Lab Setup

For this second lab you have **5 tasks** explained below. As you go along the document you have explanations and examples to help you accomplish them. You will need to provide your **source code**, **rosbags** and a **report (pdf or markdown)** with **screenshots** showing relevant result for each part alongside necessary comments. Try to keep the naming for each task as comprehensible as possible.

IMPORTANT

For each task please also include the commands you need to run to make each one work for easy checkup and grading.

Prerequisites and Workspace

Ubuntu22 + Ros 2 Humble

This code has been tested with that setup. However, it is possible it can work with other configurations.

DON'T FORGET TO ADD YOUR ROS_DOMAIN_ID CORRESPONDING TO YOU GROUP NUMBER

```
export ROS_DOMAIN_ID=<Group #>  # e.g., export ROS_DOMAIN_ID=5
```

Grading

The grading of this labs is the following:

- Grade 0: No tasks are delivered
- Grade 1: 1 task is presented
- Grade 2: 2 tasks are presented
- Grade 3: 3 tasks are presented
- Grade 4: 4 tasks are presented
- Grade 5: 5 tasks are presented

Install required packages

```
sudo apt update
sudo apt install ros-humble-turtlesim ros-humble-rqt* ros-humble-rviz2 \
    ros-humble-tf2-tools ros-humble-tf2-ros ros-humble-geometry-msgs \
    ros-humble-sensor-msgs ros-humble-nav-msgs ros-humble-gazebo-ros-pkgs \
    ros-humble-turtlebot3* ros-humble-image-tools
```

Create workspace

```
mkdir -p ~/ros2_lab2_ws/src
cd ~/ros2_lab2_ws
```

RQT Tools

RQT (Qt-based Robot User Interface) is a comprehensive framework that provides a plugin-based GUI for ROS2 systems. It helps with debugging, monitoring and interacting with ROS2 systems. It is modular because each tool can be run independently or combined in a single interface. It is important for the following:

- **System Debugging:** Quickly identify communication issues between nodes
- **Real-time Monitoring:** Watch data flow and system performance
- **Interactive Testing:** Call services and modify parameters without coding
- **Data Analysis:** Plot and analyze sensor data streams

RQT is a Qt-based framework for GUI development in ROS. It provides various tools for debugging and examining ROS systems.

Essential tools

```
rqt_graph
```

- Shows nodes, topics and their connections
- Different display modes
  - Nodes only
  - Nodes/Topics
- Real-time updates as the system changes

```
rqt_console
```

- View log messages from all nodes
- Filter by severity level, node name or message content
- It is essential for debugging

```
rqt_topic
```

- Monitor topic publication rates
- View message contents
- Plot numeric data over time

```
rqt_service_caller
```

- Call services interactively
- Useful for testing service interfaces

```
rqt_reconfigure
```

- Dynamically modify node parameters
- Real-time parameter tuning

TurtleSim Analysis

Now we will see a basic example of one of this tools. For it you need to launch the following in two separate terminals

```
ros2 run turtlesim turtlesim_node
ros2 run turtlesim turtle_teleop_key
```

You have two options:

Option 1:

In a third terminal launch:

```
rqt
```

You should have a graphical interface with the following message:

"**rqt** is a GUI framework that is able to load various plug-in tools as dockable windows. There are currently no plugins selected. To add plugins, select items from the **Plugins** menu. You may also save a particular arrangement of plugins as a *perspective* using the **Perspectives** menu."

