
if __name__ == '__main__':
    rospy.init_node('robot_test', anonymous=True)
    robot_object = Robot()
    try:
        robot_object.main()
```

Con il comando *cd* si raggiunge la directory in cui sono presenti i file. Per rendere eseguibili i due codici in python, si utilizza il comando *chmod +x ./nomefile*.

Sono stati creati due file *launch*: *robot_laser.launch* e *robot_vel.launch* con i quali è stato possibile richiamare i due codici in python.

6. COSTRUZIONE DELLA MAPPA

La mappatura è una delle applicazioni elementari del robot mobile ed è il processo di creazione di una mappa leggibile sia dall'uomo che dal robot mobile. Queste mappe sono create utilizzando un robot mobile con sensori adeguati come sensori visivi (fotocamera), sonar e laser.

La mappatura è considerata uno dei contributi più interessanti del robot mobile poiché la mappa può essere utilizzata per varie applicazioni. Ad esempio, la navigazione autonoma di robot mobili che richiede la mappa dell'area per navigare in sicurezza. Un'altra applicazione include la possibilità di essere utilizzata in robot di ricerca e soccorso, il che richiede al robot di creare una mappa durante la ricerca e mappare la posizione della potenziale vittima. In quanto tale, la mappatura è diventata una delle ricerche importanti nell'area della robotica mobile con varie tecniche di mappatura attualmente disponibili.

6.1 Mappa in 2D

Uno dei metodi di mappatura più popolari è lo SLAM (*Simultaneous Localization and Mapping*), disponibile sulla piattaforma ROS, che consente di costruire o aggiornare la mappa mantenendo allo stesso tempo traccia della posizione del robot mobile. Esistono molti algoritmi disponibili in SLAM, ad esempio GMapping, Core SLAM, Graph SLAM e Hector SLAM. Ciascuno di questi algoritmi ha i propri parametri modificabili in base all'ambiente in cui il robot sta lavorando. L'utilizzo di parametri diversi, come la velocità del robot e il ritardo di aggiornamento della mappatura, non solo è in grado di causare cambiamenti nel tempo impiegato per la mappatura ma anche nell'accuratezza della mappa. Quindi, è importante conoscere l'effetto che i diversi parametri dell'algoritmo hanno sulla qualità della mappa.

È stato utilizzato il GMapping per la mappatura in 2D, in particolare *slam_gmapping* è un wrapper per la libreria SLAM di GMapping e legge gli scan dei laser e l'odometria per calcolare una mappa.

I parametri usati dal nostro wrapper GMapping sono:

- “*base_frame*”: il *frame_id* da utilizzare per la posa della base del robot;
- “*odom_frame*”: il *frame_id* da cui è letta l'odometria.

```

slam_gmapping.launch X
home > gianluca > catkin_ws > src > auto_tractor > launch > slam_gmapping.launch
1  <?xml version="1.0"?>
2
3  <launch>
4
5      <node name="slam_gmapping" pkg="gmapping" type="slam_gmapping">
6          <remap from="scan" to="laser_scan/scan"/>
7          <param name="base_frame" value="base_link"/>
8          <param name="odom_frame" value="odom"/>
9      </node>
10
11 </launch>

```

Il tag *remap* permette di passare argomenti di rimappatura del nome al nodo ROS che si sta avviando in modo più strutturato rispetto all'impostazione diretta dell'attributo *args* di un nodo.

Gli attributi sono:

- *from*="scan", che è il topic rimappato, cioè è il nome del ROS topic che si sta rimappando;
- *to*="laser_scan/scan", che è il target name, cioè il nome del ROS topic a cui si sta puntando dal topic from.

In altre parole, se si sta rimappando da un topic "scan" a un topic "laser_scan/scan", ogni volta che il nodo pensa di sottoscrivere sul topic "scan", in realtà sta sottoscrivendo sul topic "laser_scan/scan".

Per realizzare la mappa, bisogna avviare da terminale sia Gazebo che RViz. In un'altra finestra del terminale, utilizziamo il comando `roslaunch auto_tractor slam_gmapping.launch` per far iniziare la mappatura.

Su RViz si aggiungono i topic riguardanti la Camera, il LaserScan e la Map.

