Robótica Móvil

Proyecto Final Integrador

Simulación de Robots de Batalla

Cavallo Gastón
Cogo Martín
Fernandez Juan Manuel
Flores Emiliano
Gimenez Federico
Pérez Eduardo
Pérez Carlos
Serrano Cristian
Morales Juan Martín

31 de mayo de 2024

Índice

1. Introducción

1.1. Propuesta

Como proyecto final integrador de la materia Robótica Móvil, hemos propuesto la elaboración de dos entornos de simulación 3D, dos robots de diseño propio y la integración de módulos para su navegación utilizando los conceptos aprendidos durante el cursado de la materia.

Para llevar a cabo la implementación de este proyecto se utilizó *Robot Operating System 2* (ROS 2 Humble distribution) el cual brindó el marco de trabajo para el desarrollo de software para los robots. La implementación de simulaciones se llevó a cabo con el simulador **Gazebo** que se encuentra integrado con **ROS 2**, una herramienta para simulación de entornos virtuales y algoritmos de robótica.

Además se realizó la integración correspondientes al marco de trabajo propuesto por **ROS 2** que controla los robots generados e integra con módulos como **Rviz2** para visualización, y **SLAM** para navegación, entre otros. También, hemos incorporado la posibilidad de teleoperar cada robot utilizando teclado o un *joystick* basándonos en los paquetes *teleop_twist_keyboard* y *teleop_twist_joy*.

En las siguientes secciones de este informe, profundizaremos en como se llevo acabo la aplicación de los módulos del proyecto y la integración de los paquetes provistos por **ROS 2** y su relación con los conceptos abarcados en la materia.

1.2. Motivación

El proyecto de Robótica Móvil se planteó como un desafío integrador final de la materia, con el objetivo de aplicar y consolidar los conocimientos adquiridos durante el curso. La motivación principal radica en la necesidad de desarrollar habilidades prácticas en el diseño, simulación y control de robots, utilizando tecnologías y herramientas modernas como ROS 2 y Gazebo. Este proyecto también buscaba fomentar el trabajo colaborativo y la resolución de problemas complejos en el ámbito de la robótica, preparán-

donos para enfrentar retos similares en nuestras futuras carreras profesionales.

2. Planificación y Herramientas

En esta sección, primero proporcionamos una explicación sobre cómo se llevó a cabo la división de tareas, como se detalla en la sección 2.1. Posteriormente, se abordan diversas herramientas y frameworks utilizados durante la implementación, las cuales se describen en las secciones 2.2, 2.3 y 2.4.

2.1. División del trabajo en equipos

En este trabajo hemos dividido las tareas de implementación de la siguiente manera donde cada integrante ha aportado contenido original al proyecto:

- Distribución de tareas:
  - **Juan Manuel Fernández**
- Integración y prueba de teleoperación de robots en simultáneo:
  - **Pérez Eduardo**
  - **Pérez Carlos**
- Elaboración de robots propios:
  - **Juan Martín Morales** creación de robot base, robot Don Barredora, LiDAR e integración inicial con differential_drive.
  - **Cogo Martín** Implementación del robot AxeBot con partes personalizadas.
- Elaboración de entornos de simulación:
  - **Juan Manuel Fernández**
- Implementación de navegación:
  - **Gastón Cavallo**
  - **Cristian Serrano**
- Implementación de teleoperador con joystick:
  - **Emiliano Flores**

    - **Federico Gimenez**
- Dockerización de entorno de desarrollo:
    - **Gastón Cavallo**

2.2. Repositorio de Github

Dado que este proyecto fue desarrollado de manera colaborativa entre varios miembros, hemos optado por crear un repositorio en GitHub: proyecto_robotica_ws. Con el fin de mantener una estructura organizada, hemos dividido el trabajo en las siguientes ramas principales:

- **main**: Esta rama se utilizó como base para la creación de otras ramas principales.
- **multibots**: Aquí se encuentra todo el contenido relacionado con el despliegue de dos robots en un entorno simulado, utilizando dos joysticks.
- **navigation**: Esta rama abarca todo el contenido relacionado con la navegación de un solo robot.

Para garantizar un trabajo organizado y coordinado, hemos establecido las siguientes convenciones:

1. **feature/**: Todas las ramas que no sean principales en el repositorio deben llevar este prefijo en su nombre.
2. **Pull Request**: En caso de que se requieran cambios importantes, se crea una ”pull request”para revisar qué modificaciones se realizarán y, si es necesario, corregirlas antes de su integración.

2.3. Implementación del contenedor de Docker

ChatGPT Durante la realización del trabajo, nos enfrentamos a diversas dificultades al intentar sincronizar las versiones de ROS, Gazebo y las distintas distribuciones de Linux con sus respectivos paquetes.

Por consiguiente, optamos por dockerizar el entorno de desarrollo, lo que nos permite crear un entorno “reproducible” y “temporal” en el que se instalan únicamente las versiones del software necesarias para el desarrollo.

En primer lugar, decidimos trabajar con la versión de Ubuntu 22.04 LTS, la cual está respaldada por la versión de ROS2 “Humble LTS”. Esta última es la versión más moderna de ROS con soporte a largo plazo. Esta elección se hizo con el objetivo de aprovechar al máximo las nuevas funcionalidades de ROS2 y evitar incompatibilidades con versiones más recientes de Ubuntu.

Para llevar a cabo el proceso de dockerización en estas versiones, creamos un archivo llamado ”`Dockerfile`.[en] la carpeta raíz del proyecto. Este archivo describe, en forma de capas, todo el software necesario dentro del entorno de trabajo.

Además, en dicho archivo se deben incluir todas las instrucciones pertinentes para permitir el uso de la GPU por parte de Docker, ya que gran parte del software ejecuta código orientado a la aceleración por hardware para gráficos 3D.

En este punto, ya estamos listos para realizar un “build” de nuestro contenedor, esto se lleva a cabo mediante el comando:

```
docker build -t r2_boxbots .
```

Una vez compilada la imagen de Docker, estamos listos para ejecutar el entorno. Dado que Docker no expone una interfaz gráfica de forma predeterminada, es necesario establecer una conexión entre el servidor de ventanas del sistema operativo anfitrión y el contenedor. Esto se logra mediante la creación de un volumen que replica los archivos de X11 (servidor de ventanas de Linux) del sistema operativo anfitrión y los hace disponibles para que el contenedor Docker se comunique con el usuario utilizando el mismo.

De esta manera, al ejecutar nuestro contenedor, todas las aplicaciones con interfaces gráficas se mostrarán en nuestra pantalla como si se ejecutaran de forma nativa. Para ejecutar el entorno dockerizado, podemos utilizar el archivo `run_docker_gpu.bash`.

Es importante tener instalados Docker y xhost para poder ejecutar la visualización de Gazebo.

```
./run_docker_gpu.bash
```

Una vez ejecutado el comando anterior, para ingresar al contenedor creado se debe ejecutar el siguiente comando en la terminal:

```
docker exec -it r2_boxbots_container bash
```

Entendiendo la lógica, podemos observar en el fragmento de código del cuadro 1 las diferentes opciones que se pueden añadir al comando `docker run`, junto con un comentario que explica el propósito de cada opción.

```
docker run -it \ # Ejecuta el contenedor en modo interactivo
  --name=r2_test_container \ # nombre del contenedor
  --env="DISPLAY=$DISPLAY" \ # le indica que el display de
     ↪ salida es el display de la maquina host
  --env="QT_X11_NO_MITSHM=1" \ # le indica que no se use el
     ↪ MIT-SHM extension
  --volume="/tmp/.X11-unix:/tmp/.X11-unix:rw" \ # monta el
     ↪ directorio de salida de la maquina host en el
     ↪ contenedor (para grafica)
  --volume="${PWD}:/root/project" \ # monta el directorio
     ↪ actual en el contenedor
  --env="XAUTHORITY=$XAUTH" \ # le indica que el archivo de
     ↪ autorización es el de la maquina host
  --volume="$XAUTH:$XAUTH" \ # monta el archivo de
     ↪ autorización en el contenedor
  --net=host \ # le indica que use la red de la maquina host
  --privileged \ # le da privilegios al contenedor
  project_test \ # nombre de la imagen
  bash # comando a ejecutar
```

Cuadro 1: Lista de banderas que se pueden agregar al comando `docker run` y un comentario explicando el propósito de cada una.

2.3.1. Utilización en windows

En Windows, para poder visualizar Gazebo, es necesario instalar la herramienta VcXsrv. Después de instalarla, es necesario configurarla. Una vez instalada, abra el programa y configure las opciones como se muestra en la figura 1, donde el número de pantalla 0 es el que utilizaremos posteriormente para proporcionar salida de video al contenedor.

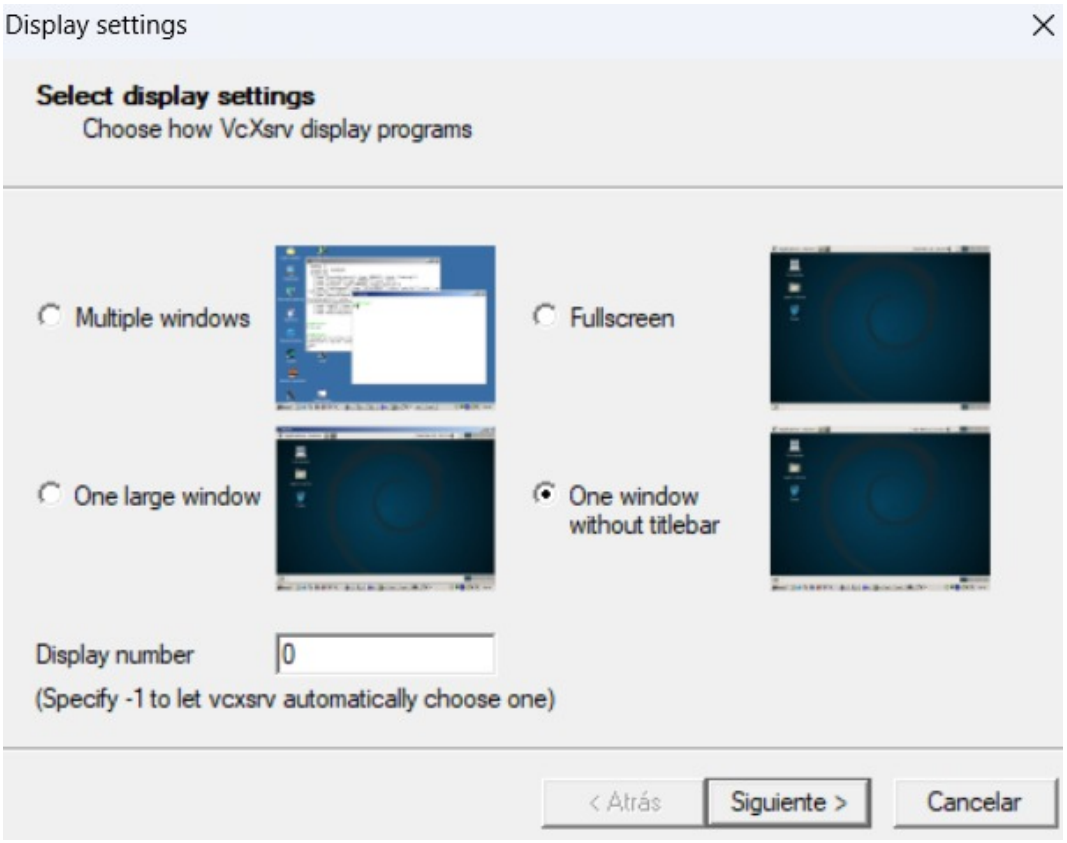

Figura 1: Pantalla de configuración de Xsrv.

Construcción Para construir la imagen de Docker se puede utilizar el comando:

```
docker build -t r2_boxbots .
```

Ejecución Para correr el entorno dockerizado se puede utilizar el archivo run_docker_gpu.bat.

```
run_docker.bat
```

Una vez ejecutado el comando anterior, para ingresar al contenedor creado se debe ejecutar el siguiente comando en la terminal:

```
docker exec -it r2_boxbots_container bash
```

En el cuadro 2, se brinda una explicación del comando a detalle.

```
docker run -it \
  --name=r2_boxbots_container \ # nombre del contenedor
  -e DISPLAY=host.docker.internal:0.0 \ # le indica que el
     ↪ display de salida es el display de la maquina host
  -e LIBGL_ALWAYS_INDIRECT=0 \ # le indica que no use el MIT-
     ↪ SHM extension
  --volume="${PWD}/../:/root/project" \ # monta el directorio
     ↪ actual en el contenedor
  r2_boxbots \ # nombre de la imagen
  bash # comando a ejecutar
```

Cuadro 2: Banderas que puede tener el comando `docker run` en Windows

2.4. Conceptos teóricos de ROS 2

ROS 2 es una evolución del conocido ROS, diseñado para superar las limitaciones de su predecesor mediante el soporte de sistemas en tiempo real, mejor interoperabilidad y escalabilidad.

