Práctica 3: Modern Gazebo Fortress

Robótica de Servicios

Máster en Robótica e Inteligencia Artificial

Resumen

Esta práctica contiene el código fuente para seguir el tutorial de ROSConES 2024 titulado: *Tutorial de Gazebo e integración con ros2_control* realizado por Jonathan Cacace de la **Unidad de Robótica de Eurecat** el **Horario**: 19 de septiembre de 2024, 9:30-11:30.

El tutorial ha sido extendido para mejorar la comprensión de conceptos esenciales para el uso de *modern Gazebo* y ROS 2.

Los objetivos planteados para esta práctica son:

- Familiarizarse con las distintas opciones a la hora de trabajar con Ignition.
- Reconocer el formato de los ficheros más habituales en Gazebo.
- Reconocer el formato de otros ficheros ROS.
- Conocer el sistema básico de Gazebo Bridge entre ROS y Gazebo[1] y [2].

Requisitos

Será necesario disponer de un entorno de trabajo base. La recomendación por simplicidad es la siguiente:

- Robot Operating System 2 (ROS 2)
  - Ubuntu 22.04
  - ROS 2 Humble
  - Ignition Gazebo Fortress

1. Instalación

Docker

El tutorial está diseñado para ROS2-Humble (LTS). Puedes instalarlo en tu sistema (Ubuntu 22.04) o en un contenedor Docker. El repositorio contiene los archivos necesarios para crear la imagen Docker e instanciar el contenedor.

Descargar el repositorio

Puedes descargar este repositorio en cualquier lugar en tu sistema. Se asume que lo clonarás en tu carpeta personal (`home`)

```
$ cd ~
$ git clone https://github.com/fjrodl/ROS2andGazebo.git
```

Construir y ejecutar la imagen

```
$ cd ROSConES/Docker
$ docker build -t ros2roscon .
$ docker compose up # You will lose the control of this terminal tab
```

Para adjuntar un nuevo terminal a este contenedor Docker, usa el comando `docker exec`

```
$ docker exec -it ros2roscon_container bash
```

Este último comando abre el acceso al contenedor Docker en el nivel superior de la carpeta Docker. Este será tu espacio de trabajo de ROS 2 (la `ros workspace`). Por ejemplo, aquí puedes compilar el espacio de trabajo:

```
$ colcon build --symlink-install
$ source /opt/ros/humble/setup.bash
```

Ahora ya puedes comenzar a compilar y ejecutar los ejemplos.

Instalación modern Gazebo (Ignition Fortress)

Ignition: Es un conjunto de herramientas y bibliotecas de simulación avanzadas que permiten modelar y probar robots en entornos virtuales. Ignition (anteriormente Gazebo clásico) es usado comúnmente para crear mundos simulados donde los robots interactúan con sensores, objetos y el entorno

Instalación de Gazebo

Para instalar en tu equipo con Ubuntu Ignition Gazebo Fortress (si no usas Docker) ejecuta los siguientes comandos:

```
$ sudo wget https://packages.osrfoundation.org/gazebo.gpg -O /usr/share/keyrings/pkgs-osrf-
    archive-keyring.gpg
$ sudo echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/pkgs-osrf-
    archive-keyring.gpg] http://packages.osrfoundation.org/gazebo/ubuntu-stable $(
    lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/gazebo-stable.list > /dev/null
$ sudo apt-get update
$ sudo apt-get install ignition-fortress ros-humble-ros-ign-bridge ros-humble-ros-gz ros-
    humble-controller-manager ros-humble-ros2-control ros-humble-ros2-controllers ros-humble
    -ign-ros2-control -y
```

Índice

Introducción a Gazebo

Puedes interactuar con la simulación de Gazebo utilizando el terminal de Linux con el comando `ign`.

```
The \texttt{ign} command provides a command line interface to the ignition tools.
\texttt{ign <command> [options]}
```

Para comenzar la simulación:

```
$ ign gazebo
```

Esto abrirá una página de bienvenida donde puedes seleccionar una escena lista o una escena vacía. Esta escena es bastante inutilizable, ya que también le falta el plano del suelo (Si agregas un objeto a la escena, caerá.). Para abrir directamente una escena vacía, usa este comando:

```
$ ign gazebo empty.sdf
```

Este es un mejor punto de partida. En esta ventana, además del escenario principal, hay elementos de la interfaz gráfica:

- **World**: todos los elementos
- **Entity Tree**: los objetos

Gazebo topics

Internamente, Gazebo comparte información utilizando *topics*, exactamente como ROS 2. Usaremos esta información para la integración de ROS 2 y Gazebo luego. Algunos comandos son útiles para entender los datos activos en el sistema:

```
$ ign topic
Options:
  -h,--help                   Print this help message and exit
  -v,--version
  -t,--topic TEXT             Name of a topic
  -m,--msgtype TEXT           Type of message to publish
  -d,--duration FLOAT Excludes: --num Duration (seconds) to run
  -n,--num INT Excludes: --duration Number of messages to echo and then exit

```

```
[Option Group: command]
  Command to be executed
  Options:
    -l,--list
    -i,--info Needs: --topic
    -e,--echo
    -p,--pub TEXT Needs: --topic --msgtype
```

Por ejemplo, para listar los tópicos activos:

```
$ ign topic -l
```

2. Ejemplos

Example 1: Key Publisher from Gazebo

Un plugin de Gazebo es un componente de software que añade funcionalidades específicas a la simulación. En este caso, vamos a usar el plugin Key Publisher, que permite capturar pulsaciones del teclado y enviarlas a la simulación.

- Abre Gazebo con una escena vacía

```
$ ign gazebo empty.sdf
```

- Añade el plugin Key Publisher:
  - Haz clic en los 3 puntos verticales -¿busca y selecciona el plugin Key Publisher
- Inicia la simulación con el botón de reproducción
- En un terminal, verifica los temas activos actuales:

```
$ ign topic -l
```

- Escucha el contenido del tópico: `/keyboard/keypress`

```
$ ign topic -e -t /keyboard/keypress
```

- Ahora, usa el teclado en la escena de Gazebo y verifica el output en el terminal.

Ahora tenemos los elementos básicos para usar Gazebo con ROS2.

Example 2: Desplegar objeto en la escena

Gazebo es un software independiente y es agnóstico a ROS 2. Soporta SDF como modelo de objeto: un robot o un objeto estático. SDF significa Simulation Description Format. Puedes encontrar diferentes simulaciones con robots complejos que utilizan directamente SDF.

El *Simulation Description Format* (**SDF**) es un formato XML utilizado para describir de manera detallada los elementos dentro de una simulación, como robots, objetos estáticos, entornos y escenas completas. Es ampliamente utilizado en simuladores como Gazebo para representar tanto aspectos físicos como propiedades de comportamiento de los objetos simulados.

Características principales de SDF

- **Formato XML**: SDF está basado en XML, lo que permite estructurar la información de manera jerárquica. Esta jerarquía es útil para describir no solo el robot, sino también sus componentes, propiedades físicas, conexiones, sensores y actuadores.
- **Agnóstico a ROS**: Aunque se utiliza frecuentemente en simulaciones con ROS, SDF es independiente de cualquier sistema de middleware robótico. Esto significa que puede usarse en simuladores sin la necesidad de ROS, lo que lo convierte en una herramienta flexible.
- **Representación física**: SDF permite definir las propiedades físicas de los objetos, como masa, fricción, inercia, geometría de colisiones, y materiales visuales. También admite la descripción de las conexiones entre diferentes partes del robot o los objetos (por ejemplo, *joints* entre enlaces).
- **Soporte para sensores y actuadores**: SDF puede describir sensores como cámaras, LiDARs, sensores de contacto, y más. También es posible definir actuadores como motores o transmisiones que controlan las articulaciones de un robot.