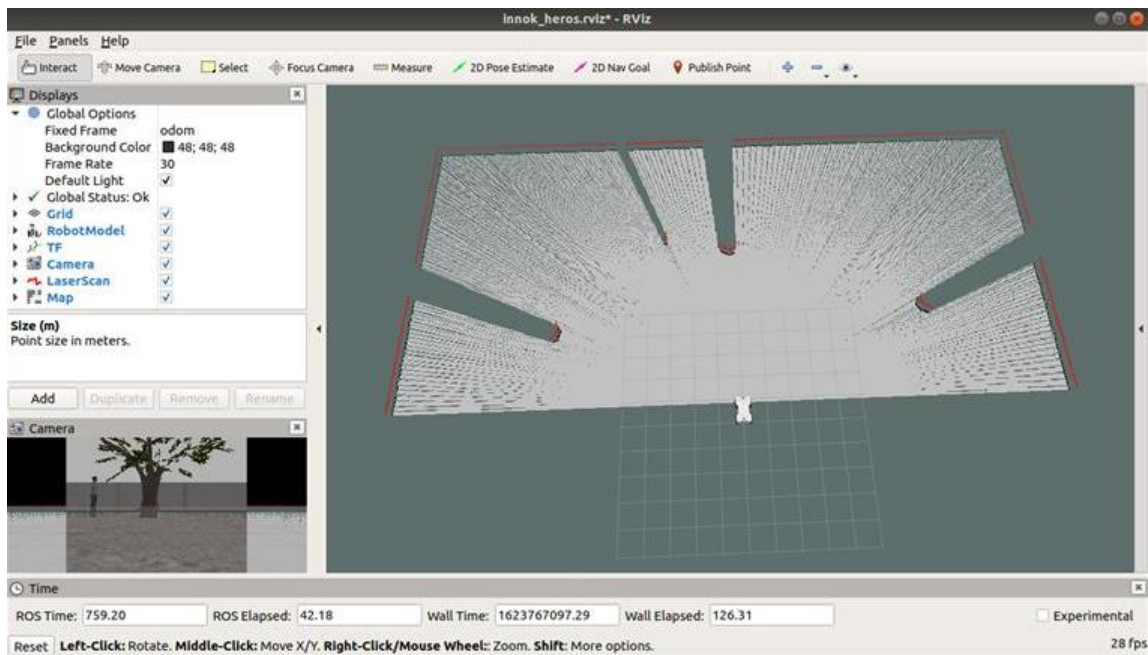


Figura 32: Vista mappa 2D su RViz

Con il file *teleop.launch* è possibile muovere il trattore da tastiera, così da esplorare l'ambiente e completare la mappa in 2D.

Per salvare la mappa appena realizzata, da terminale con l'utilizzo del comando *cd* ci si sposta nella cartella desiderata *map_provider* e si impiega il comando *roslaunch map_server map_saver -f mymap2d* grazie al quale si ottengono due file: *mymap2d.pgm* e *mymap2d.yaml*.

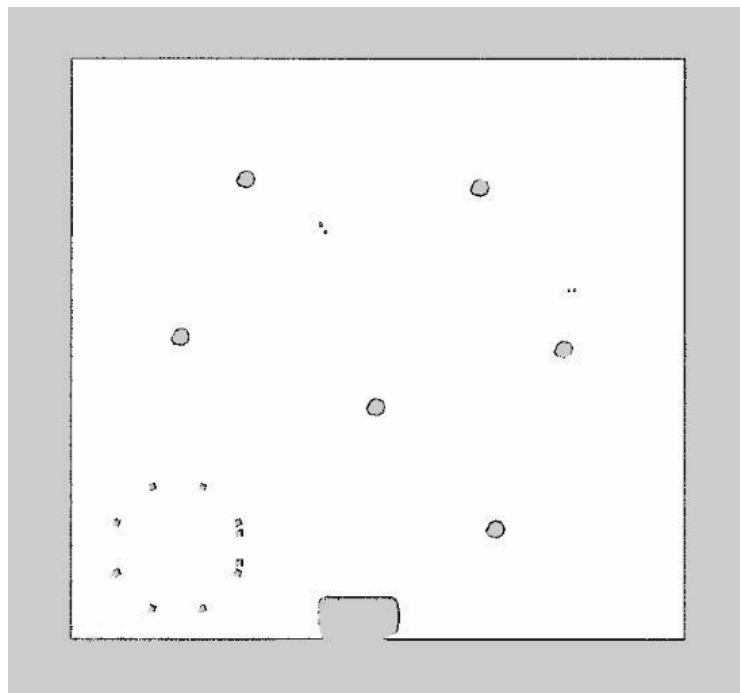


Figura 33: mymap2d.pgm

Nel file *.yaml* si possono notare i seguenti campi richiesti:

- *image*: percorso al file image contenente i dati di occupazione, può essere assoluto o relativo alla posizione del file YAML;
- *resolution*: risoluzione della mappa, espressa in metri/pixel;
- *origin*: la posa 2D del pixel in basso a sinistra nella mappa, come (x,y,yaw), dove yaw è la rotazione in senso antiorario (yaw=0 significa che non c'è rotazione);
- *negate*: indica se la semantica bianco/nero libero/occupato deve essere invertita;
- *occupied_thresh*: i pixel con probabilità di occupazione maggiore di questa soglia sono considerati completamente occupati;
- *free_thresh*: i pixel con probabilità di occupazione inferiore a questa soglia sono considerati completamente liberi.



```

! mymap2d.yaml x
home > gianluca > catkin_ws > src > map_provider > map > ! mymap2d.yaml
1  image: mymap2d.pgm
2  resolution: 0.050000
3  origin: [-100.000000, -100.000000, 0.000000]
4  negate: 0
5  occupied_thresh: 0.65
6  free_thresh: 0.196
7

```

All'interno della cartella *map_provider*, è stato creato un file *.launch*: *map_provider.launch* in cui viene richiamato il file *mymap2d.yaml* e che è utile per ricaricare la mappa su RViz. A tal fine è necessario riaprire sia Gazebo che RViz. Su un altro terminale si utilizza il comando *roslaunch map_provider map_provider.launch* e, aggiungendo il topic Map su RViz, si nota il seguente errore.