2.4.1. Arquitectura

La arquitectura de **ROS 2** se basa en una infraestructura de nodos que se comunican entre sí mediante mensajes, servicios y acciones. A lo largo del desarrollo de nuestro proyecto, hemos aprovechado las características como gestión de nodos y namespaces, así como la integración con una variedad de sensores y actuadores, lo cual nos ha permitido organizar eficientemente nodos y recursos, especialmente al utilizar múltiples robots en un mismo escenario de simulación.

2.4.2. Paquetes

- **ros-humble-xacro**: Utilizado para generar archivos URDF a partir de macros XML.
- **ros-humble-gazebo-ros-pkgs**: Paquetes necesarios para la integración de Gazebo con ROS2.

- **ros-humble-twist-mux**: Utilizado para la multiplexación de comandos de velocidad.
- **ros-humble-ros2-control**: Paquete de control para gestionar los actuadores del robot.
- **ros-humble-ros2-controllers**: Controladores específicos para diferentes tipos de actuadores.
- **ros-humble-gazebo-ros2-control**: Integración de controladores ROS 2 con Gazebo.

2.4.3. Funcionalidades

- **gazebo**: Simulador utilizado para probar y validar el comportamiento del robot en un entorno virtual antes de implementarlo en el hardware real.
- **ros2 launch**: Herramienta utilizada para lanzar múltiples nodos y configurar sus parámetros de manera simultánea, simplificando el proceso de inicio de aplicaciones complejas.
- **rviz2**: Herramienta de visualización que permite observar los datos de sensores y el estado del robot en tiempo real.
- **tf2**: Utilizado para la gestión de transformaciones espaciales, esencial para el seguimiento preciso de la posición y orientación de los robots.
- **nav2**: Este paquete es utilizado para la navegación y planificación de rutas, permitiendo a los robots moverse autónoma-mente en el entorno.
- **ros2 bag**: Herramienta para grabar y reproducir datos de sensores, facilitando la depuración y análisis post-misión.

3. Diseño e Implementación de los Robots

En esta sección se explica el diseño propio de los robots y su implementación propuestos en el proyecto.

3.1. Basado en Turtlebot 3

Los robots implementados en este proyecto, han basado su locomoción en la del turtlebot3. Un modelo ampliamente utilizado en el campo de la robótica educativa y de investigación. Siendo además el robot de referencia otorgado por la cátedra.

3.1.1. Características

- **Movilidad**: Este robot utiliza un sistema de tracción diferencial con dos ruedas motorizadas y una rueda esférica de apoyo en la parte trasera para equilibrio.
- **Grados de libertad**: Las ruedas motorizadas le permiten realizar giros en el lugar. Posee dos grados de libertad principales que corresponden con sus dos ruedas motorizadas.
- **Sensores**:
  - **LiDAR** (Light Detection and Ranging): Utiliza un sensor LiDAR 360° para mapeo y navegación autónoma. El modelo típico utilizado es el LDS-01.
  - **Odómetro**: Los motores están equipados con encoders para medir la rotación de las ruedas y calcular la posición del robot.

3.2. Geometría y configuración de ruedas

Para este proyecto se plantearon dos diseños e implementaciones diferentes para cada robot, por lo que poseen dimensiones y características distintas.

- 3.3**Don Barredora**: Es un robot que utiliza figuras geométricas básicas para la construcción de sus ruedas y chasis, pero tienen una parte especifica que cumple la función de pala en la parte frontal, que fue diseñado con un programa de diseño 3D llamado Blender.
- 3.4**AxeBot**: Este robot tienen todas sus partes creadas con un programa de diseño externo llamado Zbrush. Tiene una forma orgánica y cuenta con un hacha en la parte frontal.

3.3. Don Barredora

Este robot se encuentra divido en dos partes fundamentales que permiten la implementación de nuevos robots utilizándolo como base. Primero explicaremos como se implementó la parte base 3.3.2 y luego las partes añadidas para crear robot Don Barredora 3.3.3.

3.3.1. URDF para definición de robots

Para describir las características físicas del robot en ROS se utiliza una descripción contenida en un archivo URDF (Unified Robot Description Format). El archivo URDF describe al robot como un árbol de *links*, conectados por *joints*. Los *links* representan los componentes físicos del robot, como ruedas, chasis, sensores, etc. Los *joints* representan la unión entre dos *links* y como se moverán relativamente entre ellos.

3.3.2. Base

Toda la implementación del robot base se encuentra en un archivo URDF denominado “Robot Core”. Las dimensiones del chasis del robot base son 60cm x 40cm x 20cm (dx, dy, dz).

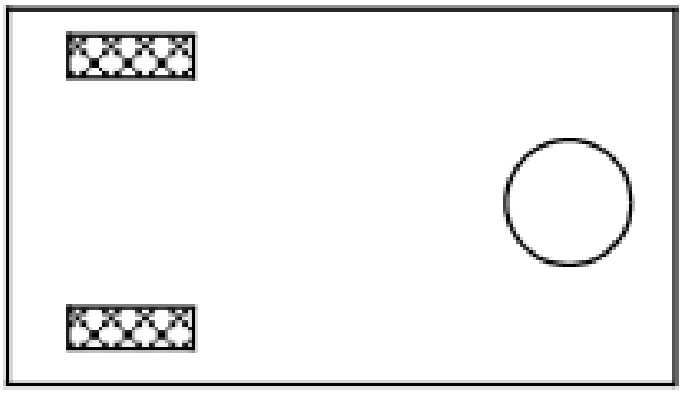

Figura 2: Representación abstracta de la relación entre las ruedas del robot. Los rectángulos representan las 2 ruedas y el circulo la rueda esférica

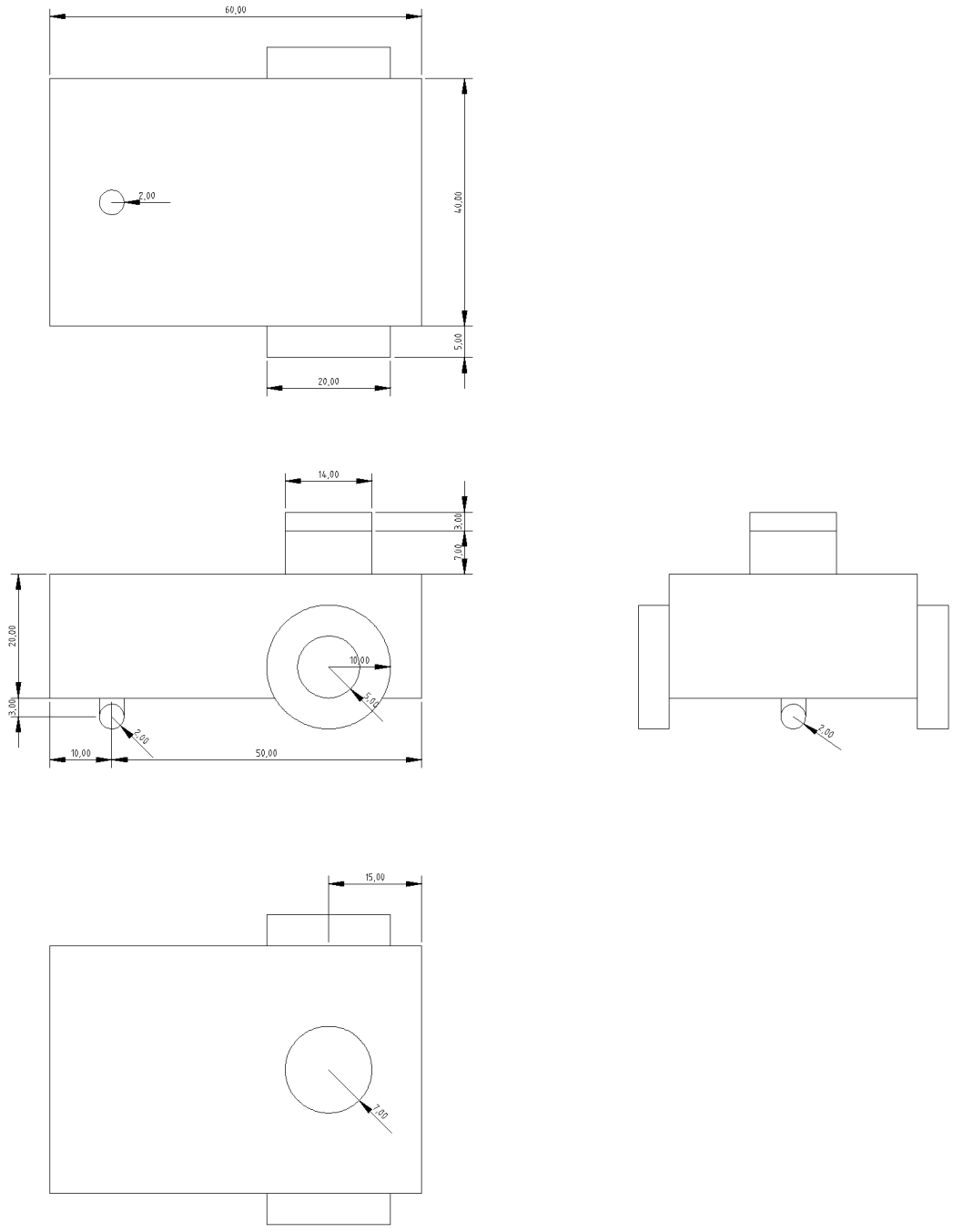

Figura 3: Robot desde distintas perspectivas en un sistema diédrico y sus componentes: **chasis**, con forma de caja; **ruedas estándar**, dos unidades; **rueda omnidireccional esférica**; **soporte para el LiDAR**; **sensor LiDAR**.

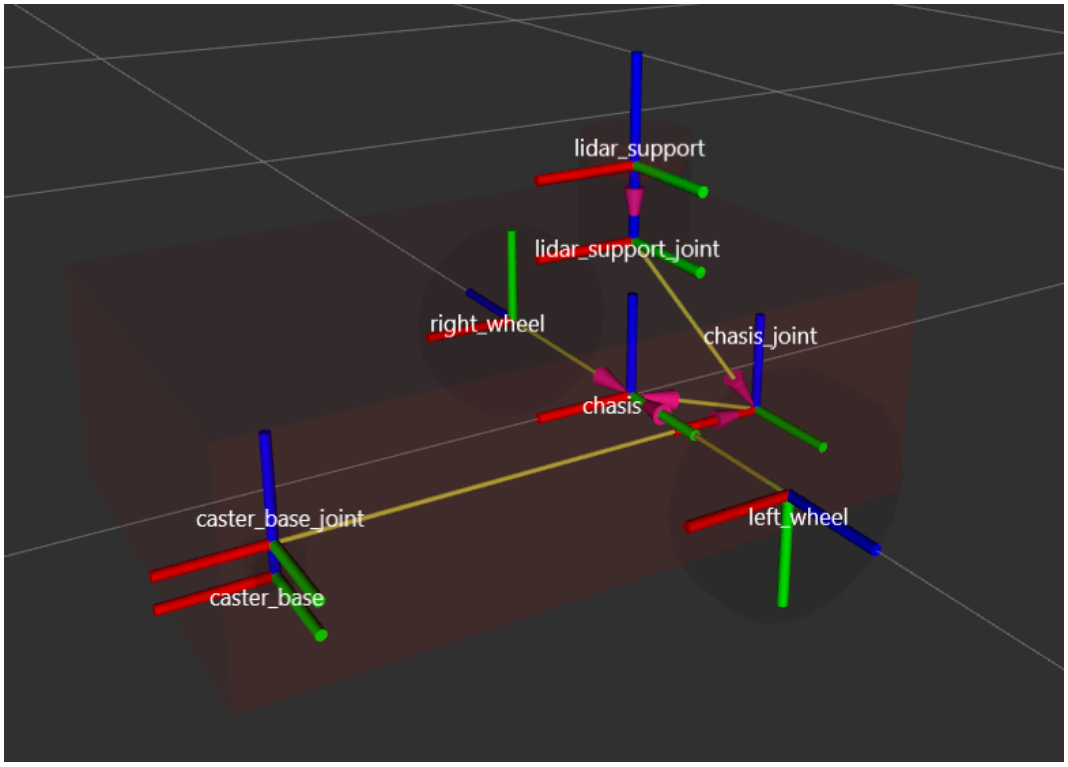

Figura 4: Un espacio 3D lanzado con **Rviz2** que muestra los ejes de referencia de cada parte y los vectores con flecha rosada apuntan hacia el sistema de coordenadas con el cual se referencia.