SDF y robots complejos

El formato SDF es esencial para simular robots complejos, ya que permite definir:

- La estructura del robot, incluyendo sus enlaces (*links*) y las articulaciones (*joints*) que conectan esos enlaces.
- Los sensores y actuadores que se usan para interactuar con el entorno, como cámaras, IMUs (unidades de medición inercial), y ruedas controladas por motores.
- Los aspectos físicos del robot, como las masas y los momentos de inercia de cada parte del robot.

Un robot simulado con SDF puede incluir múltiples enlaces, actuadores, y sensores, permitiendo realizar simulaciones realistas de robots complejos como manipuladores industriales, vehículos autónomos o robots cuadrúpedos.

Además, SDF es utilizado comúnmente para describir entornos completos, como edificios, terrenos, y otros objetos estáticos que conforman el mundo simulado.

Ejemplo básico de un archivo SDF

A continuación se muestra un ejemplo simplificado de un archivo SDF para describir un robot con un solo enlace (`link`) y una caja como geometría visual:

```
<?xml version="1.0" ?>
<sdf version="1.6">
  <model name="simple_robot">
    <link name="base_link">
      <visual>
        <geometry>
          <box>
            <size>1 1 1</size>
          </box>
        </geometry>
      </visual>
      <collision>
        <geometry>
          <box>
            <size>1 1 1</size>
```

```
          </box>
        </geometry>
      </collision>
      <inertial>
        <mass>1.0</mass>
        <inertia>
          <ixx>0.083</ixx>
          <iyy>0.083</iyy>
          <izz>0.083</izz>
        </inertia>
      </inertial>
    </link>
  </model>
</sdf>
```

Este archivo describe un modelo simple llamado `simple_robot`, que contiene un único enlace (`base_link`) con geometría en forma de caja (`box`). Se incluyen las propiedades físicas necesarias para las simulaciones, como la masa y la inercia, así como los componentes visuales y de colisión.

Uso de SDF en simulaciones

SDF se utiliza en simuladores como Gazebo para crear modelos tanto de robots como de entornos. En el contexto de ROS 2, SDF se puede integrar fácilmente con ROS mediante el uso de plugins y herramientas como `ros_ign_bridge`, que permiten comunicar datos entre Gazebo y ROS.

En resumen, SDF es una herramienta poderosa para describir de manera detallada tanto robots como entornos en simulaciones realistas, siendo agnóstica a cualquier framework de control y altamente compatible con simuladores populares como Gazebo.

Uso de DAE en simulaciones

Los **DAE** (Collada Digital Asset Exchange) son un formato de archivo estándar basado en XML utilizado para la interoperabilidad entre diferentes herramientas de gráficos 3D y simulación. Los archivos DAE permiten almacenar información sobre modelos tridimensionales, incluidas geometrías, texturas, materiales, luces, cámaras, y también animaciones y estructuras jerárquicas.

En el contexto de la robótica y la simulación, especialmente en entornos como *Gazebo*, los archivos DAE se utilizan principalmente para describir la geometría y las visualizaciones de robots y otros objetos. A continuación se describen algunas de sus características clave:

- **Interoperabilidad**: Es un formato abierto y ampliamente soportado por muchas herramientas de modelado 3D, como Blender, Maya y 3ds Max. Esto facilita la creación y modificación de modelos tridimensionales para simulaciones.
- **Descripción geométrica**: Al igual que otros formatos 3D (como STL u OBJ), los archivos DAE contienen descripciones detalladas de las geometrías de los objetos, como vértices, caras y normales, lo que permite representaciones visuales precisas de los robots en simuladores como Gazebo.
- **Texturas y materiales**: A diferencia de algunos formatos más simples, como STL, los archivos DAE pueden contener información sobre texturas y materiales, lo que permite definir cómo se ve la superficie de un objeto en una simulación o visualización gráfica.

- **Animaciones y esqueleto**: El formato DAE también soporta la animación y la definición de estructuras esqueléticas, lo que puede ser útil para modelar robots con partes móviles complejas o para simular movimientos detallados.

En el contexto de simuladores como Gazebo, el archivo DAE se utiliza a menudo para definir la apariencia visual del robot o los objetos del entorno, mientras que el comportamiento físico y las propiedades dinámicas se definen en archivos URDF o SDF.

Launcher .py en ROS 2

Vamos a crear un nuevo paquete que contendrá el modelo de un objeto simple y su *launch file*.

Un *launcher* `.py` en el contexto de ROS 2 y simulaciones con Gazebo es un archivo de lanzamiento (*launch file*) escrito en Python que automatiza la configuración y ejecución de nodos y otros componentes del sistema. Los archivos de lanzamiento en ROS 2 permiten organizar y simplificar el proceso de iniciar múltiples nodos, establecer parámetros y realizar conexiones entre los diferentes componentes del sistema.

En lugar de iniciar manualmente cada nodo desde la terminal con comandos individuales, un *launch file* en Python centraliza todo esto, lo que facilita el despliegue de simulaciones complejas o sistemas robóticos.

Funcionalidades principales de un launcher .py

- **Iniciar Nodos**: Los archivos de lanzamiento definen qué nodos deben ejecutarse y con qué parámetros. Esto incluye desde nodos ROS hasta la inicialización de plugins, simulaciones en Gazebo, y otros componentes.
- **Definir Parámetros**: Puedes establecer los parámetros específicos para los nodos, como la configuración de sensores, valores predeterminados o configuraciones de los controladores.
- **Remapeo de Tópicos**: Permite cambiar la conexión entre tópicos (*remapping*), de modo que un nodo publique o suscriba a diferentes tópicos según la necesidad.
- **Modularidad**: Los *launch files* pueden incluir otros *launch files*, lo que permite dividir configuraciones complejas en archivos más manejables y modulares.

Ejemplo básico de un launch file en Python

El siguiente ejemplo muestra cómo lanzar dos nodos en ROS 2: uno para el controlador de un robot (`robot_driver`) y otro para visualizar datos en RViz.

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='my_robot_package',   # Nombre del paquete
            executable='robot_driver',    # Nodo a ejecutar
            name='robot_driver',          # Nombre del nodo
            output='screen',              # Redirigir salida a la terminal
            parameters=[{'use_sim_time': True}]  # Parametros adicionales
        ),
        Node(
            package='rviz2',              # Nodo para visualizar el entorno
            executable='rviz2',
```

```
            name='rviz2',
            output='screen'
        )
    ])
```

En este ejemplo, el archivo `.py` lanza dos nodos: uno para el controlador de un robot (`robot_driver`) y otro para visualizar datos en RViz. Se incluyen opciones como la redirección de salida a la terminal (`output='screen'`) y el uso del tiempo de simulación (`use_sim_time`).

El uso de *launch files* hace que trabajar con simulaciones robóticas o sistemas distribuidos sea más eficiente y estructurado.