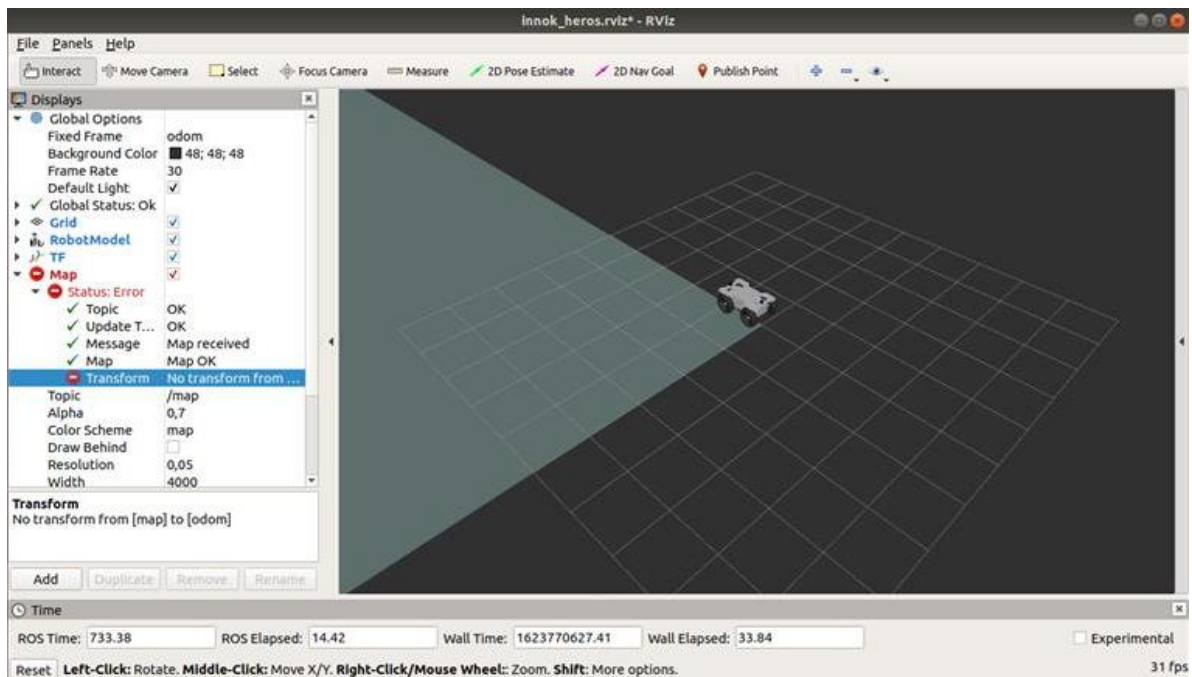


Figura 34: Errore nel topic Map

Per risolvere questo errore, è stato usato il seguente codice che include il file: *amcl.launch*.

```

map.launch x
home > gianluca > catkin_ws > src > auto_tractor > launch > map.launch
1  ?xml version="1.0" ?
2
3  <launch>
4    <arg name="map_file" value="$(find map_provider)/map/mymap2d.yaml"/>
5    <node pkg="map_server" type="map_server" name="map_server" args="$(arg map_file)"/>
6
7    <include file="$(find auto_tractor)/launch/amcl.launch" />
8
9  </launch>

```

In particolare, *amcl* è un sistema di localizzazione probabilistico per un robot che si muove in 2D. Implementa l'approccio di localizzazione adattivo Monte Carlo, che utilizza un filtro antiparticolato per tracciare la posa di un robot rispetto a una mappa nota, basata su laser e scansioni laser. *amcl* trasforma i messaggi e genera stime di posa. All'avvio, *amcl* inizializza il suo filtro antiparticolato in base ai parametri forniti.

amcl trasforma le scansioni laser in arrivo nel frame dell'odometria. Quindi deve esistere un percorso attraverso l'albero tf del frame in cui vengono pubblicate le scansioni laser al frame dell'odometria. Alla ricezione della prima scansione laser, *amcl* cerca la trasformazione tra il frame del laser e il frame di base e la blocca per sempre. Quindi *amcl* non può gestire un laser che si muove rispetto alla base. Le figure seguenti mostrano la differenza tra la localizzazione mediante odometria e *amcl*.

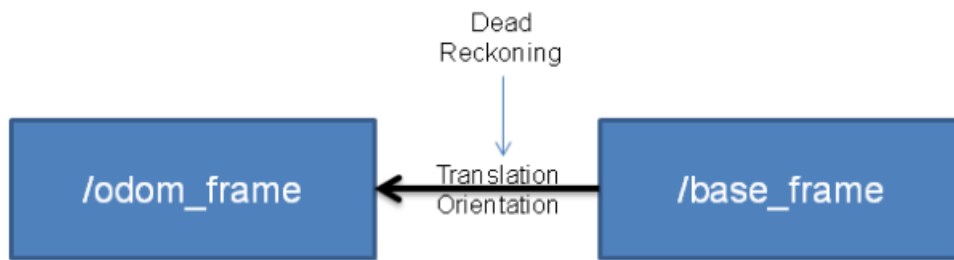


Figura 35: Odometry Localization

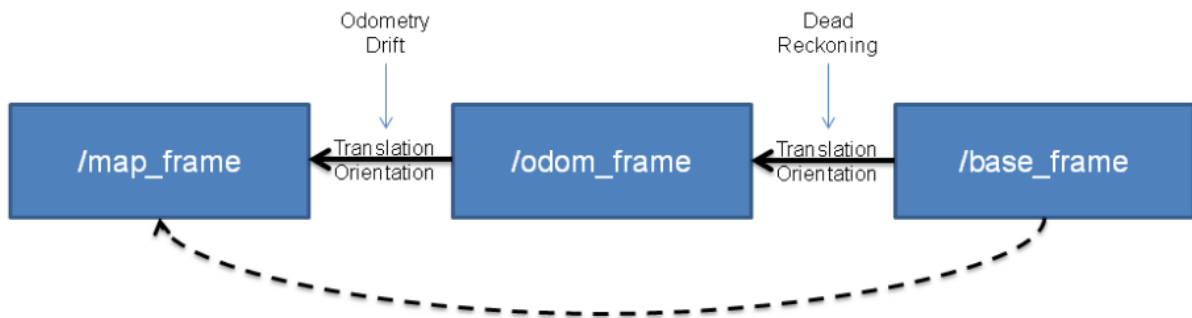


Figura 36: AMCL Map Localization

Durante l'operazione `amcl` stima la trasformazione del frame base rispetto al frame globale ma pubblica solo la trasformazione tra il frame globale e il frame odometry. In sostanza, questa trasformazione tiene conto della deriva che si verifica utilizzando Dead Reckoning.