Adicionalmente, se definieron las propiedades de colisión, correspondientes a la geometría, y las propiedades físicas. Cada *link* a simular requiere una tag “inertial” el cual contiene información de:

- La masa (en kg) del componente.
- La matriz de inercia rotacional. Calculada, para cada *link* con una forma geométrica primitiva, usando este listado de momentos de inercia.
- A la esfera de apoyo se le agregaron coeficientes de contacto $mu1$ y $mu2$ cercanos a 0, para simular una rueda esférica.

A partir del diseño planteado en la figura 3, se calcularon todos los orígenes y dimensiones de los *links*, junto a sus *joints*. El árbol de *links*, en la figura 5, se puede observar mediante el programa *view_frame* del paquete *tf2_tools*.

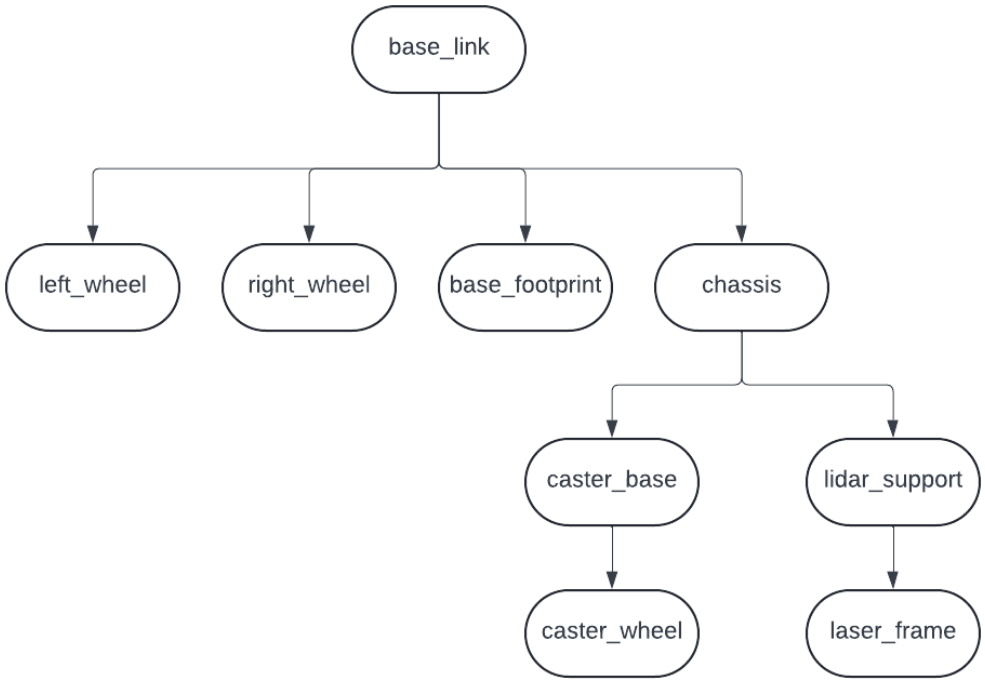

Figura 5: Árbol que muestra la relación entre los distintos links del robot base.

3.3.3. Rampa Don Barredora

Utilizando de base el modelo URDF “Robot Core”, se procedió a su expansión mediante la incorporación de la rampa mostrada en la figura 6, dando lugar a una variante del robot base denominado “Don Barredora”7.

La rampa se modeló utilizando Blender, y los momento de inercia fueron calculados utilizando MeshLab, específicamente la herramienta “Compute Geometric Measures”.

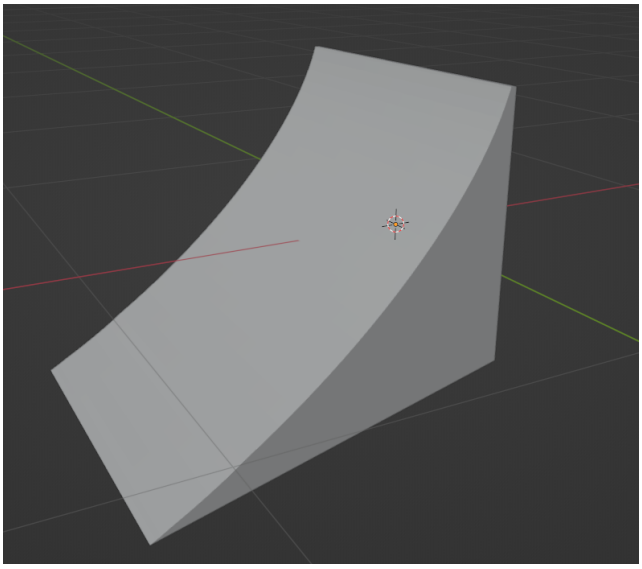

Figura 6: Mesh de la rampa integrada al modelo base para crear el robot“Don Barredora”

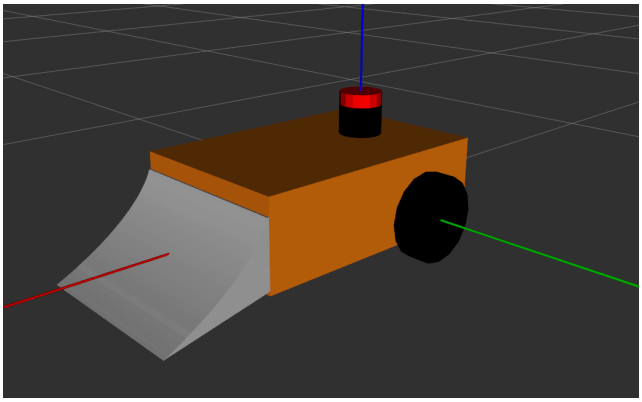

Figura 7: Modelo del robot Don Barredora

3.4. AxeBot

Este robot comparte características similares con el “Robot Core”, como la altura del LiDAR y la distancia entre las ruedas, no pudo construirse a partir de esa base. Esto se debe a que todas las partes del robot son personalizadas o lo que se denominan “custom meshes”. Para crear este robot se realizaron los siguientes pasos:

1. **Esculpido** de todas las partes del robot utilizando un software de es-

culpido 3D llamado zbrush.

2. Al trabajar con Zbrush la resolución de polígonos es alta. Debido a esto, se realizo un proceso de **retopología** que disminuye la cantidad de polígonos de todas las partes utilizando un plugin en el mismo software.

3. Se realizo un **escalado** del modelo para que tenga las proporciones similares las del modelo base, este modelo tiene proporciones de 100cm x 30cm x 40cm (dx, dy, dz).

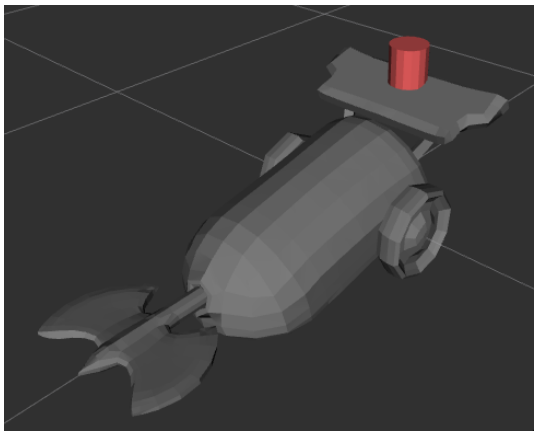

Figura 8: Mesh del robot AxeBot

4. Se **exportó** cada parte del modelo en formato .dae.

5. Al describir el modelo con sus respectivas partes en el archivo *.xacro* fue necesario un **reposición manual de los ejes** de origen de cada objeto que componía un *link*. (Para futuros robots se recomienda una corrección de ejes de referencia en el programa de desarrollo seleccionada para evitar este paso).

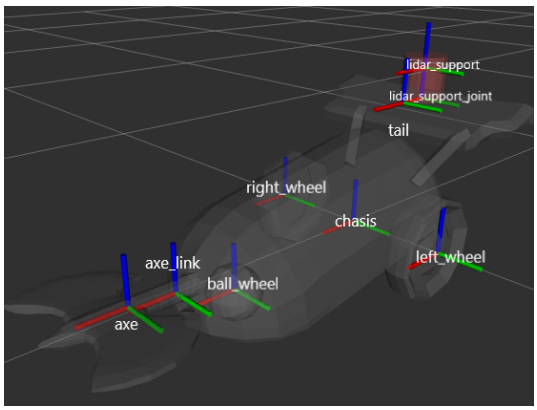

Figura 9: Ejes de referencia luego de la reposición.

6. Para cada parte del robot se **definió inercia** como las de una caja, las cuales no necesariamente deben coincidir con las dimensiones de la malla, y se les asigno un peso acorde.

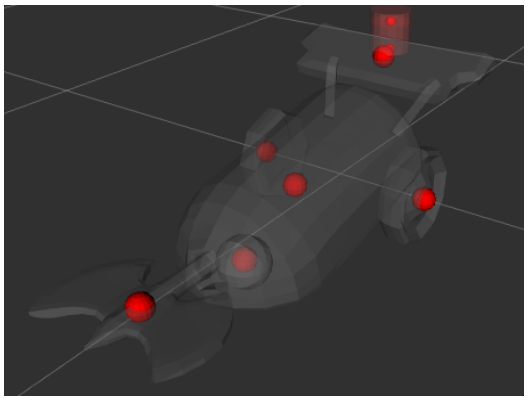

Figura 10: Punto de masa de los objetos.

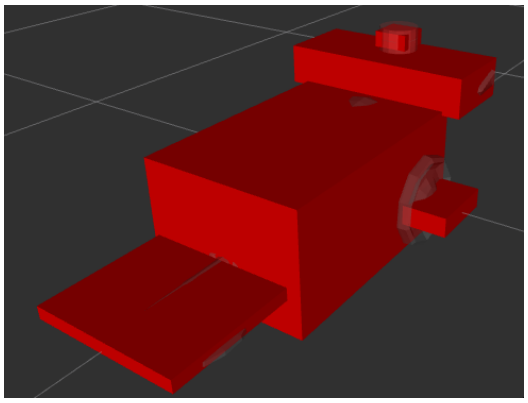

Figura 11: Modelo de cajas que representan la inercia de cada una de las partes del robot.

7. Se **integraron los sensores** definidos en el robot base, el LiDAR y el sensor odómetricos.

4. Sensores y Actuadores

4.1. LiDAR

Con el fin de observar el entorno, ubicarse y orientarse en él, además de generar un mapa de este, se añadió a nuestros robots un sistema LiDAR que utiliza rayos láser para medir distancias y movimientos precisos en tiempo real.

Para leer los datos que genera el LiDAR de los robots, se utilizó el plugin libgazebo_ros_ray_sensor.so. Este plugin es parte de la biblioteca ROS-Gazebo y cumple la funcionalidad de controlador de sensores de tipo ray, como el LIDAR, que es crucial para la percepción y el mapeo en los entornos de simulación.

4.1.1. Gazebo ROS ray sensor

El sensor ray requiere un elemento *scan* y otro *range*. El elemento *scan* define la disposición y el número de rayos, y el elemento *range* define las propiedades de un rayo individual.

Dentro del elemento *scan*>se encuentran los elementos *horizontal* y *vertical*. Estos definen los rayos que se abren en abanico en un plano horizontal, y los rayos que se abren en un plano vertical, respectivamente.