```
$ ros2 pkg create spawn_simple_object --dependencies xacro
$ cd spawn_simple_object
$ mkdir launch
$ mkdir urdf
$ cd urdf && touch cube.urdf.xacro
```

El modelo del robot es el siguiente. Es solo un `link` con la forma de un cubo.

```
<?xml version="1.0"?>
<robot name="cube" xmlns:xacro="http://www.ros.org/wiki/xacro">
  <link name="base_link">
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <box size="0.2 0.2 0.2"/>
      </geometry>
    </visual>
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <box size="0.2 0.2 0.2"/>
      </geometry>
    </collision>
    <inertial>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <mass value="100"/>
      <inertia
        ixx="1.0" ixy="0.0" ixz="0.0"
        iyy="1.0" iyz="0.0"
        izz="1.0"/>
    </inertial>
  </link>
</robot>
```

Utilizamos un *launch file* para añadir el robot a la simulación. Vamos a escribir este archivo:

```
$ cd spawn_simple_object/launch
$ touch spawn_model.launch.py
```

- Empezamos importando los módulos relevantes

```
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import Node
from launch.substitutions import PathJoinSubstitution
from ament_index_python.packages import get_package_share_directory
from launch.substitutions import Command
```

```
from launch_ros.substitutions import FindPackageShare
```

- Recupera el archivo del modelo del robot (el xacro)

```
def generate_launch_description():

    ld = LaunchDescription()
    xacro_path = 'urdf/cube.urdf.xacro'

    robot_description = PathJoinSubstitution([
        get_package_share_directory('spawn_simple_object'),
        xacro_path
    ])

    robot_state_publisher_node = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='screen',
        parameters=[{
            'robot_description':Command(['xacro ', robot_description])
        }]
    )
```

- Spawnear el robot a partir del tópico `robot_description`

```
    spawn_node = Node(package='ros_gz_sim', executable='create',
                arguments=[
                   '-name', 'cube',
                   '-x', '1',
                   '-y', '1',
                   '-z', '0.5',
                   '-r', '0',
                   '-p', '0',
                   '-Y', '3.14',
                   '-topic', '/robot_description'],
                output='screen')
```

- Añadimos el módulo para iniciar la simulación:

```
    ignition_gazebo_node = IncludeLaunchDescription( PythonLaunchDescriptionSource(
                [PathJoinSubstitution([FindPackageShare('ros_gz_sim'),
                                       'launch',
                                       'gz_sim.launch.py'])]),
                                        launch_arguments=[('gz_args', [' -r -v 4 empty.
    sdf'])])
```

- Instalamos los directorios del paquete con el `CMakeLists.txt`

```
install(DIRECTORY launch urdf DESTINATION share/${PROJECT_NAME})
```

Ahora puede compilar y cargar la ROS workspace antes de iniciar la simulación

```
$ colcon build --symlink-install
$ source install/setup.bash
$ ros2 launch spawn_simple_object spawn_model.launch.py
```

Example 3: Sensores

Complicaremos nuestro modelo añadiendo algunos sensores adicionales. En particular, una cámara estándar, una cámara de profundidad y un sensor lidar. Los sensores se añaden al mismo cubo que simulamos antes. Un parámetro de input a las *launch file* define qué sensor usar en la simulación. Este paquete utiliza los paquetes `realsense2-camera` y `realsense2-description` para simular el sensor de profundidad.

```
$ sudo apt-get install ros-humble-realsense2-camera
$ sudo apt-get install ros-humble-realsense2-description
```

Vamos a crear el nuevo paquete `gazebo_sensors`.

```
$ ros2 pkg create gazebo_sensors --dependencies xacro realsense2_description
```

Define el modelo (xacro): XACRO (XML Macros) es una herramienta en ROS que permite crear archivos URDF (Unified Robot Description Format) de forma más eficiente mediante el uso de macros. En lugar de repetir las mismas líneas de código en un archivo URDF, puedes utilizar macros XACRO para generar descripciones del robot de manera más modular y flexible.

Editamos el nuevo modelo

```
$ cd gazebo_sensors
$ mkdir urdf
$ touch urdf/cube_with_sensors.urdf.xacro
```

Aquí tenemos algunos parámetros para decidir qué sensor activar.

```
<?xml version="1.0"?>
<robot name="cube" xmlns:xacro="http://www.ros.org/wiki/xacro">
    <xacro:arg name="use_camera" default="false" />
    <xacro:arg name="use_depth"  default="false" />
    <xacro:arg name="use_lidar"  default="false" />
    <xacro:property name="use_camera" value="$(arg use_camera)" />
    <xacro:property name="use_depth" value="$(arg use_depth)" />
    <xacro:property name="use_lidar" value="$(arg use_lidar)" />
```

Definimos el `base_link` y el `sensor_link`, conectado con un `fixed joint`. Los sensores estarán conectados con el `sensor_link`

```
<link name="base_link">
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <box size="0.2 0.2 0.2"/>
        </geometry>
    </visual>
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <box size="0.2 0.2 0.2"/>
        </geometry>
    </collision>
    <inertial>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <mass value="100"/>
        <inertia
        ixx="1.0" ixy="0.0" ixz="0.0"
        iyy="1.0" iyz="0.0"
        izz="1.0"/>
    </inertial>
```

```
</link>

<link name="sensor_link">
    <inertial>
    <mass value="0.1"/>
    <origin xyz="0 0 0"/>
    <inertia ixx="0.01" ixy="0.0" ixz="0.0" iyy="0.01" iyz="0.0" izz="0.01"/>
</inertial>
<visual>
    <geometry>
        <cylinder length="0.1" radius="0.02"/>
    </geometry>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <material name="black"/>
</visual>
<collision>
    <geometry>
        <cylinder length="0.1" radius="0.02"/>
    </geometry>
</collision>
</link>

<joint name="base_to_sensor" type="fixed">
    <parent link="base_link"/>
    <child link="sensor_link"/>
    <origin xyz="0 0 0.2" rpy="0.0 0.0 3.1415"/>
</joint>
```

Si el parámetro `use_camera` está `True`, añade el sensor de la cámara.

```
<xacro:if value="${use_camera}">
    <gazebo reference="sensor_link">
        <sensor name="cube_camera" type="camera">
            <always_on>1</always_on>
            <update_rate>30</update_rate>
            <visualize>1</visualize>
            <pose>0 0.0175 0.5125 0 -0 0</pose>
            <topic>/cube_camera/image_raw</topic>
            <camera name="camera">
                <horizontal_fov>1.21126</horizontal_fov>
                <image>
                    <width>640</width>
                    <height>480</height>
                    <format>RGB_INT8</format>
                </image>
                <clip>
                    <near>0.1</near>
                    <far>100</far>
                </clip>
                <noise>
                    <type>gaussian</type>
                    <mean>0</mean>
                    <stddev>0.007</stddev>
                </noise>
            </camera>
        </sensor>
    </gazebo>
</xacro:if>
```

Si el parámetro `use_depth` está `True`, añade el sensor de profundidad.