Fatta questa correzione e avviando il codice appena descritto con il comando `roslaunch auto_tractor map.launch`, si può notare che l'errore è stato corretto e la mappa viene visualizzata con il trattore.

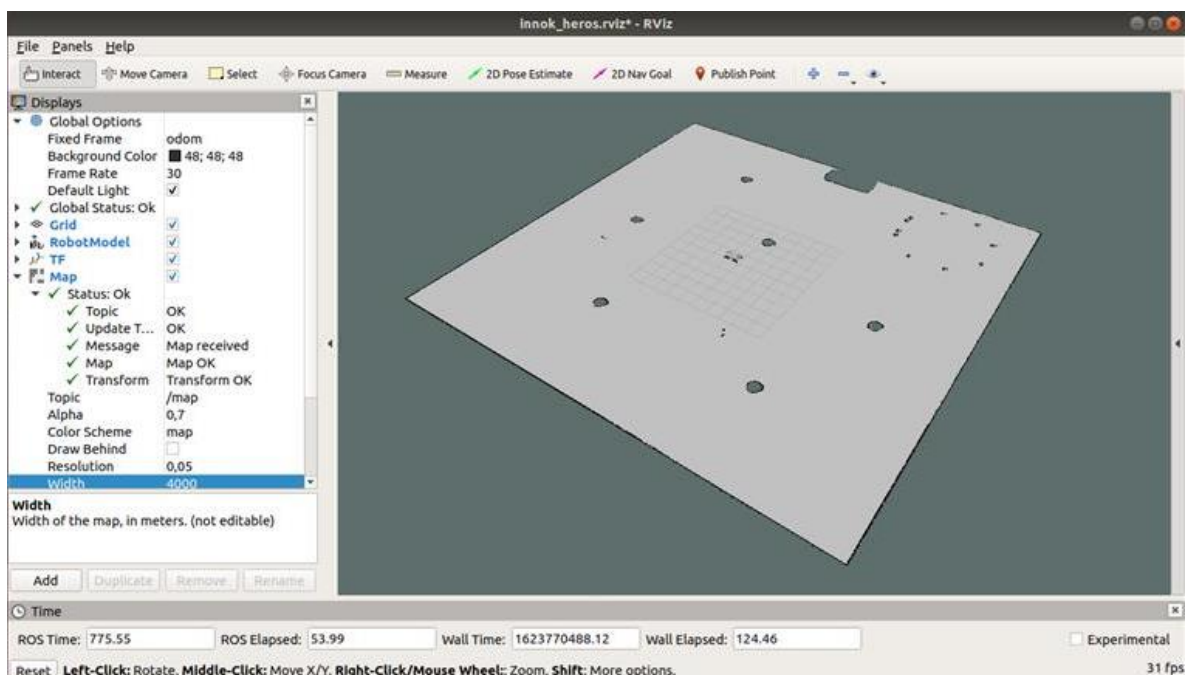


Figura 37: Mappa 2D e trattore

6.2 Mappa in 3D

Per realizzare la mappa in 3D è stata usata la libreria OctoMap. Essa implementa un approccio di mappatura della griglia di occupazione 3D, fornendo strutture dati e algoritmi di mappatura in C++ particolarmente adatti per la robotica. L'implementazione della mappa si basa su un octree ed è progettata per soddisfare i seguenti requisiti:

- **Modello 3D completo:** la mappa è in grado di modellare ambienti arbitrari senza preconcetti al riguardo. La rappresentazione modella sia le aree occupate sia lo spazio libero. Le aree sconosciute dell'ambiente sono implicitamente codificate nella mappa. Mentre la distinzione tra spazio libero e occupato è essenziale per la navigazione sicura del robot, le informazioni su aree sconosciute sono importanti, ad esempio, per l'esplorazione di un ambiente.
- **Aggiornabile:** è possibile aggiungere nuove informazioni o letture del sensore in qualsiasi momento. La modellazione e l'aggiornamento vengono eseguiti in modo probabilistico. Ciò tiene conto del rumore del sensore o delle misurazioni che risultano da cambiamenti dinamici nell'ambiente, ad esempio a causa di oggetti dinamici. Inoltre, più robot sono in grado di contribuire alla stessa mappa e una mappa registrata in precedenza è estensibile quando vengono esplorate nuove aree.