Para poder leer los datos generados se debe incluir el elemento *plugin*, mencionado antes, dentro del elemento *sensor*. Este proporciona una interfaz entre Gazebo y ROS para sensores de rayos, permitiendo la simulación precisa y la integración directa con el sistema ROS. A continuación, se detalla la configuración de nuestros LIDARs.

```
<gazebo reference="laser_frame">
    <sensor name="laser" type="ray">
        <pose> 0 0 0 0 0 0 </pose>
        <visualize>false</visualize>
        <update_rate>10</update_rate>
        <ray>
            <scan>
                <horizontal>
                    <samples>360</samples>
                    <resolution>$36</resolution>
                    <min_angle>-3.14</min_angle>
                    <max_angle>3.14</max_angle>
                </horizontal>
                <vertical>
                    <samples>1</samples>
                    <resolution>1</resolution>
                    <min_angle>0</min_angle>
                    <max_angle>0</max_angle>
                </vertical>
            </scan>
            <range>
                <min>0.3</min>
                <max>20</max>
            </range>
        </ray>
        <plugin name="laser_controller"
        filename="libgazebo_ros_ray_sensor.so">
            <ros>
                <remap from="~/out" to="scan"/>
            </ros>
            <output_type>sensor_msgs/LaserScan</output_type>
            <frame_name>laser_frame</frame_name>
        </plugin>
    </sensor>
</gazebo>
```

Figura 12: Contenido del archivo XML que configura el ray sensor

Este plugin lanza el nodo /laser_controller, y luego de remapea su salida hacia el tópico /scan. Este nodo publicará, con la frecuencia indicada (10 en el ejemplo de arriba), mensajes del tipo sensor_msgs/msg/LaserScan, con información del LiDAR. Como se ve el la tabla 3, poseen la siguiente estructura:

Campo	Tipo
header	std_msgs/Header
angle_min	float32
angle_max	float32
angle_increment	float32
time_increment	float32
scan_time	float32
range_min	float32
range_max	float32
ranges	float32[]
intensities	float32[]

Cuadro 3: Tabla que muestra la estructura de los mensajes enviados por el tópico /scan

4.2. IMU

Los sensores IMU (Inertial Measurement Units) son dispositivos que combinan acelerómetros, giroscopios y, a veces, magnetómetros para medir la orientación, la velocidad y la aceleración de un objeto. Estos datos pueden ser valiosos para el control y la navegación de robots, especialmente en entornos dinámicos o impredecibles. Sin embargo, en nuestro caso, hemos decidido **prescindir** del sensor IMU debido a los siguientes motivos:

- Complejidad: Integrar un IMU con otros sensores para mejorar la odometría, requiere el uso de un filtro de Kalman extendido (EKF). El EKF combina datos de múltiples sensores para estimar la posición y orientación del robot, utilizando una aproximación de Taylor para sistema no lineales, muy utilizado en sistemas de navegación. La implementación de este filtro añade una capa significativa de complejidad, por lo cual, optamos por eliminar el sensor IMU.
- Redundancia: La principal justificación para eliminar el sensor IMU es la redundancia innecesaria de información para estimar la posición y orientación del robot. Los datos obtenidos de los encoders y el LiDAR son suficientes para detectar y corregir deslizamientos o errores en el movimiento.

Para este trabajo, se ha determinado que el uso de los sensores de encoders y LiDAR proporciona resultados óptimos y precisos para la navegación,

en base a las simulaciones y aplicaciones realizadas. Al ejecutar SLAM sin un IMU, se logra una eficiencia más que suficiente para nuestros robots. En conclusión, la decisión de omitir el IMU, permite un diseño más sencillo sin comprometer la precisión.

4.3. Locomoción y Encoders

Un encoder es un dispositivo que se utiliza para convertir una posición o un movimiento en una señal eléctrica que puede ser leída por un sistema de control, como un microcontrolador o una computadora. En la robótica móvil, los encoders son esenciales para medir el movimiento y la velocidad de las ruedas del robot, proporcionando retroalimentación precisa para el control de su locomoción.

4.3.1. Differential Drive

La cinemática de los robots móviles creados utiliza el modelo de conducción diferencial (*differential drive*). La conducción diferencial implica que cada rueda tiene su propio motor que opera independientemente al otro. El modelo propuesto utiliza dos ruedas, cada una de diámetro r, y dado un punto P centrado entre las dos ruedas, cada rueda se encuentra a una distancia l desde P. Esta configuración permite al robot moverse en diferentes direcciones en el plano 2D. Dado r, l, θ y la velocidad de giro de cada rueda ($\dot{\varphi}_l$ y $\dot{\varphi}_r$), el modelo de cinemática directa puede predecir la velocidad total del robot en el sistema de referencia global:

$$\dot{\xi}_I = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = f(l, r, \theta, \dot{\varphi}_l, \dot{\varphi}_r)$$

Utilizando la siguiente ecuación $\dot{\xi}_R = R(\frac{\pi}{2})\dot{\xi}_I$ donde

$$R(\theta) = \begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Así podemos computar el movimiento del robot en el marco de la referencia global a partir del movimiento en sus sistema de referencia local $\dot{\xi}_R = R(\theta)^{-1}\dot{\xi}_R$

Los encoders permiten al sistema de control del robot determinar las velocidades angulares de las ruedas $\dot{\varphi}_l, \dot{\varphi}_r$, y con esta información, calcular su posición y orientación en el espacio.

4.3.2. Plugin Differential Drive

Gazebo ofrece un plugin llamado **Plugin Differential Drive**, que proporciona un controlador básico parasico para robots de conducción diferencial. Este plugin simula el comportamiento de los encoders y otros componentes necesarios para la locomoción, como los motores. Para el correcto funcionamiento de este, fue necesario definir la siguiente información:

- **Nombre del link de la rueda izquierda**.
- **Nombre del link de la rueda derecha**.
- **Diámetro de las ruedas** (en metros).
- **Distancia entre el centro de las ruedas** (en metros).
- **Máximo torque** que la rueda puede producir (en Nm).
- **Máxima aceleración** (en rad/s22).

El controlador creará el nodo `diff_drive`, el cual se suscribirá al tópico `/cmd_vel` y publicará en los tópicos `/tf` y `/odom`, como se puede ver en la Figura 13. Este nodo permite que el robot reciba comandos de velocidad y actualice su posición y orientación respecto al sistema de coordenadas del mundo.

```
ros@ros:~/proyecto_robotica_ws$ ros2 node info /diff_drive
/diff_drive
  Subscribers:
    /clock: rosgraph_msgs/msg/Clock
    /cmd_vel: geometry_msgs/msg/Twist
    /parameter_events: rcl_interfaces/msg/ParameterEvent
  Publishers:
    /odom: nav_msgs/msg/Odometry
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /tf: tf2_msgs/msg/TFMessage
```

Figura 13: Salida del comando de ROS 2 para poder ver información de un nodo.

La información de los encoders permite detectar y corregir deslizamiento o errores en el movimiento, mejorando la estabilidad y precisión del robot. El plugin también simula las fuerzas y torques aplicados a las ruedas, la fricción con la superficie, y otras dinámicas relevantes que afectan el movimiento.

5. Implementación de los entornos

La herramienta utilizada para la construcción de los entornos fue el editor de modelos de Gazebo. El proceso de construcción consistió en el uso de geometrías básicas propias del editor, tanto para los obstáculos como para los objetos inamovibles.

En el caso de los objetos obstáculo, Gazebo permite configurar su masa, inercia y escala, elementos que se modificaron para asegurar su correcta interacción con los robots.

Por otro lado, los objetos que delimitan los entornos, como los muros, tienen aplicada la configuración de *kinematic* con el fin de evitar que estos interactúen con el motor de físicas y actúen como objetos estáticos y delimitantes.

5.1. Formato SDF

El formato que se utiliza para trabajar entornos y modelos 3D en *Gazebo* es el formato .sdf, el cual es un estándar construido con XML mediante el uso de etiquetas predefinidas para describir no solo la forma de la malla del modelo, sino que también, todas las características dirigidas al motor de físicas de Gazebo. Algunas de estas etiquetas son:

- **<geometry>** Define el inicio de una figura geométrica.
- **<mesh>** Define el modelo 3d utilizado mediante una uri apuntando a un archivo de la computadora o a un recurso alocado en internet.
- **<friction>**, **<contact>**,**<collision>** Definen variables de como debe afectan la interacción del objeto con el motor de físicas.

5.2. Descripcion de entornos

El proyecto cuenta con 2 entornos utilizables, **Race.xml** (/src/boxbots/-worlds/Race_World) de la figura 14 y **Arena.xml** (/src/ boxbots/worlds/Are-

na_World) de la figura 15, los cuales están compuestos por objetos móviles e inmóviles.

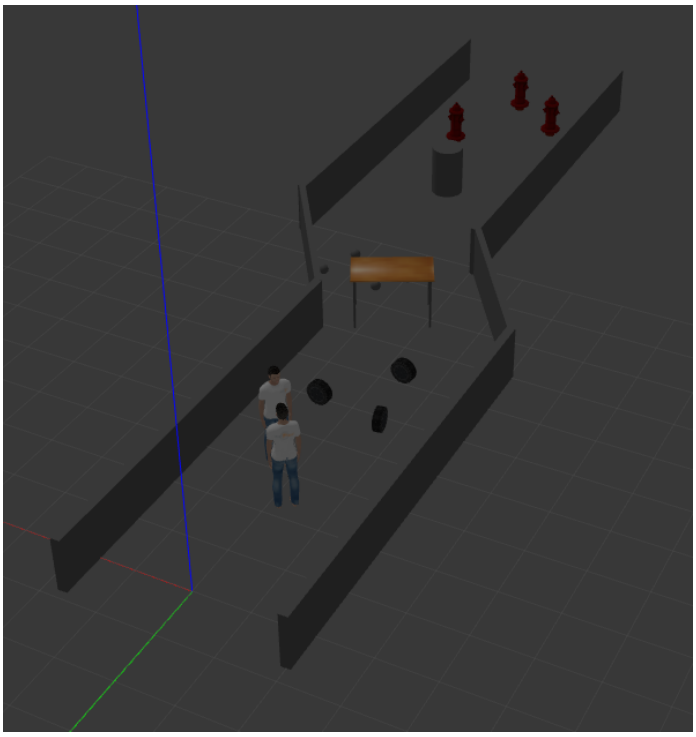

Figura 14: El otro entorno definido en **Race.xml**, define un pequeño camino de obstáculos con muros paralelos y modelos abiertos provistos por Gazebo.

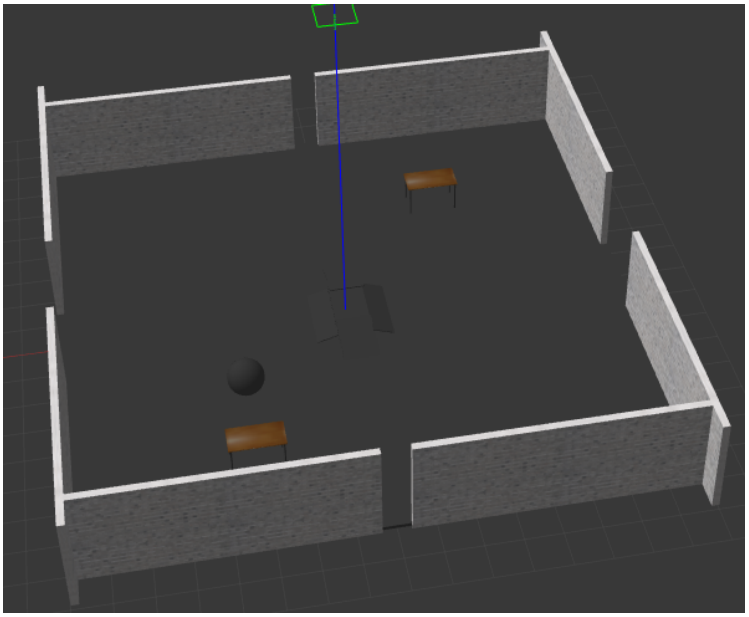

Figura 15: El entorno definido en **Arena.xml** es un pequeño “ring” con juegos para múltiples robots, nuevamente compuesto de muros, rampas y modelos abiertos de Gazebo.

6. Teleoperación

La teleoperación se refiere al control de un robot desde una ubicación remota, es decir, mediante control remoto. Esto se contrapone a la operación autónoma, donde al robot se le asigna una tarea específica que debe completar de manera independiente.

Aunque se busca que los robots sean lo más autónomos posible, en muchas situaciones todavía se requiere la intervención humana, especialmente durante las fases de desarrollo, configuración o en situaciones complejas.

La teleoperación suele constar de dos partes:

- **Envío de señales de comando al robot**: Esto suele implicar el envío de comandos de velocidad u otras instrucciones de movimiento.
- **Recepción de datos de sensores desde el robot**: Esta parte puede ser opcional si el operador está físicamente presente con el robot, pero es crucial para la operación remota, ya que proporciona retroalimentación visual y de otro tipo que ayuda a guiar el control del robot.

Después de publicar la descripción del robot (en /robot_description) y los sistemas de coordenadas iniciales (en /tf), mediante el nodo robot_state_publisher y el servicio de spawn, podemos visualizar en RViz el modelo en su estado inicial. Como se muestra en la figura 16, a partir de la percepción del robot mediante sus sensores, un teleoperador puede enviar comandos de velocidad (de tipo Twist) al controlador de conducción diferencial, para luego cerrar el ciclo actualizando los sistemas de coordenadas (incluyendo la odometría).