```
<xacro:if value="${use_depth}">
    <gazebo reference="base_link">
        <xacro:include filename="$(find realsense2_description)/urdf/_d435.urdf.xacro"/>
        <xacro:sensor_d435 parent="base_link" use_nominal_extrinsics="true" add_plug="false"
     use_mesh="true">
            <origin xyz="0 0 0.5" rpy="0 0 0"/>
        </xacro:sensor_d435>

        <sensor name='d435_depth' type='depth_camera'>
            <ignition_frame_id>camera_link</ignition_frame_id>
            <always_on>1</always_on>
            <update_rate>90</update_rate>
            <visualize>1</visualize>
            <topic>/cube_depth/image_raw</topic>
            <pose>0 0.0175 0.0 0 -0 0</pose>
            <camera name='d435'>
                <ignition_frame_id>camera_link</ignition_frame_id>
                <horizontal_fov>1.48702</horizontal_fov>
                <image>
                    <width>1280</width>
                    <height>720</height>
                </image>
                <clip>
                    <near>0.1</near>
                    <far>100</far>
                </clip>
                <noise>
                    <type>gaussian</type>
                    <mean>0</mean>
                    <stddev>0.1</stddev>
                </noise>
            </camera>
        </sensor>
    </gazebo>
</xacro:if>
```

Si el parámetro `use_lidar` está `True`, añade el sensor LIDAR

```
<xacro:if value="${use_lidar}">
    <gazebo reference="sensor_link">
        <sensor name="gpu_lidar" type="gpu_lidar">
            <pose>-0.064 0 0.121 0 0 0</pose>
            <topic>/cube/scan</topic>
            <ignition_frame_id>base_scan</ignition_frame_id>
            <update_rate>5</update_rate>
            <ray>
                <scan>
                    <horizontal>
                        <samples>360</samples>
                        <resolution>1</resolution>
                        <min_angle>0</min_angle>
                        <max_angle>6.28</max_angle>
                    </horizontal>
                </scan>
                <range>
                    <min>0.120000</min>
                    <max>20.0</max>
                    <resolution>0.015000</resolution>
                </range>
                <noise>
                    <type>gaussian</type>
                    <mean>0.0</mean>
```

```
                    <stddev>0.01</stddev>
                </noise>
            </ray>
            <always_on>true</always_on>
            <visualize>1</visualize>
        </sensor>
    </gazebo>
</xacro:if>
```

Añade el plugin para los sensores.

En esta nueva versión de Gazebo, tenemos un plugin para todos los sensores. Recuerda que en el Gazebo clásico, cada sensor tiene su propio plugin.

```
<xacro:if value="${use_camera or use_depth or use_lidar}">
    <gazebo>
        <plugin filename="libignition-gazebo-sensors-system.so"
        name="ignition::gazebo::systems::Sensors">
            <ignition_frame_id>camera_link</ignition_frame_id>
            <render_engine>ogre</render_engine>
        </plugin>
    </gazebo>
</xacro:if>
</robot>
```

Creamos el `launch file`.

```
$ mkdir launch
$ touch launch/cube_with_sensors.launch.py
```

- Importamos los módulos relevantes

```
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import Node
from launch.substitutions import PathJoinSubstitution
from ament_index_python.packages import get_package_share_directory
from launch.substitutions import Command
from launch_ros.substitutions import FindPackageShare
import xacro
import os
from pathlib import Path
```

- Añadimos el modelo de descripción del robot, inicializando con los parámetros. En el primer caso, `use_camera` está `True`.

```
def generate_launch_description():
    ld = LaunchDescription()
    xacro_path = 'urdf/cube_with_sensors.urdf.xacro'

    robot_description = xacro.process_file(Path(os.path.join(
    get_package_share_directory('gazebo_sensors'), xacro_path)), mappings={'use_camera'
    : "True", 'use_depth': "False", 'use_lidar': "False"})
```

- Define el nodo de publicación del `robot_state_publisher`

```
    robot_state_publisher_node = Node(
        package='robot_state_publisher',
```

```
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='screen',
        parameters=[{
            'robot_description':robot_description.toxml()
        }]
    )
```

- Vamos a spawnar el robot. Se utiliza el tema `robot_description`.

```
    spawn_node = Node(package='ros_gz_sim', executable='create',
                arguments=[
                   '-name', 'cube_with_sensors',
                   '-x', '1',
                   '-y', '1',
                   '-z', '0.5',
                   '-r', '0',
                   '-p', '0',
                   '-Y', '3.14',
                   '-topic', '/robot_description'],
                output='screen')
```

- El nodo para iniciar la simulación

```
    ignition_gazebo_node = IncludeLaunchDescription(PythonLaunchDescriptionSource(
        [PathJoinSubstitution([FindPackageShare('ros_gz_sim'),
                               'launch',
                               'gz_sim.launch.py'])]),
                               launch_arguments=[('gz_args', [' -r -v 4 empty.sdf'])])

    ld.add_action(robot_state_publisher_node)
    ld.add_action(spawn_node)
    ld.add_action(ignition_gazebo_node)
    return ld
```

- Instalamos los directorios del paquete con el `CMakeLists.txt`

```
install(DIRECTORY launch urdf DESTINATION share/${PROJECT_NAME})
```

Example 4: ROS2 Bridge

Los datos en Gazebo no son útiles si no podemos utilizarlos mediante ROS. Para transferir los datos de Gazebo a ROS 2 debemos utilizar el `ros_gz_bridge`. El `bridge` creará un puente que permite el intercambio de mensajes entre ROS y Gazebo. Su compatibilidad está limitada a determinados tipos de mensajes. No todos son compatibles ahora.

El `ros_gz_bridge` es una herramienta que permite la integración entre ROS 2 e Ignition Gazebo (anteriormente conocido como Gazebo Classic). Este puente actúa como un intermediario para facilitar la comunicación entre ambos sistemas, permitiendo el intercambio de mensajes y servicios entre ROS 2 e Ignition Gazebo. Es una herramienta clave cuando se simulan robots en Gazebo y se requiere controlarlos o monitorearlos a través de ROS 2, dado que ambos utilizan diferentes formatos de mensajes. Sus caracteristicas son:

- **Traducción de mensajes**: ROS 2 utiliza sus propios tipos de mensajes (por ejemplo, `std_msgs`, `sensor_msgs`, etc.), mientras que Gazebo utiliza tipos propios de mensajes. El `ros_gz_bridge` se encarga de traducir estos tipos para que puedan ser intercambiados entre los dos sistemas. Esto permite que los datos de sensores simulados en Gazebo sean accesibles desde ROS 2, y viceversa.

- **Compatibilidad con tópicos y servicios**: El puente permite el intercambio de tópicos y servicios entre ROS 2 y Gazebo, lo que significa que los nodos de ROS 2 pueden interactuar con sensores y actuadores simulados en Gazebo.
- **Bidireccionalidad**: El puente puede configurarse para permitir que la comunicación entre ROS 2 y Gazebo sea en ambos sentidos. De esta manera, ROS 2 puede tanto enviar mensajes a Gazebo como recibir mensajes desde Gazebo.
- **Especificación de tipos de mensajes**: El `ros_gz_bridge` permite configurar qué tipos de mensajes se desean compartir entre ROS 2 y Gazebo usando una sintaxis específica que delimita el tópico y los tipos de mensajes de cada sistema.

Supongamos que tenemos un sensor de cámara en Gazebo y queremos recibir las imágenes en ROS 2. El proceso sería el siguiente:

1. Se inicia Gazebo con un modelo de robot o sensor que simula una cámara.
2. Se utiliza `ros_gz_bridge` para crear el puente entre Gazebo y ROS 2.

El comando para crear el puente sería algo como:

```
ros2 run ros_gz_bridge parameter_bridge /camera/image@sensor_msgs/msg/Image@ignition.msgs.
    Image
```

En este caso:

- `/camera/image` es el nombre del tópico donde se publican las imágenes en Gazebo.
- `sensor_msgs/msg/Image` es el tipo de mensaje en ROS 2.
- `ignition.msgs.Image` es el tipo de mensaje en Gazebo.

Este puente permitirá que cualquier nodo en ROS 2 pueda suscribirse al tópico `/camera/image` y recibir imágenes directamente desde la simulación en Gazebo.

El `ros_gz_bridge` admite tres modos para la transmisión de mensajes:

- **Bidireccional** (`@`): ROS 2 y Gazebo pueden enviar y recibir mensajes a través del puente.
- **Unidireccional desde Gazebo a ROS 2** (`[`): Los mensajes fluyen solo desde Gazebo hacia ROS 2.
- **Unidireccional desde ROS 2 a Gazebo** (`]`): Los mensajes fluyen solo desde ROS 2 hacia Gazebo.

Las ventajas del `ros_gz_bridge` en este caso es:

- **Simulación avanzada**: Facilita el uso de Gazebo como simulador de robots en proyectos ROS 2, permitiendo aprovechar las capacidades avanzadas de simulación física y de sensores que ofrece Gazebo.
- **Integración eficiente**: Permite utilizar Gazebo para la simulación y ROS 2 para el control de robots, manteniendo la flexibilidad y eficiencia en el desarrollo de proyectos robóticos.

En resumen, el `ros_gz_bridge` es un componente esencial para los proyectos de robótica que usan Gazebo como simulador y ROS 2 como plataforma de control, proporcionando una conexión fluida entre ambos sistemas.

De este modo, el `bridge` se puede utilizar de dos formas:

- Command line ROS program
- Launch file

La sintaxis es

```
$ ros2 run ros_gz_bridge parameter_bridge /TOPIC@ROS_MSG@GZ_MSG
```

El comando `ros2 run _ros_gz_bridge_ _parameter_bridge_` simplemente ejecuta el código `parameter_bridge` del paquete `ros_gz_bridge`. Luego, especificamos nuestro nombre de tópico `/TOPIC` utilizado para transmitir los mensajes. El primer símbolo `@` delimita el nombre del tema de los tipos de mensajes. Después del primer símbolo `@` se encuentra el tipo de mensaje de ROS.

El tipo de mensaje ROS va seguido de un símbolo `@`, `[` o `]` donde:

- `@` es un *bridge* bidireccional.
- `[` es un *bridge* desde Gazebo hacia ROS.
- `]` es un *bridge* desde ROS hacia Ignition.

Un caso simple: Publicar pulsaciones de teclas en ROS

- Después de haber iniciado Gazebo, anda al plugin *Key Publisher*:
  - Haz clic en los 3 puntos verticales − > busca y selecciona el plugin *Key Publisher*.
- Obtienes el tipo de mensaje Gazebo con este comando

```
$ ign topic -i -t /keyboard/keypress
Publishers [Address, Message Type]:
tcp://172.18.0.1:37005, ignition.msgs.Int32
```

- Inicia el `bridge`

```
$ ros2 run ros_gz_bridge parameter_bridge /keyboard/keypress@std_msgs/msg/Int32[
    ignition.msgs.Int32
```

- Escucha el mensaje en ROS

```
$ ros2 topic echo /keyboard/keypress
```

Bridge the Sensor data

Vamos a crear un *launch file* que inicializa los tres *bridges* para los tres sensores.

1. Inicia la simulación y obtén la información sobre los tópicos que queremos llevar a ROS 2

```
$ ros2 launch gazebo_sensors cube_with_sensors.launch.py
```

2. Verifica los tipos de mensajes de los tópicos:
   - `/cube_camera/camera_info` − > `CameraInfo`

```
$ ign topic -i -t /cube_camera/camera_info
Publishers [Address, Message Type]:
tcp://172.18.0.1:42637, ignition.msgs.CameraInfo
```

- /cube_camera/image_raw – > Image

```
$ ign topic -i -t /cube_camera/image_raw
Publishers [Address, Message Type]:
tcp://172.18.0.1:42637, ignition.msgs.Image
```

- /cube_depth/image_raw/points – >PointCloudPacked

```
$ ign topic -i -t /cube_depth/image_raw/points
Publishers [Address, Message Type]:
tcp://172.18.0.1:42637, ignition.msgs.PointCloudPacked
```

- /lidar – > LaserScan

```
$ ign topic -i -t /lidar
Publishers [Address, Message Type]:
tcp://172.18.0.1:42637, ignition.msgs.LaserScan
```

3. Creamos un *launch file* para iniciar el *bridge*: gazebo_bridge.launch.py

```
from launch import LaunchDescription
from launch_ros.actions import Node
def generate_launch_description():
    bridge = Node(
        package='ros_gz_bridge',
        executable='parameter_bridge',
        arguments=[
            # Lidar (IGN -> ROS2)
            '/cube/scan@sensor_msgs/msg/LaserScan[ignition.msgs.LaserScan',
            # Camera (IGN -> ROS2)
            '/cube_camera/image_raw@sensor_msgs/msg/Image[ignition.msgs.Image',
            '/cube_camera/camera_info@sensor_msgs/msg/CameraInfo[ignition.msgs.
    CameraInfo',
            # Depth (PointClouds)
            '/cube_depth/image_raw/points@sensor_msgs/msg/PointCloud2[ignition.msgs.
    PointCloudPacked'],
            output='screen'
    )

    remappings=[
        ("/lidar", "/cube/lidar"),
    ],
    output='screen'

    return LaunchDescription([bridge])

```

4. Necesitamos publicar manualmente la transformación del *output* del sensor al *base frame* de la cámara de profundidad y del LIDAR.

```
depth_cam_data2sensor_link = Node(package='tf2_ros',
    executable='static_transform_publisher',
    name='cam3Tolink',
    output='log',
    arguments=['0.0', '0.0', '0.0', '0.0', '0.0', '0.0', 'sensor_link', '
    cube_with_sensors/base_link/d435_depth'])
```

```
lidar2sensor_link = Node(package='tf2_ros',
    executable='static_transform_publisher',
    name='cam3Tolink',
    output='log',
    arguments=['0.0', '0.0', '0.0', '0.0', '0.0', '0.0', 'sensor_link', 'base_scan'])

return LaunchDescription([bridge, depth_cam_data2sensor_link, lidar2sensor_link])
```

Ahora podemos lanzar la simulación.

```
$ colcon build --symlink-install
$ source install/setup.bash
$ ros2 launch gazebo_sensors cube_with_sensors.launch.py
$ ros2 launch gazebo_sensors gazebo_bridge.launch.py
```

Ahora puedes usar `rqt` para visualizar los datos. En el Docker ya está todo instalado. Sin embargo, puedes instalar lo que necesitas con estos comandos:

- `rqt`

```
$ sudo apt-get install ros-humble-rqt-*
$ ros2 run rqt-image-view rqt-image-view
```

- `rviz2`

  Con `rviz2` es posible visualizar el LIDAR y los `PointClouds`, simplemente añadiendo el plugin adecuado utilizando el menú.

Example 5: Differential drive robot

Comencemos a implementar nuestro primer robot móvil. Esto también permitirá empezar con los controladores de robots.

Vamos a crear un robot con 2 ruedas pasivas y 2 ruedas activas.