- **Flessibile:** l'estensione della mappa non deve essere conosciuta in anticipo. Invece, la mappa viene espansa dinamicamente secondo necessità. La mappa è multi-risoluzione in modo che, ad esempio, un pianificatore di alto livello sia in grado di utilizzare una mappa grossolana, mentre un pianificatore locale può operare utilizzando una risoluzione fine. Ciò consente anche visualizzazioni efficienti che vanno da panoramiche approssimative a viste ravvicinate dettagliate.
- **Compatta:** la mappa viene archiviata in modo efficiente, sia in memoria che su disco. È possibile generare file compressi per un utilizzo successivo o un comodo scambio tra robot anche con vincoli di larghezza di banda.

È stato creato il file *octomap.launch*, riportato e analizzato.

```

octomap.launch u x
home > gianluca > catkin_ws > src > auto_tractor > launch > octomap.launch
1  <?xml version="1.0"?>
2
3  <launch>
4    <node pkg="octomap_server" type="octomap_server_node" name="octomap_server">
5      <param name="resolution" value="0.05" />
6
7      <param name="frame_id" type="string" value="odom" />
8
9      <!-- maximum range to integrate (speedup!) -->
10     <param name="sensor_model/max_range" value="5.0" />
11
12     <!-- data source to integrate (PointCloud2) -->
13     <remap from="cloud_in" to="/depth_camera/depth/points" />
14
15   </node>
16 </launch>

```

octomap_server costruisce e distribuisce mappe di occupazione 3D volumetriche come stream binario OctoMap e in vari formati compatibili con ROS, ad esempio per evitare o visualizzare ostacoli. La mappa può essere un file OctoMap statico .bt (come argomento della riga di comando) o può essere creata in modo incrementale dai dati in entrata (come PointCloud2). Octomap_server inizia con una mappa vuota se non viene fornito alcun argomento della riga di comando. In generale, octomap_server crea e pubblica solo su argomenti che sono sottoscritti.

I parametri sono:

- *resolution*: risoluzione in metri per la mappa quando si inizia con una mappa vuota, altrimenti viene utilizzata la risoluzione del file caricato;
- *frame_id*: frame globale statico in cui verrà pubblicata la mappa, quando si costruiscono le mappe in modo dinamico, deve essere disponibile una trasformazione dai dati del sensore a questo frame;
- *sensor_model/max_range*: intervallo massimo in metri per l'inserimento dei dati della nuvola di punti durante la creazione dinamica di una mappa, limitare la portata in maniera utile (ad esempio 5 m) impedisce punti errati spuri lontani dal robot.

Nel codice, *cloud_in* rappresenta la nuvola di punti 3D in arrivo per l'integrazione della scansione. È necessario rimappare questo argomento sui dati del sensore e fornire una trasformazione tf tra i dati del sensore e la cornice della mappa statica. Il *frame_id* della nuvola di punti deve essere il frame del sensore.

È possibile avviare il file con il comando `roslaunch auto_tractor octomap.launch`. Su RViz si aggiunge il MarkerArray con il topic che viene pubblicato da *octomap_server*, ovvero *occupied_cells_vis_array*.

Si ottiene il seguente risultato.

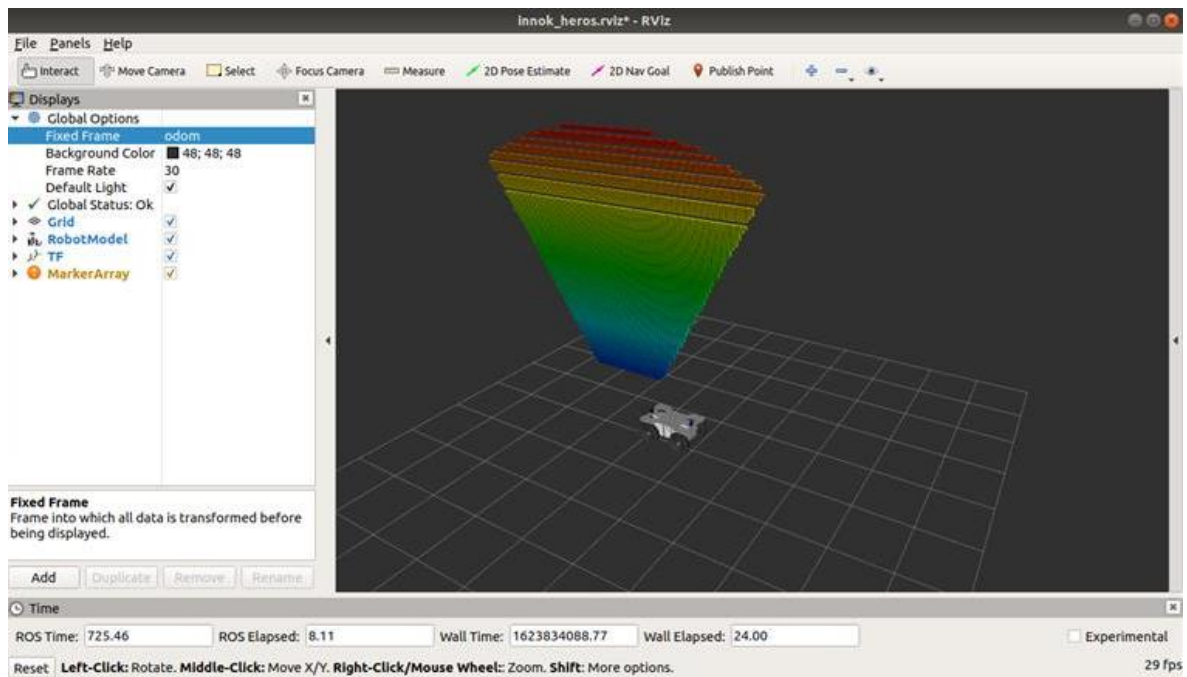


Figura 38: Posa della nuvola dei punti sbagliata

La posa della nuvola dei punti è sbagliata, dato che fluttua nell'aria. Si tratta di un bug di Gazebo, in quanto la posa tf del frame di Gazebo è prevista solo per il topic dell'immagine, non per il topic della nuvola dei punti.