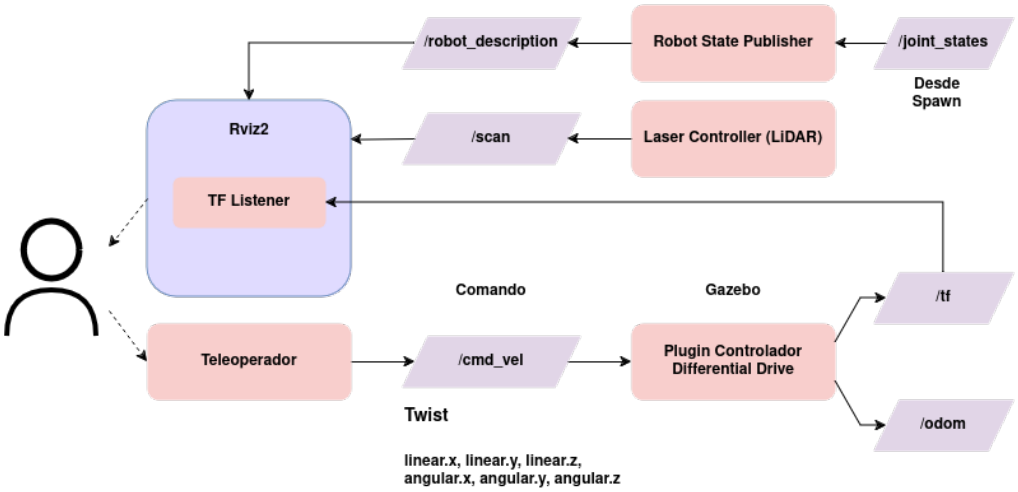

Figura 16: Esquema de comunicación entre el teleoperador y el robot

6.1. Teleoperación utilizando teclado

Una forma de teleoperar los robots es utilizando `teleop_twist_keyboard`, el cual es una herramienta para la teleoperación básica de robots móviles. Permite controlar un robot utilizando los comandos del teclado de una computadora.

Por ejemplo para teleoperar al robot Don Barredora con teclado se debe ejecutar el siguiente comando en la terminal:

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard --ros-args -r
    ↪  __ns:=/donBarredora
```

Este comando lanzará un nodo que traducirá las pulsaciones del teclado en mensajes que controlan la velocidad lineal y angular del robot. Tiene ciertas limitaciones, a seber:

- No proporciona una interfaz gráfica sofisticada ni control avanzado del robot.
- Requiere que la ventana de terminal se mantenga abierta y activa

Por lo tanto se buscó una solución más eficiente para teleoperar a los robots, la cual es utilizar joysticks o controles.

6.2. Teleoperación utilizando joystick

Los gamepads o joysticks son ideales para la robótica porque están diseñados para dirigir algo con facilidad, generalmente en un contexto virtual. Tienen varios ejes y numerosos botones, lo que permite activar diversas funciones necesarias para la operación del robot.

Con ROS, se puede decidir si conectar el gamepad directamente a la computadora del robot o a una estación base o máquina de desarrollo. Conectar a la máquina de desarrollo facilita el cambio entre simulación y el robot real sin modificar la configuración, aunque conectar directamente al robot puede ofrecer mejor rendimiento debido a la menor latencia.

6.2.1. Configuración en Linux

Para verificar que el gamepad funciona en Linux, se pueden instalar herramientas útiles:

```
sudo apt install joystick jstest-gtk evtest
```

Con el controlador conectado (vía USB o Bluetooth), se puede usar evtest para comprobar que los botones y ejes del gamepad funcionan correctamente.

6.3. Integración en ROS

ROS incluye paquetes que facilitan la integración de gamepads y joysticks. En lugar de ejecutar un programa que genere mensajes Twist directamente, el proceso se divide en dos partes:

- Comunicación con el joystick: A través de los controladores de Linux,

se publica un mensaje sensor_msgs/Joy que contiene el estado de los botones y ejes del gamepad.

- Uso de los datos del joystick: Otros nodos pueden utilizar estos datos para desencadenar comportamientos específicos, como mover una cámara o controlar el movimiento del robot.

Para verificar qué controladores puede ver ROS, se puede usar:

```
ros2 run joy joy_enumerate_devices
```

Usualmente, se usa el controlador con el ID 0.

Para ejecutar el nodo joy en ROS:

```
ros2 run joy joy_node
```

6.4. Creación de archivos de lanzamiento y parámetros con Python

Para configurar el control del gamepad, es necesario crear un archivo de lanzamiento y un archivo de parámetros. Esto es importante porque el siguiente nodo que vamos a ejecutar tiene muchos parámetros que especificar.

6.4.1. Archivo de parámetros

Debemos crear un archivo llamado joystick.yaml en el directorio config. Por ejemplo, puede contener los siguientes parámetros y valores:

```
joy_node:
  ros__parameters:
    device_id: 0
    deadzone: 0.05
    autorepeat_rate: 20.0
```

Nosotros especificamos los parámetros dentro de la generación del nodo `joy_node`:

```
#el nombre del robot que se quiere controlar con joystick
bot_namespace = 'donBarredora'

def generate_launch_description():
    use_sim_time = LaunchConfiguration('use_sim_time')

    joy_node = Node(
            package='joy',
            executable='joy_node',
            namespace=bot_namespace,
            parameters=[
                    {'deadzone': 0.05},
                    {'autorepeat_rate': 20.0},
                    {'use_sim_time': use_sim_time},
                    {'device_id': 0}],
                #en device_id se debe establecer el numero
                #que el sistema asigno al dispositivo que queremos utilizar
        )
```

Figura 17: Código del nodo joystick

6.4.2. Conversión de Joy a Twist

Una vez que los datos del joystick están disponibles en /joy, es necesario convertirlos a un mensaje Twist usando el paquete teleop_twist_joy. Esto permite controlar el robot basándose en la entrada del joystick.

Se puede configurar un archivo de parámetros joystick.yaml para especificar las características del gamepad, como los ejes utilizados para el movimiento lineal y angular, las velocidades máxima y turbo, y los botones de habilitación.

Ejemplo de parámetros en el siguiente chunk 4:

```yaml
teleop_node:
  ros__parameters:
    axis_linear:
      x: 1
    scale_linear:
      x: 0.5
    scale_linear_turbo:
      x: 1.0
    axis_angular:
      yaw: 0
    scale_angular:
      yaw: 0.5
    scale_angular_turbo:
      yaw: 1.0
    require_enable_button: true
    enable_button: 6
    enable_turbo_button: 7
```

Cuadro 4: Parametros que se configuran en el nodo teleop.

Particularmente, en la figura 18 siguiente se aprecia los parámetros especificados del yaml utilizando Python.

```python
teleop_node = Node(
        package='teleop_twist_joy',
        executable='teleop_node',
        name='teleop_node',
        namespace=bot_namespace,
        parameters=[
        {'axis_linear.x': 1},
        {'enable_button': 6},
        {'require_enable_button': False},
        {'enable_turbo_button': 5},
        {'scale_linear.x': 0.3},
        {'scale_linear_turbo.x': 2.0},
        {'axis_angular.yaw': 0},
        {'scale_angular.yaw': 1.0},
        {'scale_angular_turbo.yaw': 2.0},
        {'axis_linear.x': 1},
        {'use_sim_time': use_sim_time}],
        remappings=[('/cmd_vel','/cmd_vel_joy')]
    )
```

Figura 18: Figura que muestra los parámetros que se le envían al nuevo nodo teleop

Estos parámetros se establecieron en base a un joystick de PS4. Ante la imposibilidad de movimiento se debe comprobar los controles asignados (específicamente `axis_linear.x` y `axis_angular.yaw` que establecen qué botones del control se utilizan para controlar el movimiento `lineal` y `angular` respectivamente).

Para el movimiento del robot se utiliza el botón `L3` del control (hacia arriba y abajo se controla el movimiento lineal, y hacia la derecha e izquierda se controla el movimiento angular).Y el parámetro `enable_turbo_button`, fijado en el botón `R1`, establece qué botón se utiliza para asignarle mayor velocidad (no incremental).

6.4.3. Archivos de lanzamiento (launch)

- El archivo `joystick0.launch.py` en el directorio de lanzamiento, lanza los nodos `joy_node` y `teleop_node` con sus respectivos parámetros especificados anteriormente, controlando el robot `donBarredora` y utilizando el joystick que se le haya asignado `device_id`: 0. El comando de lanzamiento del archivo mencionado es el siguiente:

```
ros2 launch boxbots joystick0.launch.py
```

- El archivo joystick1.launch.py realiza el lanzamiento de los mismos nodos pero controlando al robot `axeBot` y utilizando el joystick que se le haya asignado `device_id`: 1. El comando de lanzamiento del archivo mencionado es el siguiente:

```
ros2 launch boxbots joystick1.launch.py
```

6.5. Obtener Retroalimentación

La retroalimentación es crucial en la teleoperación, ya que permite al operador recibir información sobre el estado y entorno del robot. RViz es una herramienta potente en ROS para visualizar esta información, aunque no siempre es la mejor opción para todas las aplicaciones.
Los tipos de retroalimentación contribuyen en el efecto de obtener una mayor precisión del estado físico real del robot. Entre sus tipos están:

- Odometría: Proporciona una idea general de la posición y el movimiento del robot basado en los encoders de los motores.
- Feed de Cámara: Permite al operador ver lo que el robot ”ve”, proporcionando una retroalimentación visual directa.
- Feed de Lidar: Proporciona información sobre el entorno del robot, especialmente útil en espacios interiores y corredores.

7. Teleoperación de dos robots en simultáneo

Dado que los robots construidos están diseñados a partir de una misma base, con un número idéntico de actuadores y sensores, los nodos y tópicos utilizados por cada robot comparten los mismos nombres. Esto puede generar problemas al comunicarse con los nodos, ya que estarían utilizando los mismos canales para transferir datos, lo que puede resultar en que la información sea leída por el robot equivocado.

Para solucionar esto se utilizan nombres de espacio o *namespaces* para agrupar los nodos en contextos individuales, permitiendo que dos nodos con el mismo nombre estén unívocamente identificados. Este efecto se logra estableciendo como parámetro el nombre del namespace al crear los nodos y plugins de los robots:

```
# Create a robot_state_publisher node
params = {'robot_description': robot_description_config.toxml(), 'use_sim_time': use_sim_time}
node_robot_state_publisher = Node(
    package='robot_state_publisher',
    namespace= robot_name,
    executable='robot_state_publisher',
    output='screen',
    parameters=[params],
    remappings=[('/robot_description', robot_description_topic)]
)
```

Figura 19: Segmento de código usado para incluir el archivo rsp.launch.py que describe el robot en la simulación utilizando los archivos .xacro

```
<plugin name="diff_drive" filename="libgazebo_ros_diff_drive.so">
    <ros>
        <namespace>/$(arg robot_name)</namespace>
    </ros>
```

Figura 20: Etiqueta para agregar el namespace

Al simular varios robots en un mismo entorno, se puede ejecutar el comando `ros2 node list` en una nueva terminal para mostrar la lista de nodos activos junto con sus namespaces correspondientes.