```
$ ros2 pkg create diff_drive_description --dependencies xacro
$ mkdir diff_drive_description/urdf
$ mkdir diff_drive_description/launch
```

1. Crea dos archivos: uno para contener los macros y uno para el `xacro` principal.

```
    $ touch diff_drive_description/urdf/diff_drive.urdf.xacro
    $ touch diff_drive_description/urdf/diff_drive_macro.xacro

```

2. Edita el `diff_drive_macro.xacro`

```
<?xml version="1.0"?>
<robot name="diff_robot" xmlns:xacro="http://www.ros.org/wiki/xacro">
    <xacro:property name="M_PI" value="3.1415926535897931" />

    <material name="Black">
      <color rgba="0.0 0.0 0.0 1.0"/>
    </material>
```

```
    <material name="White">
      <color rgba="1.0 1.0 1.0 1.0"/>
    </material>

    <xacro:macro name="cylinder_inertia" params="m r h">
        <inertia  ixx="${m*(3*r*r+h*h)/12}" ixy = "0" ixz = "0"
                iyy="${m*(3*r*r+h*h)/12}" iyz = "0"
                izz="${m*r*r/2}" />
    </xacro:macro>

```

3. Define el macro de las ruedas pasivas:

```
    <xacro:macro name="passive_wheel_joint" params="name parent child *origin">
      <joint name="${name}" type="fixed" >
        <parent link="${parent}" />
        <child link="${child}" />
        <xacro:insert_block name="origin" />
      </joint>
    </xacro:macro>

```

4. Define el macro de los enlaces de las ruedas:

```
    <xacro:macro name="passive_wheel_link" params="name *origin">
        <link name="${name}">
            <visual>
                <xacro:insert_block name="origin" />
                <geometry>
                    <sphere radius="${passive_wheel_radius}" />
                </geometry>
                <material name="Black" />
            </visual>
            <collision>
                <geometry>
                    <sphere radius="${passive_wheel_radius}" />
                </geometry>
                <origin xyz="0 0.02 0" rpy="${M_PI/2} 0 0" />
            </collision>
            <inertial>
                <mass value="${passive_wheel_mass}" />
                    <origin xyz="0 0 0" />
                    <inertia  ixx="0.001" ixy="0.0" ixz="0.0"
                            iyy="0.001" iyz="0.0"
                                izz="0.001" />
            </inertial>
        </link>
    </xacro:macro>

```

5. Define el macro de las ruedas activas:

```
    <xacro:macro name="wheel" params="side parent translateX translateY">
      <link name="${side}_wheel">
        <visual>
          <origin xyz="0 0 0" rpy="${M_PI/2} 0  0 " />
          <geometry>
            <cylinder length="${wheel_height}" radius="${wheel_radius}" />
          </geometry>
                <material name="Black" />
```

```
        </visual>
        <collision>
          <origin xyz="0 0 0" rpy="${M_PI/2} 0 0 " />
          <geometry>
            <cylinder length="${wheel_height}" radius="${wheel_radius}" />
          </geometry>
        </collision>
        <inertial>
          <mass value="${wheel_mass}" />
          <origin xyz="0 0 0" />
          <inertia ixx="0.001" ixy="0.0" ixz="0.0"
                   iyy="0.001" iyz="0.0"
                   izz="0.001" />
        </inertial>
      </link>

      <joint name="${side}_wheel_joint" type="continuous">
        <parent link="${parent}"/>
        <child link="${side}_wheel"/>

          <origin xyz="0 ${translateY} 0" rpy="0 0 0" />
        <axis xyz="0 1 0" rpy="0 0 0" />
        <limit effort="100" velocity="100"/>
        <joint_properties damping="0.0" friction="0.0"/>
      </joint>

    </xacro:macro>

```

6. Define la estructura del robot en el `xacro` principal:

```
<?xml version="1.0"?>

<robot name="diff_robot" xmlns:xacro="http://ros.org/wiki/xacro">
<xacro:include filename="$(find diff_drive_description)/urdf/diff_drive_macro.xacro" />

<xacro:property name="base_radius" value="0.15" />
<xacro:property name="passive_wheel_height" value="0.04" />
<xacro:property name="passive_wheel_radius" value="0.025" />
<xacro:property name="passive_wheel_mass" value="0.5" />
<xacro:property name="wheel_radius" value="0.039" />
<xacro:property name="wheel_height" value="0.02" />
<xacro:property name="wheel_mass" value="2.5" />

<link name="base_link">
    <inertial>
        <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
        <mass value="5" />
        <origin xyz="0 0 0" />
        <cylinder_inertia m="5" r="${base_radius}" h="0.02" />
    </inertial>
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>
            <cylinder length="0.02" radius="${base_radius}" />
        </geometry>
        <material name="White" />
    </visual>
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0 " />
        <geometry>
```

```
            <cylinder length="0.02" radius="${base_radius}" />
        </geometry>
    </collision>
</link>

<xacro:passive_wheel_joint name="passive_wheel_front_joint"
    parent="base_link"
    child="passive_wheel_front_link">
    <origin xyz="0.115 0.0 0.007" rpy="${-M_PI/2} 0 0"/>
</xacro:passive_wheel_joint>

<xacro:passive_wheel_link name="passive_wheel_front_link">
    <origin xyz="0 0.02 0" rpy="${M_PI/2} 0 0" />
</xacro:passive_wheel_link>

<xacro:passive_wheel_joint
    name="passive_wheel_back_joint"
    parent="base_link"
    child="passive_wheel_back_link">
    <origin xyz="-0.135 0.0 0.009" rpy="${-M_PI/2} 0 0"/>
</xacro:passive_wheel_joint>

<xacro:passive_wheel_link
    name="passive_wheel_back_link">
    <origin xyz="0.02 0.02 0 " rpy="${M_PI/2} 0 0" />
</xacro:passive_wheel_link>

<xacro:wheel side="right" parent="base_link" translateX="0" translateY="-${base_radius}
    " />
<xacro:wheel side="left" parent="base_link" translateX="0" translateY="${base_radius}"
    />

<link name="lidar_link">
    <visual>
        <origin xyz="0 0 0.05" rpy="0 0 0"/>
        <geometry>
            <box size="0.05 0.05 0.05" />
        </geometry>
    </visual>
    <collision>
        <origin xyz="0 0 0.05" rpy="0 0 0"/>
        <geometry>
            <box size="0.05 0.05 0.05" />
        </geometry>
    </collision>
    <inertial>
        <origin xyz="0 0 0.05" rpy="0 0 0"/>
        <mass value="1"/>
        <inertia
        ixx="1.0" ixy="0.0" ixz="0.0"
        iyy="1.0" iyz="0.0"
        izz="1.0"/>
    </inertial>
</link>

<joint name='lidar_sensor_joint' type='fixed'>
    <parent link="base_link"/>
    <child link="lidar_link"/>
</joint>

<gazebo>
```

```
    <plugin filename="libignition-gazebo-sensors-system.so" name="ignition::gazebo::
    systems::Sensors">
        <render_engine>ogre2</render_engine>
    </plugin>
    <plugin filename="libignition-gazebo-diff-drive-system.so" name="ignition::gazebo::
    systems::DiffDrive">
        <left_joint>left_wheel_joint</left_joint>
        <right_joint>right_wheel_joint</right_joint>
        <wheel_separation>${2*base_radius}</wheel_separation>
        <wheel_radius>${wheel_radius}</wheel_radius>
        <odom_publish_frequency>10</odom_publish_frequency>
        <topic>cmd_vel</topic>
        <robotBaseFrame>base_link</robotBaseFrame>
    </plugin>
</gazebo>
</robot>

```

7. Vamos a crear un `launch file`