Per risolvere il bug, nell'URDF del trattore, è stato aggiunto un collegamento vuoto (*camera_link_fake*) e un giunto di regolazione (*camera_joint_fake*) per regolare la posa dell'immagine della nuvola dei punti.

```
<link name="camera_link_fake"></link>
<joint name="camera_joint_fake" type="fixed">
  <origin xyz="0 0 0" rpy="-1.5708 0 -1.5708 "/>
  <parent link="depth_camera"/>
  <child link="camera_link_fake"/>
</joint>-->
```

È stato, inoltre, modificato il frame name della camera con *camera_link_fake*.

Come si vede dall'immagine successiva, il bug è stato risolto.

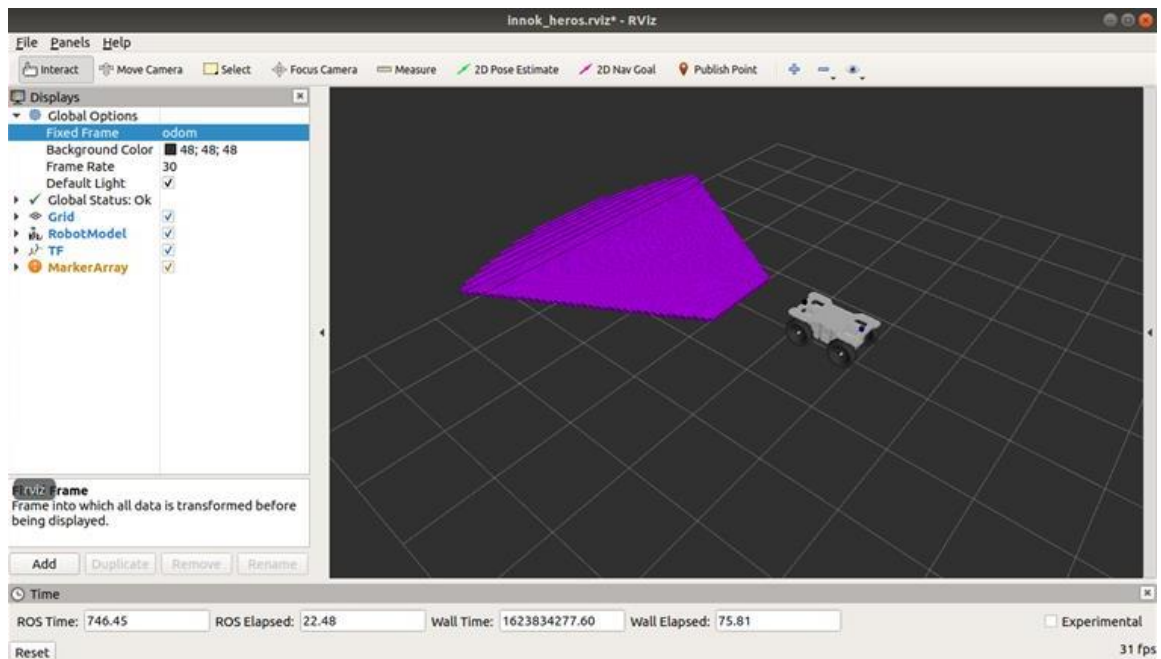


Figura 39: Bug risolto

Possiamo muovere il trattore e iniziare a far raccogliere i dati per la costruzione della mappa.