```
f@f-VirtualBox:~/proyecto_robotica_ws$ ros2 node list
/axeBot/diff_drive
/axeBot/imu_plugin
/axeBot/laser_controller
/axeBot/robot_state_publisher
/axeBot/teleop_twist_keyboard
/donBarredora/diff_drive
/donBarredora/imu_plugin
/donBarredora/laser_controller
/donBarredora/robot_state_publisher
/donBarredora/teleop_twist_keyboard
/gazebo
```

Figura 21: El nombre de cada nodo con un namespace al listarlos con el comando

7.1. Lanzamiento de simulación con dos robots

Para ejecutar el lanzamiento de la simulación con dos robots utilizando Arena.world, se debe ejecutar el siguiente comando en la terminal:

```
ros2 launch boxbots launch_sim_arena.launch.py
```

Las Figuras 22 y 23 presentan el estado inicial de la simulación con dos robots utilizando el mapa Arena.world.

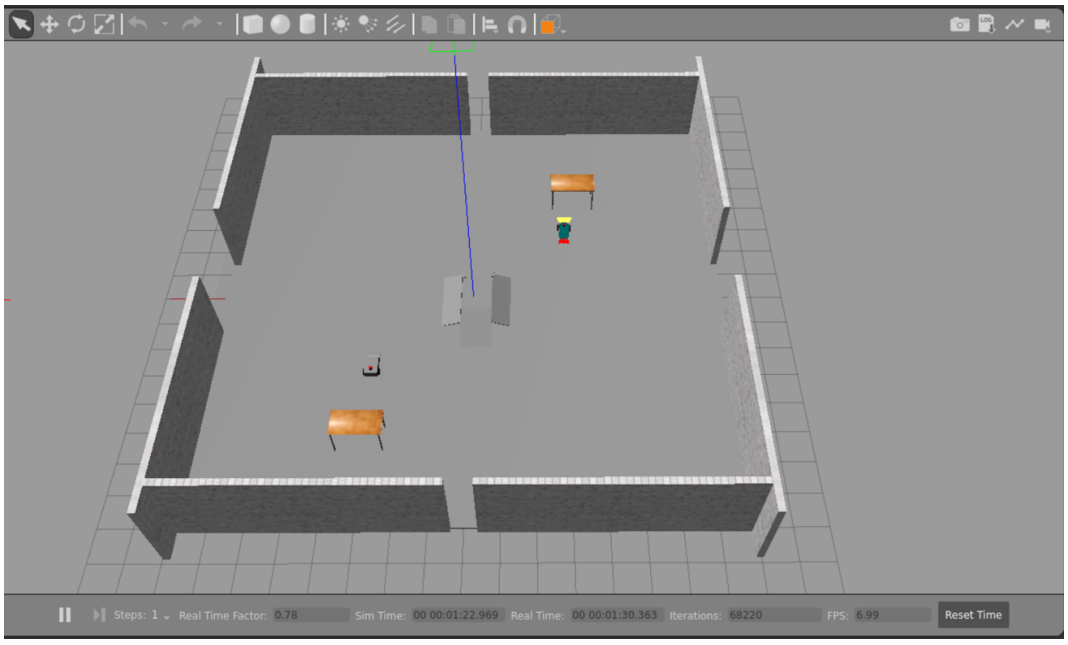

Figura 22: Captura 1 de simulación con dos robots en Arena.world

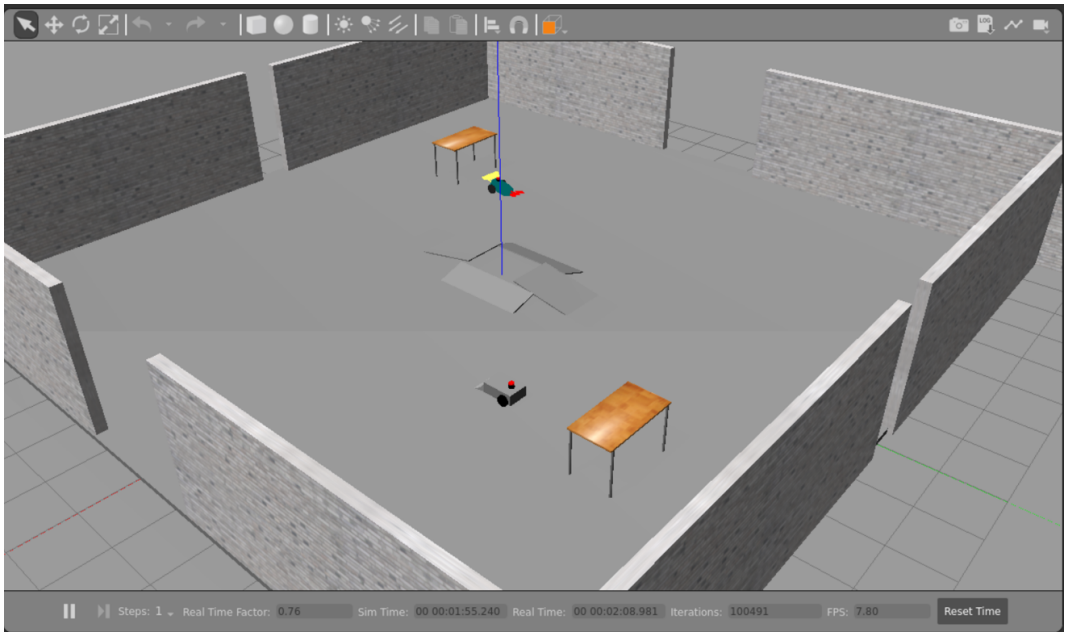

Figura 23: Captura 2 de simulación con dos robots en Arena.world

Para ejecutar el lanzamiento de la simulación con dos robots utilizando Race.world, se debe ejecutar el siguiente comando en la terminal:

```
ros2 launch boxbots launch_sim_race.launch.py
```

Las Figuras 24 y 25 presentan el estado inicial de la simulación con dos robots utilizando el mapa Race.world.

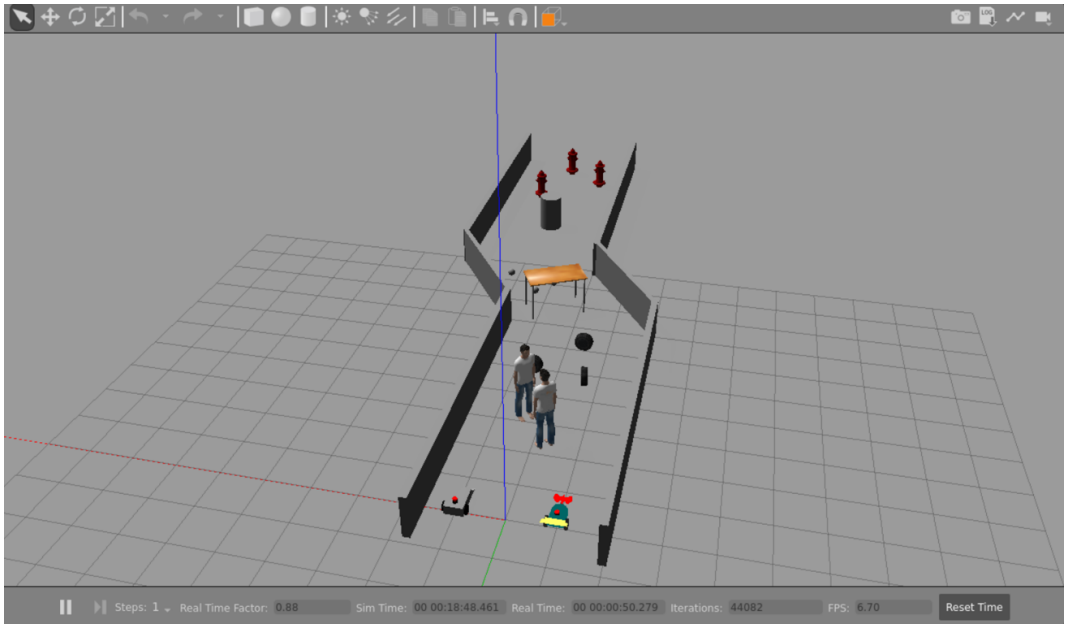

Figura 24: Captura 1 de simulación con dos robots en Race.world

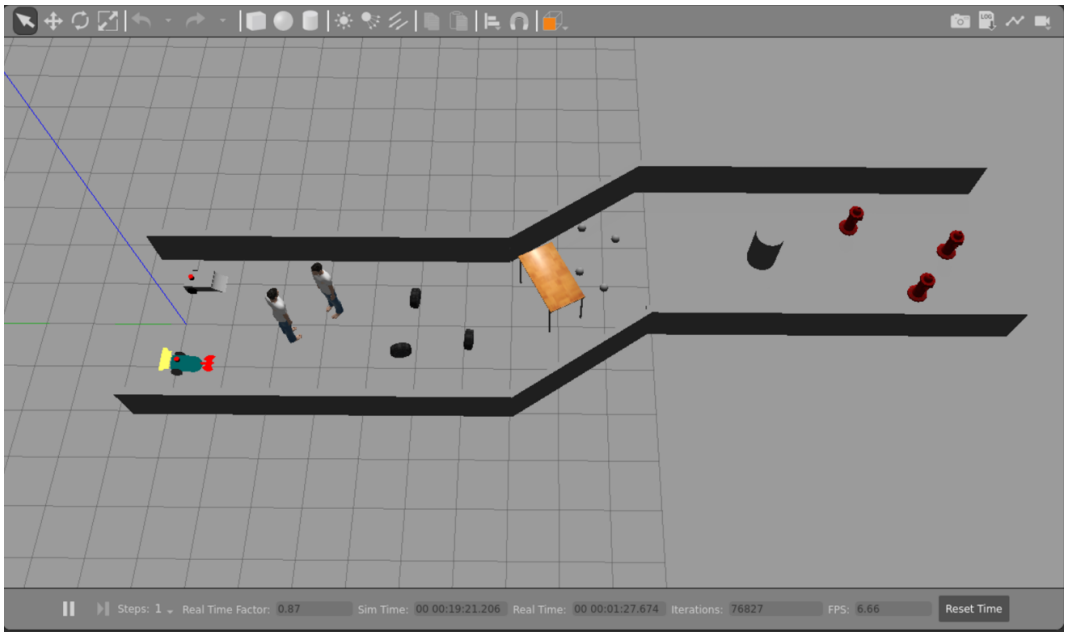

Figura 25: Captura 2 de simulación con dos robots en Race.world

7.2. Teleoperación en red

Adicionalmente, ROS2 está construido sobre un Data Distribution Service (DDS); un middleware que permite una intercomunicación mucho más

efectiva en los nodos que su predecesor en ROS1. Con el uso de namespaces, el DDS expone automáticamente los nodos de ROS2 a la red, pudiendo estos ser accedidos fácilmente y de manera remota por otro terminal. Así, en nuestro caso de estudio, si una terminal A levanta nodos y en la misma red a la que está conectada la terminal A hay una terminal B, entonces el comando `ros2 node list` mostrará los nodos disponibles en terminal A.

Una forma de evitar un potencial solapamiento entre ambas terminales que utilizan el mismo namespace, es mediante identificadores de dominio. Éstos permiten aislar cada namespace, evitando el conflicto en nodos que están corriendo en diferentes máquinas o en diferentes procesos en la misma máquina. (ver figura 26)

Figura 26: Dos dominios de ROS2 con sus respectivos namespaces. El primer dominio (ROS_DOMAIN_ID=1) contiene dos nodos: *turtlebot4_1* y *turtlebot4_2*. El segundo dominio (ROS_DOMAIN_ID=2) contiene dos nodos de ROS: *neobotix_1* y *neobotix_2*. Ambos dominios están conectados a través de una red, representada por un router. [1]

Para configurarlos, se debe establecer la variable de entorno *ROS_DOMAIN_ID* con el valor deseado. El comando *ros2* cargará automáticamente el valor de esta variable para ejecutar sus acciones de acuerdo a ese identificador. Sin embargo, si se desea ejecutar comandos desde una misma terminal con un identificador de dominio específico, se puede asignar el valor de esta variable antes de ejecutar el comando. Por ejemplo:

```
% ROS_DOMAIN_ID=1 ros2 run joy joy_node
% ROS_DOMAIN_ID=2 ros2 run joy joy_node
```

7.3. Visualización simultánea

La visualización simultánea de ambos robots utilizando Rviz2 es un tema complejo que no hemos implementado completamente debido a limitaciones de tiempo del desarrollo del proyecto. Existen soluciones para este problema, que implican el uso de plugins para realizar transformaciones y combinaciones de los mapas percibidos por cada robot.

Para la correcta simulación de múltiples robots, es crucial que cada robot tenga sus propios tópicos y namespaces, especialmente para tópicos críticos como /odom. Compartir un único tópico /odom entre varios robots puede causar conflictos y comportamientos no deseados.

En la figura 27 y 28, se presentan capturas de pantalla que muestran cómo podrían verse dos robots compartiendo un tópico común como /odom, aunque esta solución no es ideal. Simplemente se muestran para ilustrar lo que pudimos lograr con el tiempo de desarrollo disponible.

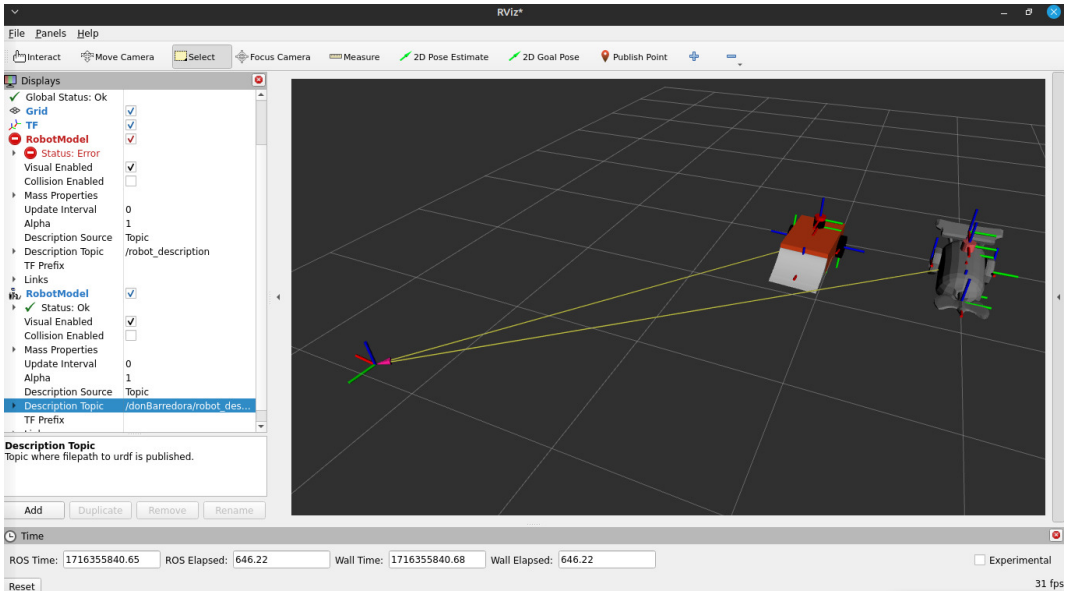

Figura 27: Captura de pantalla que muestra los dos robots en **Rviz2** a partir de **un mismo odom**, permitiendo que se puedan visualizar simultáneamente

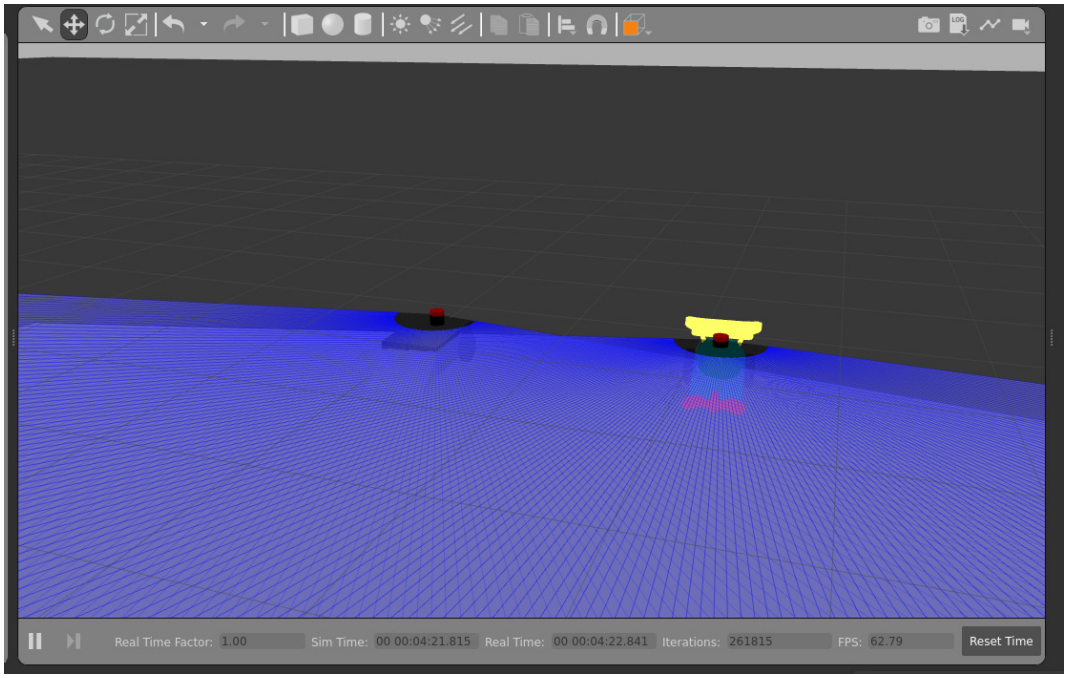

Figura 28: Captura de la misma escena de la figura 28 pero visualizándolo utilizando **Gazebo**

8. Navegación

8.1. SLAM (Simultaneous Localization and Mapping)

Utilizamos SLAM (Simultaneous Localization and Mapping) para mapear el entorno y localizar el robot en tiempo real. Para ello, cargamos el paquete slam_toolbox de ROS 2.

Para instalar el paquete slam_toolbox, se debe ejecutar el siguiente comando en la terminal:

```
sudo apt install ros-humble-slam-toolbox
```

Para correr el nodo slam_toolbox, se debe ejecutar el siguiente comando en la terminal:

```
ros2 launch slam_toolbox online_sync_launch.py slam_params_file
    ↪ :=./src/boxbots/config/mapper_params_online_sync.yaml
    ↪ use_sim_time:=true
```

- Aquí hay dos partes importantes a tener en cuenta:
  - Si queremos mappear el entorno en tiempo real, debemos modificar en el archivo mapper_params_online_sync.yaml el valor de mode en los ROS params de la siguiente manera: `yaml mode: "mapping"` y comentar el map_file_name y map_start_at_dock
  - Luego si queremos utilizar ese mapa debemos cambiar la linea por `yaml mode: "localization"` y descomentar el map_file_name y map_start_at_dock

Para visualizar el mapa generado por el nodo slam_toolbox, se debe ejecutar el siguiente comando en la terminal:

```
ros2 launch slam_toolbox nav2_map_server map_saver_cli -f map
```

8.1.1. Practica de SLAM

Aquí mostramos capturas del funcionamiento, para este ejemplo utilizamos 3 instancias de terminal abriendo en simultaneo el modulo de teleoperación, el visualizador rviz y una ejecución de gazebo que integra slam_toolbox

En una primera instancia, al abrir las terminales podemos observar el robot en su punto inicial, la terminal de teleoperación y un primer vistazo que publica el topico /map y recibe rviz

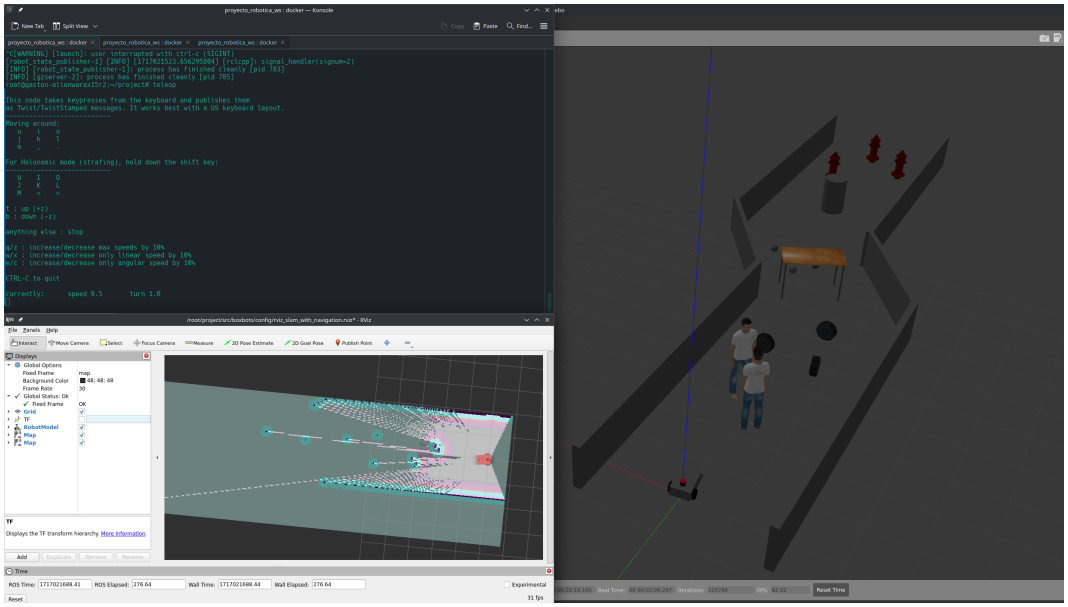

Figura 29: Inicialización de la prueba de slam.

A continuación, vemos el mapa una vez recorrida cierta distancia, aquí rviz ha guardado lo que publico el tópico /map y también las continuas capturas

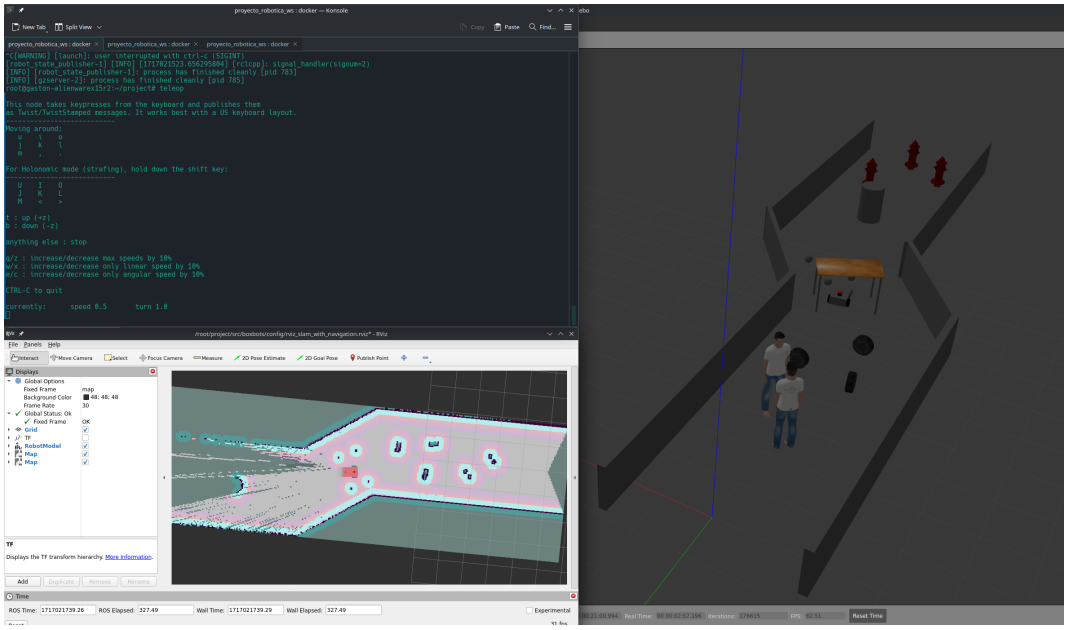

Figura 30: Figura que muestra el funcionamiento de slam.

8.2. AMCL (Adaptive Monte Carlo Localization)

AMCL es un algoritmo de localización ampliamente utilizado en robótica para estimar la posición de un robot móvil en un entorno desconocido utilizando sensores de percepción. Este método se basa en el enfoque de los filtros de partículas, donde se representa la creencia del robot sobre su posición en el espacio mediante una colección de partículas, cada una con su propia estimación de la posición del robot. A medida que el robot se mueve y recibe datos de los sensores, las partículas se actualizan y se ponderan de acuerdo con la probabilidad de que la observación sea consistente con la posición estimada del robot.

En ROS 2, el paquete `amcl` proporciona implementaciones de este algoritmo, permitiendo que los robots realicen la localización en tiempo real en entornos dinámicos y cambiantes.

En caso de que no este instalado, se debe ejecutar el siguiente comando en la terminal:

```
sudo apt install ros-humble-navigation2 ros-humble-nav2-bringup
```

Para utilizar el método de montecarlo para la localización del robot, se

debe ejecutar el siguiente comando en la terminal:

```
ros2 run nav2_map_server map_server --ros-args -p yaml_file_name
   ↪ :=./my_map_save.yaml -p use_sim_time:=true
```

Luego en otra terminal

```
ros2 run nav2_util lifecycle_bringup map_server
```

Esto activa el nodo map_server que se encarga de publicar el mapa en el topico /map y de suscribirse a los comandos de localización del robot.

Para correr el nodo amcl, se debe ejecutar el siguiente comando en la terminal:

```
ros2 run nav2_amcl amcl --ros-args -p use_sim_time:=true
```

Esto activa el nodo amcl que se encarga de localizar el robot en el mapa. Y en otra terminal