```
$ touch launch/diff_drive.launch.py

```

8. Importa los módulos

```
from launch import LaunchDescription
from launch_ros.actions import Node
import launch_ros.descriptions
from launch.substitutions import Command
from launch.substitutions import PathJoinSubstitution
from ament_index_python.packages import get_package_share_directory
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.substitutions import FindPackageShare

def generate_launch_description():

```

9. Carga las descripciones del robot

```
    xacro_path = 'urdf/diff_drive.urdf.xacro'

    robot_description = PathJoinSubstitution([
        get_package_share_directory('diff_drive_description'),
        xacro_path
    ])

    robot_state_publisher_node = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='screen',
        parameters=[{
            'robot_description':Command(['xacro ', robot_description])
        }]
    )

```

10. Vamos a *spawn* el robot en Gazebo

```
    spawn_node = Node(package='ros_gz_sim', executable='create',
                  arguments=[
                     '-name', 'diff_drive',
                     '-x', '0',
                     '-y', '0',
                     '-z', '0.1',
                     '-r', '0',
                     '-p', '0',
                     '-Y', '0',
                     '-topic', '/robot_description'],
                  output='screen')

```

11. Inicia Gazebo

```
    ignition_gazebo_node = IncludeLaunchDescription(PythonLaunchDescriptionSource(
                [PathJoinSubstitution([FindPackageShare('ros_gz_sim'),
                                       'launch',
                                       'gz_sim.launch.py'])]),
                                       launch_arguments=[('gz_args', [' -r -v 4 empty.
    sdf'])])

```

12. Creamos el *bridge*

```
    bridge = Node(
        package='ros_gz_bridge',
        executable='parameter_bridge',
        arguments=['lidar@sensor_msgs/msg/LaserScan@gz.msgs.LaserScan',
                '/lidar/points@sensor_msgs/msg/PointCloud2@gz.msgs.PointCloudPacked',
                'cmd_vel@geometry_msgs/msg/Twist]gz.msgs.Twist',
                '/model/diff_drive/odometry@nav_msgs/msg/Odometry@gz.msgs.Odometry'],
        output='screen'
    )
    return LaunchDescription([
        robot_state_publisher_node,
        spawn_node,
        ignition_gazebo_node,
        bridge
    ])

```

13. Modifica el `CMakeLists.txt`

```
install(DIRECTORY launch urdf DESTINATION share/${PROJECT_NAME})

```

14. Compila la *workspace* y lanza la simulación

```
$ colcon build --symlink-install
$ source install/setup.bash
$ ros2 launch diff_drive_description diff_drive.launch.py

```

15. Controla el robot usando `rqt`

```
$ rqt

```

Example 6: Gazebo plugin

En este ejemplo vamos a desarrollar *plugins* personalizados para Gazebo. Un plugin es un componente de software que añade funcionalidades adicionales a la simulación en Gazebo. Los plugins permiten que el robot simulado se comporte de manera más realista o añaden capacidades como el control de movimiento, la detección de colisiones, o la simulación de sensores como cámaras o LiDAR.

En principio, vamos a desarrollar un *plugin* que se añadirá a la configuración del mundo de Gazebo o directamente al modelo del robot (como se hizo con el `differential drive plugin`).

Un *plugin* de Gazebo permitirá acceder directamente al modelo simulado y crear algoritmos de control más rápidos y performantes. Vamos a ver tres ejemplos:

- Un *plugin* independiente para comprender la estructura de un *plugin*.
- Uno que integra ROS 2 y Gazebo.
- Un *plugin* más complejo para controlar el robot con transmisión diferencial.

Basic plugin

Este NO es un paquete ROS, entonces, tenemos que crear manualmente la estructura del paquete.

1. Crea el *plugin* `hello_world` en el *ROS workspace*.

```
$ mkdir hello_world && cd hello_world
$ touch HelloWorld.cpp
$ touch CMakeLists.txt
$ touch hello_world_plugin.sdf

```

2. Edita el `HelloWorld.cpp` para escribir un mensaje en la consola de Gazebo.

```
#include <string>
#include <gz/common/Console.hh>
#include <gz/plugin/Register.hh>
#include <gz/sim/System.hh>

namespace hello_world
{
    class HelloWorld:
        public gz::sim::System,
        public gz::sim::ISystemPostUpdate
    {
        public: void PostUpdate(const gz::sim::UpdateInfo &_info,
                const gz::sim::EntityComponentManager &_ecm) override;
    };
}
IGNITION_ADD_PLUGIN(hello_world::HelloWorld, gz::sim::System, hello_world::HelloWorld::
    ISystemPostUpdate)

void HelloWorld::PostUpdate(const gz::sim::UpdateInfo &_info, const gz::sim::
    EntityComponentManager &/*_ecm*/)
{
    std::string msg = "Hello, world! Simulation is ";
    if (!_info.paused)
        msg += "not ";
    msg += "paused.";
```

```
    ignmsg << msg << std::endl;
}

```

3. Modifica el `CMakeLists.txt` para compilar el *plugin*

```
cmake_minimum_required(VERSION 3.10.2 FATAL_ERROR)
find_package(ignition-cmake2 REQUIRED)
project(Hello_world)

ign_find_package(ignition-plugin1 REQUIRED COMPONENTS register)
set(IGN_PLUGIN_VER ${ignition-plugin1_VERSION_MAJOR})
ign_find_package(ignition-gazebo6 REQUIRED)
set(IGN_GAZEBO_VER ${ignition-gazebo6_VERSION_MAJOR})

add_library(HelloWorld SHARED HelloWorld.cpp)
set_property(TARGET HelloWorld PROPERTY CXX_STANDARD 17)
target_link_libraries(HelloWorld
  PRIVATE ignition-plugin${IGN_PLUGIN_VER}::ignition-plugin${IGN_PLUGIN_VER}
  PRIVATE ignition-gazebo${IGN_GAZEBO_VER}::ignition-gazebo${IGN_GAZEBO_VER})

```

4. Crea el archivo `sdf` para incluir el *plugin*.

```
<?xml version="1.0" ?>
<sdf version="1.6">
  <world name="default">
    <plugin filename="HelloWorld" name="hello_world::HelloWorld"></plugin>
  </world>
</sdf>

```

5. Compila el *plugin*.