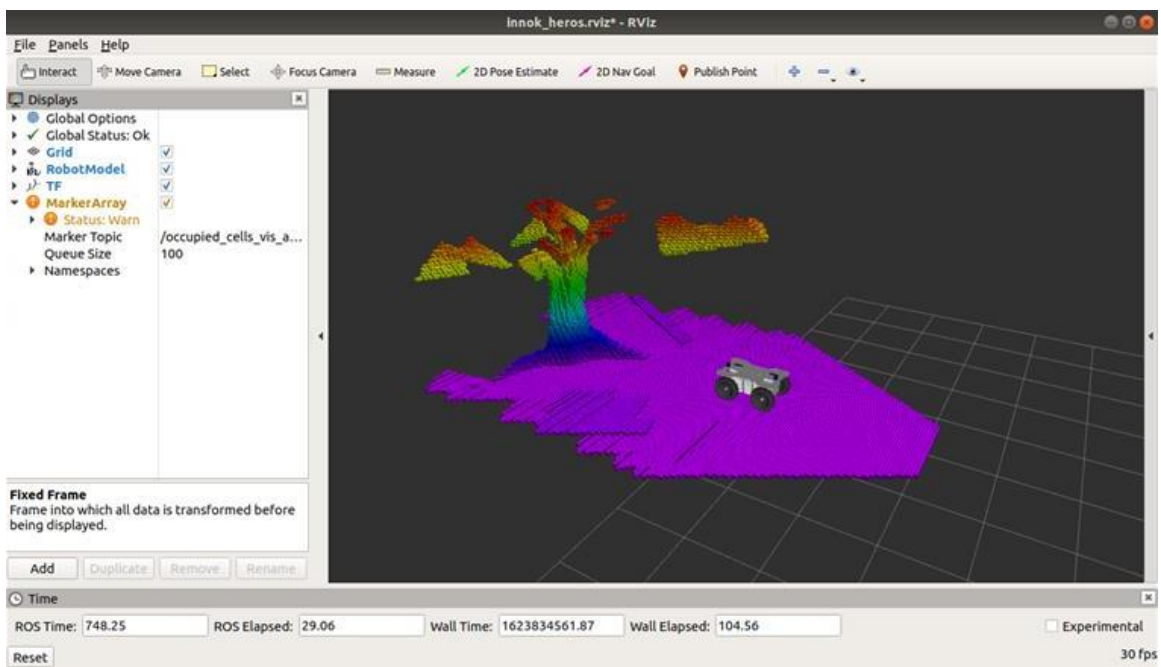


Figura 40: Inizio esplorazione dell'ambiente

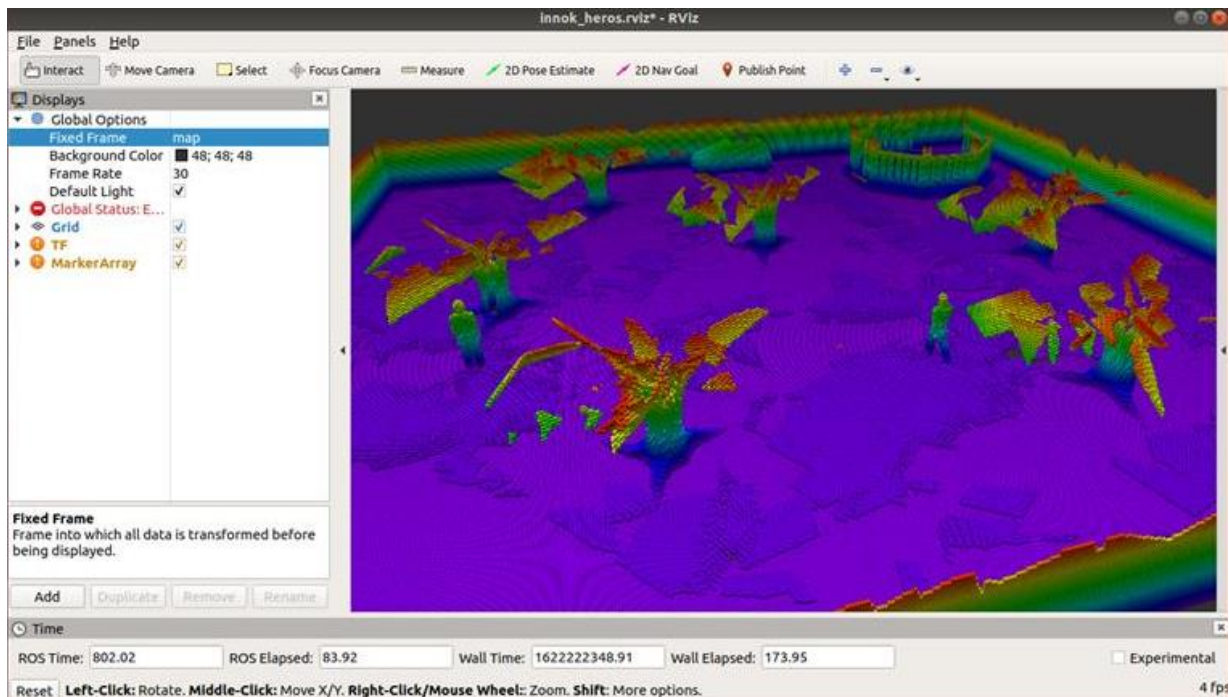


Figura 41: Mappa 3D completa

Una volta completata la mappa, è possibile salvarla spostandosi sempre nella cartella desiderata con il comando `cd`, utilizzando il comando `roslaunch octomap_server octomap_saver -f map3d.bt`, in cui `octomap_saver` è uno strumento da riga di comando per richiedere file octomap da un `octomap_server`.

Per ricaricare la mappa su RViz si utilizza il comando `roslaunch octomap_server octomap_server_node map3d.bt`. Per visualizzare la mappa, è utile modificare il `fixed frame` con `map` e aggiungere il `MarkerArray` nuovamente con il topic `occupied_cells_vis_array`.

7. CONCLUSIONI E LAVORO FUTURO

Per questo lavoro è stato utilizzato il Robotic Operating System per simulare un trattore controllato da tastiera (in anello aperto), in grado di muoversi in un ambiente realizzato con Gazebo costituito da alberi e altri ostacoli fissi al fine di emulare l'ambiente di campagna.

Tramite Visual Studio sono stati scritti i codici in cui si sono stati specificati joint e link per la costruzione e il controllo del trattore, che è possibile visualizzare su RViz e su Gazebo grazie agli opportuni comandi su terminale.