```
ros2 run nav2_util lifecycle_bringup amcl
```

Luego se debe señalar el punto de inicio del robot en el mapa en Rviz, para ello se debe hacer click en el boton 2D Pose Estimate y seleccionar el punto de inicio del robot en el mapa.

Para la navegación autónoma de los robots, utilizamos el paquete `nav2_bringup`. Este paquete proporciona las funcionalidades básicas para definir las rutas por las cuales los robots se desplazarán durante la prueba.

8.3. Funcionamiento

Al analizar el código fuente, observamos que se generan diversos nodos que nos ayudarán a configurar un grupo de nodos para la navegación de los robots los cuales se detallan a continuación.

- `nav2_controller`: Ejecuta el servidor del controlador de navegación. Es responsable de generar comandos de velocidad para que el robot

siga una trayectoria específica. Utiliza los parámetros configurados y remapea las conexiones de los tópicos, como el tópico de velocidad de comandos (cmd_vel).

- `nav2_smoother`: Suaviza la trayectoria generada por el planificador de navegación. Ayuda a evitar movimientos bruscos y mejora la estabilidad del robot.
- `nav2_planner`: Es responsable de generar una trayectoria global para el robot. Utiliza información del mapa y otros datos para planificar la ruta.
- `nav2_behaviors`: Maneja los comportamientos de alto nivel del robot, como detenerse, evitar obstáculos o seguir una referencia de posición.
- `nav2_bt_navigator`: Implementa un árbol de comportamiento (Behavior Tree) para coordinar las acciones de navegación del robot.
- `nav2_waypoint_follower`: Sigue una serie de puntos de referencia (waypoints) en la trayectoria global. Es útil para seguir rutas predefinidas.
- `nav2_velocity_smoother`: Suaviza aún más las velocidades de comandos antes de enviarlas al controlador. Esto ayuda a evitar cambios bruscos en la aceleración.
- `nav2_lifecycle_manager`: Gestiona el ciclo de vida de los nodos de navegación. Puede iniciar, detener o reiniciar los nodos según sea necesario.

8.4. Ejecución

Para poder ejecutar el script correspondiente a la navegación de nodos, primero debemos asegurarnos de instalar las dependencias correspondientes. Si se ha ejecutado el comando dependencies.sh no hay de que preocuparse, caso contrario se debe hacer o instalar el paquete `twist-mux` con el siguiente comando:

```
sudo apt install ros-humble-twist-mux
```

8.4.1. Ejecución Automática

Una vez tengamos las dependencias instaladas podemos ejecutar el siguiente script el cual abrirá una ventana con la simulación en gazebo y otra en rviz en el mapa que se haya definido dentro del archivo de lanzamiento `launch_sim_slam`. Esto se puede realizar con el siguiente comando:

```
./start_navigation.sh
```

Cabe destacar que debemos estar posicionados en la carpeta `scripts` del proyecto dentro de la terminal.

8.4.2. Ejecución Manual

Si no queremos que se ejecute lo antes mencionado, podemos seguir los siguientes pasos para realizarlo manualmente o solo por partes.

1) Ejecución de simulación en Gazebo.

```
ros2 launch boxbots navigation_launch.launch.py
```

2) Ejecución de simulación en RVIZ. *(Verifique el path hacia la configuración)*

```
   bash ros2 run rviz2 rviz2 -d boxbots/config
      ↪ rviz_slam_with_navigation.rviz
```

Esta configuración muestra el mapa resultante de slam + el mapa resultante del navigation.

3) Si se quiere utilizar teleoperación usando joystick

```
ros2 run twist_mux twist_mux --ros-args --params -file
   ↪ boxbots/config/twist_mux.yaml -r cmd_vel_out:=
   ↪ diff_cont/cmd_vel_unstamped
```

8.4.3. Practica de navegación

En esta sección mostramos como es la vista en el uso de la navegación. Aqui podemos ver que al hacer click en el botón 2D Goal Pose podemos seleccionar la posición a la que queremos que el robot llegue y la dirección a la cual queremos que apunte cuando finalice

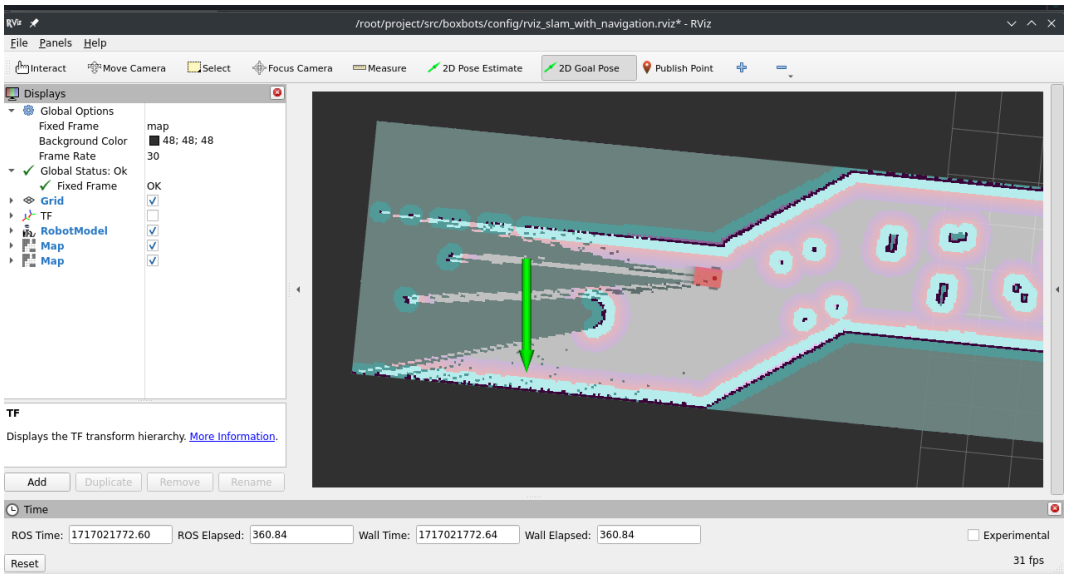

Figura 31: Figura que muestra el funcionamiento de navegación.

Luego de cierto tiempo el robot se dirigirá a la ubicación estimada.

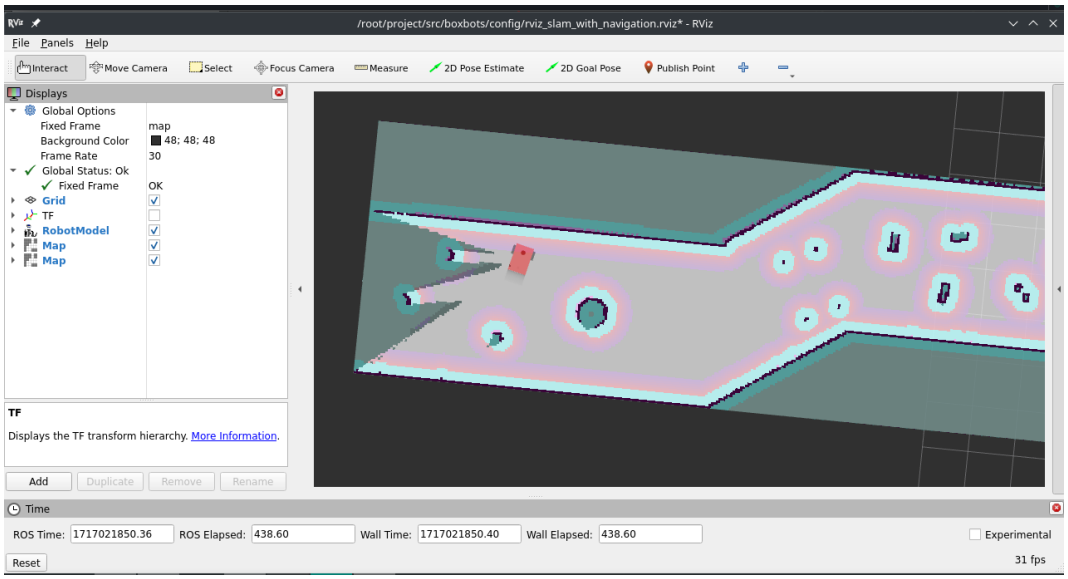

Figura 32: Navegación concluida.

9. Conclusión

En este trabajo, hemos explorado y aplicado una variedad de conocimientos fundamentales e introductorios en el campo de la robótica móvil, utilizando el framework ROS 2. A través de este proyecto, hemos aprendido sobre su arquitectura, que incluye la gestión y comunicación de nodos y el uso de namespaces para organizar un sistema de simulación de robots. Además hemos aprendido como utilizar herramientas como *ros2 launch* y *colcon*.

El diseño introductorio de robots móviles nos ha permitido comprender la integración de hardware y software, utilizando diversos sensores y actuadores. Hemos implementado y controlado robots móviles, beneficiándonos de la modularidad y flexibilidad que ofrece ROS 2.

Los paquetes de ROS 2 para visualización y simulación, como *rviz2* y *Gazebo*, han sido cruciales para el desarrollo y nos han permitido visualizar datos en tiempo real, simular escenarios complejos y realizar pruebas exhaustivas sin riesgo de dañar los componentes físicos.

9.1. Aprendizaje Obtenido a lo largo del Proyecto

Durante el desarrollo del proyecto de Robótica Móvil, obtuvimos un amplio aprendizaje en diversas áreas de la robótica y la integración de tecnologías. A continuación, se detallan los aprendizajes más significativos:

1. **Uso de ROS 2**: Aprendimos a utilizar el Robot Operating System 2 (ROS 2), específicamente la distribución Humble, como marco de trabajo principal para el desarrollo de software de robots. ROS 2 nos permitió gestionar nodos, servicios y acciones, facilitando la comunicación entre diferentes componentes del sistema. Además, la integración con sensores y actuadores fue crucial para el control y la operación de los robots.

2. **Simulación con Gazebo**: Implementamos simulaciones utilizando Gazebo, un simulador de entornos virtuales y algoritmos de robótica. Esta herramienta nos permitió validar el comportamiento de los robots en un entorno controlado antes de llevarlo a hardware real. La simula-

ción incluyó la creación de entornos 3D y la integración de los robots diseñados.

3. **Dockerización del Entorno de Desarrollo**: Para manejar las dificultades de compatibilidad entre versiones de software y sistemas operativos, aprendimos a crear contenedores Docker. Esto nos permitió establecer un entorno de desarrollo consistente y replicable, asegurando que todos los miembros del equipo trabajaran con las mismas versiones de software y evitando problemas de configuración.

4. **Diseño y Construcción de Robots**: Diseñamos y construimos dos robots personalizados, Don Barredora y AxeBot. Esto incluyó la definición de sus características físicas y funcionales mediante archivos URDF (Unified Robot Description Format) y la configuración de sus sensores y actuadores.

5. **Teleoperación y Control**: Implementamos sistemas de teleoperación utilizando teclado y joystick, basándonos en los paquetes 'teleop_twist_keyboard' y 'teleop_twist_joy' de ROS 2. Además, aprendimos a gestionar la multiplexación de comandos de velocidad utilizando 'twist_mux' y a integrar estos sistemas en un entorno de simulación.

6. **Navegación y Localización**: Utilizamos algoritmos de SLAM (Simultaneous Localization and Mapping) y AMCL (Adaptive Monte Carlo Localization) para la navegación autónoma de los robots. Esto nos permitió mapear entornos desconocidos y localizar los robots dentro de esos mapas, esencial para tareas de navegación complejas.

7. **Trabajo Colaborativo**: A lo largo del proyecto, trabajamos de manera colaborativa utilizando un repositorio de GitHub para gestionar el código y los recursos del proyecto. Establecimos convenciones para la creación de ramas y la realización de pull requests, asegurando un flujo de trabajo organizado y eficiente.

Este proyecto nos brindó una experiencia práctica y profunda en el desarrollo de sistemas robóticos complejos, desde la simulación y diseño de robots hasta su control y navegación en entornos virtuales. La combinación de herramientas modernas como ROS 2, Gazebo y Docker, junto con un enfoque colaborativo, nos permitió abordar desafíos técnicos y organizativos de manera efectiva.

10. Referencias

1. Siegwart, R., Nourbakhsh, I.R. and Scaramuzza, D. (2011) *Introduction to autonomous mobile robots*. Cambridge, Mass: MIT Press.
2. *ROS 2 Documentation - ROS 2 Documentation: Humble documentation*. Available at: `https://docs.ros.org/en/humble/index.html`.
3. *Gazebo Docs*. Available at: `https://gazebosim.org/docs`.
4. *Articulated Robotics RSS*. Available at: `https://articulatedrobotics.xyz/tutorials/`.
5. Prakash, P. (2023) Robot localization and AMCL, LinkedIn. Available at: `https://www.linkedin.com/pulse/robot-localization-amcl-pratheesh-prakash/`.