```
$ cd hello_world
$ mkdir build
$ cd build
$ cmake ..
$ make

```

6. Descubre el *plugin* exportando la variable de entorno `GZ_SIM_SYSTEM_PLUGIN_PATH`

```
$ export GZ_SIM_SYSTEM_PLUGIN_PATH=`pwd`/build

```

7. Inicia la simulación

```
$ ign gazebo -v 3 hello_world_plugin.sdf

```

Y así, hemos completado los ejemplos básicos y avanzados para desarrollar e integrar *plugins* en Gazebo.

3. Información extra

3.1. Launchers

En ROS 2, los archivos de lanzamiento (*launch files*) permiten automatizar el arranque de múltiples nodos y procesos asociados a un sistema robótico. Existen tres formas principales de definir archivos de lanzamiento en ROS 2:

- **Python-based launch files**
- **XML-based launch files**
- **YAML-based launch files**

A continuación, se detallan las características y ejemplos de cada tipo.

3.1.1. Archivos de lanzamiento basados en Python

El formato más flexible para definir archivos de lanzamiento en ROS 2 es el basado en Python. Estos archivos permiten utilizar lógica de programación como condicionales, bucles y funciones, lo cual resulta muy útil para configuraciones complejas.

3.1.2. Características

- Permiten el uso de lógica de programación.
- Soportan construcciones avanzadas como condicionales y bucles.
- Se definen usando el módulo `launch` de ROS 2.

3.1.3. Ejemplo en Python

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='demo_nodes_cpp',
            executable='talker',
            name='talker_node',
            output='screen',
        ),
        Node(
            package='demo_nodes_cpp',
            executable='listener',
            name='listener_node',
            output='screen',
        ),
    ])
```

Este archivo de lanzamiento en Python lanza dos nodos: un `talker` y un `listener`, ambos del paquete `demo_nodes_cpp`.

3.1.4. Archivos de lanzamiento basados en XML

El formato XML es más antiguo y se utiliza principalmente para configuraciones más simples o en casos donde no se necesita lógica avanzada. Este formato es familiar para quienes ya han trabajado con los archivos de lanzamiento en ROS 1.

3.1.5. Características

- Fácil de leer y escribir para configuraciones simples.
- No permite lógica de programación avanzada.
- Utiliza una estructura fija basada en elementos XML.

3.1.6. Ejemplo en XML

```xml
<launch>
    <node
        pkg="demo_nodes_cpp"
        exec="talker"
        name="talker_node"
        output="screen">
    </node>

    <node
        pkg="demo_nodes_cpp"
        exec="listener"
        name="listener_node"
        output="screen">
    </node>
</launch>
```

Este archivo en XML realiza la misma tarea que el ejemplo en Python, lanzando un `talker` y un `listener`.

3.1.7. Archivos de lanzamiento basados en YAML

El formato YAML es relativamente nuevo en ROS 2 y ofrece una sintaxis más compacta. Sin embargo, al igual que en XML, no permite lógica avanzada, pero es ideal para configuraciones básicas que requieren simplicidad y brevedad.

3.1.8. Características

- Sintaxis simple y concisa.
- No permite lógica avanzada.
- Recomendado para configuraciones sencillas y cuando se busca reducir la cantidad de código.

3.1.9. Ejemplo en YAML

```yaml
launch:
  - node:
      pkg: demo_nodes_cpp
      exec: talker
      name: talker_node
      output: screen
  - node:
      pkg: demo_nodes_cpp
      exec: listener
      name: listener_node
      output: screen
```

En este archivo YAML, también se lanzan un `talker` y un `listener` de forma similar a los ejemplos anteriores.

3.1.10. Comparación de formatos

- **Python-based launch files**: recomendados para configuraciones complejas o dinámicas, ya que permiten lógica de programación avanzada.
- **XML-based launch files**: más adecuados para configuraciones simples o en proyectos heredados de ROS 1.
- **YAML-based launch files**: útiles para configuraciones sencillas donde la brevedad y simplicidad son deseadas.

En conclusión, dependiendo de la complejidad del proyecto y la flexibilidad que se necesite, ROS 2 permite utilizar diferentes tipos de archivos de lanzamiento para orquestar los nodos y procesos de manera eficiente.

3.2. URDF

URDF (Unified Robot Description Format) es un formato XML utilizado en ROS (Robot Operating System) para describir la configuración física de un robot. SDF (Simulation Description Format) es otro formato XML, más versátil y orientado a simulaciones, que se utiliza comúnmente en simuladores como Gazebo. A continuación se describen ambos formatos y sus principales características.

URDF (Formato de Descripción Unificada de Robots): Utilizado principalmente para la descripción estática y cinemática de robots en ROS.

- **Enlaces (Links)**: Representan los cuerpos rígidos del robot. Cada enlace puede tener propiedades visuales, de colisión e inerciales.
  - **Visual**: Cómo se visualiza el enlace, generalmente definido mediante una malla o una forma geométrica (caja, cilindro, esfera).
  - **Colisión**: La forma utilizada para las interacciones físicas, como las colisiones.
  - **Inercial**: Describe la masa e inercia del enlace, lo cual es importante para las simulaciones físicas.
- **Articulaciones (Joints)**: Definen cómo los enlaces están conectados y cómo pueden moverse unos con respecto a otros.
  - **Tipos de articulaciones**:
    - **Fija (Fixed)**: Los enlaces están conectados de manera rígida.

    - **Revoluta (Revolute)**: Permite movimiento rotacional alrededor de un eje.
    - **Continua (Continuous)**: Similar a la revoluta, pero sin límites.
    - **Prismática (Prismatic)**: Permite movimiento lineal a lo largo de un eje.
    - **Flotante (Floating)** y **Planar**: Permiten movimientos más complejos.
- **Propiedades de las articulaciones**: Incluyen límites (por ejemplo, rango de movimiento), amortiguamiento y fricción.

SDF (Formato de Descripción para Simulación): Es un formato más completo que URDF, utilizado para describir tanto robots como ambientes complejos en simuladores como Gazebo.

- **Ambientes y Físicas**: SDF no solo describe robots, sino que también permite modelar escenarios completos, incluyendo elementos del terreno, objetos estáticos, iluminación y física avanzada.
    - **Modelos complejos**: Permite definir estructuras robóticas más complejas, con múltiples sensores, cámaras y sistemas de física avanzados, como fricción dinámica y colisiones detalladas.
- **Simulación de sensores**: SDF permite una simulación más precisa de sensores como cámaras, LIDAR, GPS, entre otros, con configuraciones detalladas como rango de detección y resolución.
- **Dinámica y control**: Soporta definiciones detalladas de las propiedades dinámicas, como la resistencia al aire, la fricción y otros parámetros de la física que afectan el movimiento y control de los robots.

Ambos formatos tienen sus propios usos y propósitos. URDF es más simple y comúnmente utilizado para describir robots en ROS debido a su integración con el sistema, mientras que SDF está diseñado para simulaciones más detalladas y complejas en entornos como Gazebo.

Diferencias clave entre URDF y SDF:

- **URDF** es más adecuado para la descripción estructural de robots en aplicaciones de ROS, mientras que **SDF** es más potente para simulaciones físicas completas.
- SDF soporta una mayor gama de características físicas, como terrenos y ambientes complejos, que no son parte del objetivo de URDF.
- URDF tiene limitaciones en cuanto a la descripción de sensores y propiedades físicas avanzadas, mientras que SDF permite un control detallado sobre todos estos aspectos.

Finalmente, si bien ROS utiliza URDF para describir robots en su entorno, cuando se usa Gazebo para simulaciones, los modelos URDF pueden ser convertidos a SDF para aprovechar sus capacidades avanzadas de simulación.

.

Referencias

[1] Open Robotics. Gazebo fortress installation guide. https://gazebosim.org/docs/fortress/install/, 2024. Accessed: 2024-10-24.

[2] Open Robotics. Gazebo simulation integration tutorial for ros 2 humble. https://docs.ros.org/en/humble/Tutorials/Advanced/Simulators/Gazebo/Gazebo.html, 2024. Accessed: 2024-10-24.