Si sono installati poi sensori di vario tipo, come laser, camere, ecc., per i seguenti scopi:

- costruzione di una mappa in 2D, con algoritmi di SLAM;
- costruzione di una mappa in 3D, con pacchetto *Octomap*;
- analisi dei dati raccolti da sensore laser per osservare la vicinanza del trattore rispetto agli ostacoli in tempo reale e per bloccare l'avanzata del trattore stesso in una certa direzione nel momento in cui l'ostacolo risultasse più vicino di una determinata quantità.

Entrambe le mappe sono state salvate e possono essere visualizzate in simulazioni successive.

7.1 Sviluppi futuri

Come sviluppo futuro si può passare alla pianificazione di un path, ovvero di un percorso, da far seguire al trattore. Il path viene generato a partire da mission commands, cioè i comandi che il robot deve seguire. Nel caso di path planning si considerano missioni in cui si vuole che il robot vada da un punto iniziale in una configurazione iniziale fino a un punto finale in una configurazione voluta.

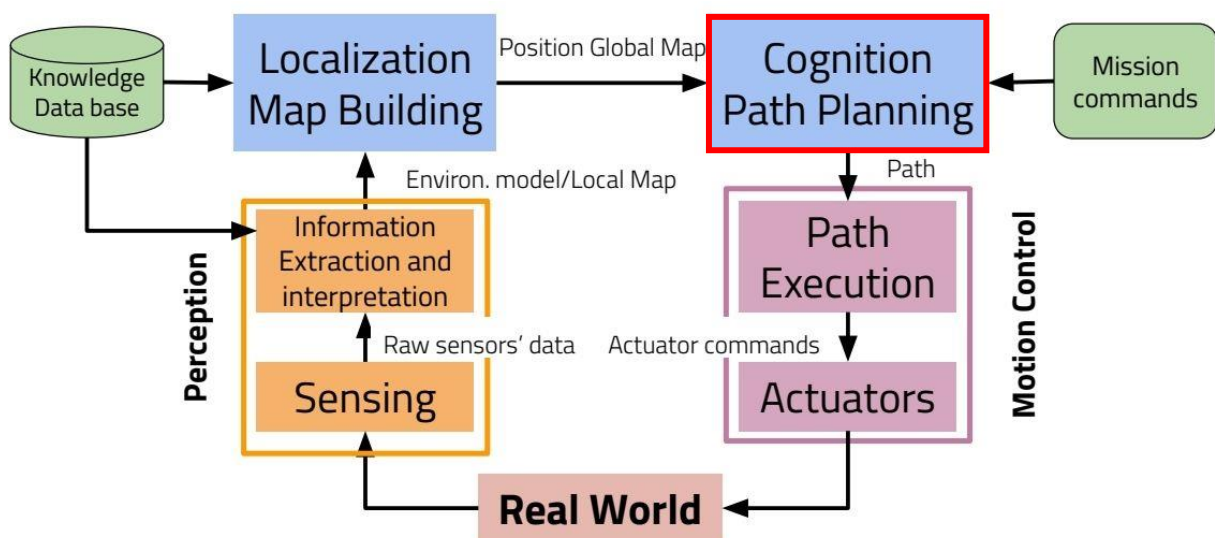


Figura 42: Loop di controllo con blocco evidenziato

A partire dalla mappa globale si può generare un path usato per gli algoritmi di motion control.

Il trattore si muoverà nell'ambiente di lavoro, anche detto *workspace*, in cui sono presenti gli ostacoli già definiti. Obiettivo del path planning sarà quello di pianificare un percorso affinché il trattore eviti collisioni.

Tale pianificazione potrà essere del tipo *offline planning* essendo l'ambiente già interamente conosciuto e analizzato.

Altro interessante caso di studio può riguardare l'obstacle avoidance in cui, data una traiettoria, venga identificata la presenza di un ostacolo ed esso venga evitato posto che il target venga sempre raggiunto. L'obstacle avoidance presuppone che, durante il moto, un ostacolo non previsto possa trovarsi davanti al robot; in tal caso occorre:

- identificare la presenza dell'ostacolo, operazione che si effettua tramite opportuni sensori;
- evitare l'ostacolo, operazione che comporta l'uso di opportuni algoritmi.

8. RIFERIMENTI

- [1] Slide di Mobile Robotics A.A. 2019/2020
- [2] http://wiki.ros.org/innok_heros_gazebo/Tutorials/Simulating%20Innok%20Heros
- [3] <http://wiki.ros.org/urdf>
- [4] <http://wiki.ros.org/xacro>
- [5] <https://www.stereolabs.com/docs/ros/rviz/>
- [6] http://gazebosim.org/tutorials?tut=ros_overview
- [7] http://wiki.ros.org/twist_mux
- [8] <https://blog.stackpath.com/yaml/>
- [9] https://web.media.mit.edu/~achoo/tr/3d_benefits_limits.pdf
- [10] <https://www.fotografareperstupire.com/sensore-ccd-sensore-cmos/>
- [11] <https://iopscience.iop.org/article/10.1088/1757-899X/705/1/012037/pdf>
- [12] <http://wiki.ros.org/amcl>
- [13] <https://octomap.github.io/>
- [14] http://wiki.ros.org/octomap_server
- [15] <https://www.programmersought.com/article/32506063801/>