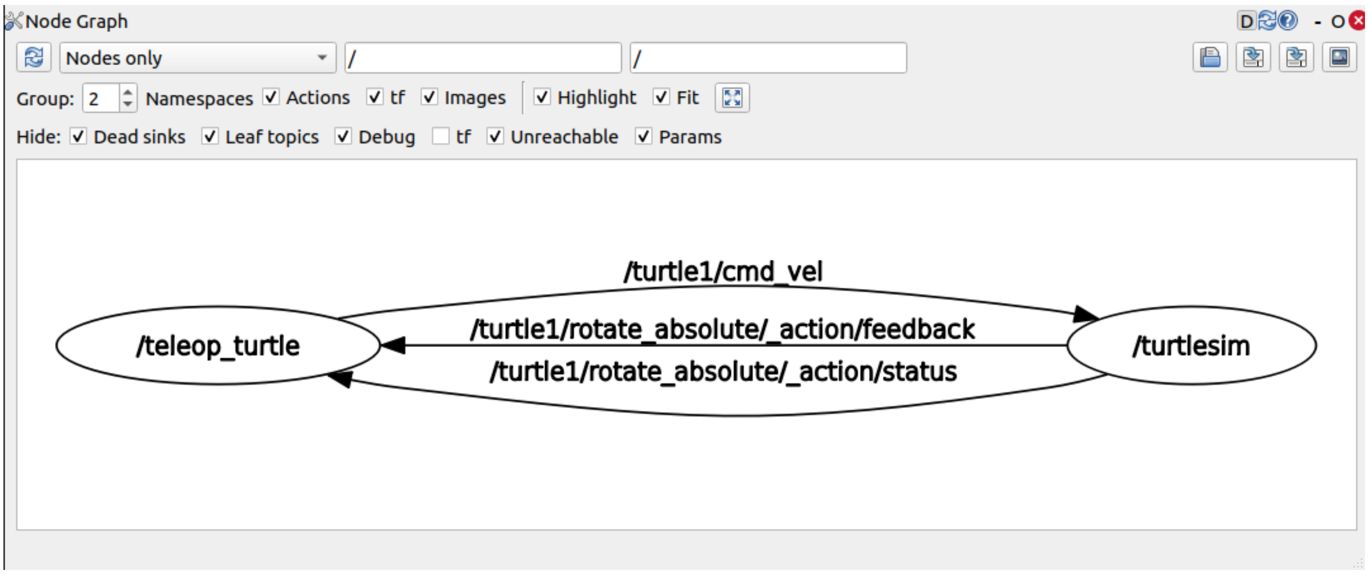
From here, choose plugins -> introspection -> node graph.

Option 2

In a terminal launch the following:

```
ros2 run rqt_graph rqt_graph
```

With either of the options you should get the following:



You may need to refresh the graph.

TASK 1

- From *rqt_graph* see what happens when you change the view from *Nodes only* to *Node/Topics*. Describe what are the differences and why.
- Explain the message flow of the system.
- Use rqt_topics and check the topic and its types. Which one is the one making the turtlebot move? What is the type of message
- Plot x and y positions of the turtlebot as you move the robot using the turtlesim turtle_teleop_key node

Include screenshots of all your steps.

RViz2 Visualization

ROS Visualization 2 or RViz2 is the primary 2D visualization tool for ROS2 systems. It helps to understand what their robots are "thinking" by providing a visual representation of sensor data, robot states, planning algorithms and more.

Some core components:

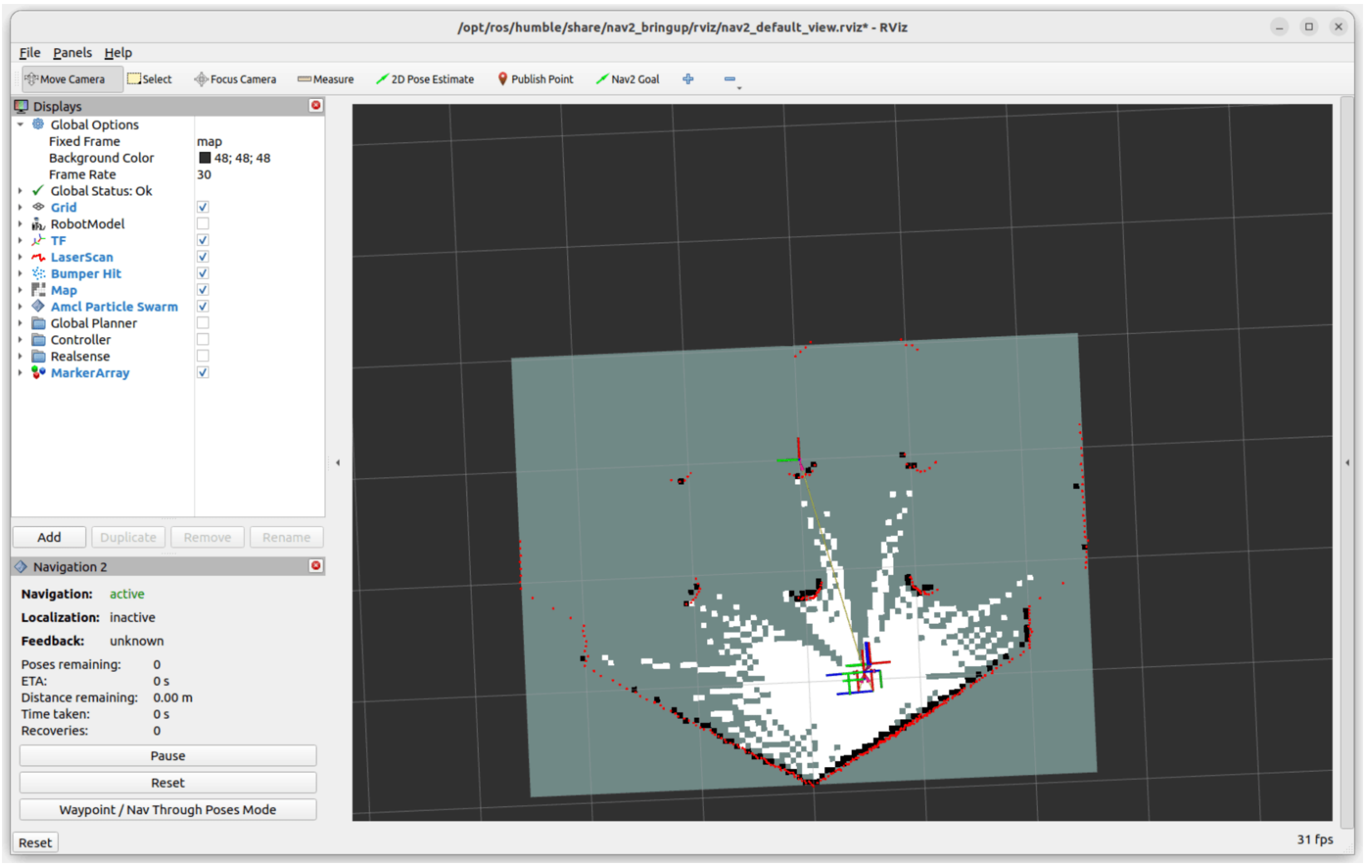
- **Displays:** plugins that render different types of data such as LaserScan, PointCloud, Images, etc.
- **Fixed Frame:** the coordinate frame that serves as the reference for all the different visualizations.
- **Target Frame:** the frame that the camera or sensor tracks.
- **Transforms:** rviz2 uses TF2 transforms to automatically transform data between different coordinate frames.

Common use cases include:

- **Robot Development:** to visualize the robot models and joint states

- **Sensor Integration:** to display camera images, LiDAR scans, point clouds
- **Navigation:** to show planned paths, obstacles, and localization
- **Algorithm Development:** to visualize intermediate results and debug algorithms

For example, for navigation tool map creation, you can see the parts of the map the robot know from its laser information:



/opt/ros/humble/share/nav2_bringup/rviz/nav2_default_view.rviz* - RViz

Global Options
Fixed Frame map
Background Color 48; 48; 48
Frame Rate 30

Navigation: active
Localization: inactive
Feedback: unknown
Poses remaining: 0
ETA: 0 s
Distance remaining: 0.00 m
Time taken: 0 s
Recoveries: 0

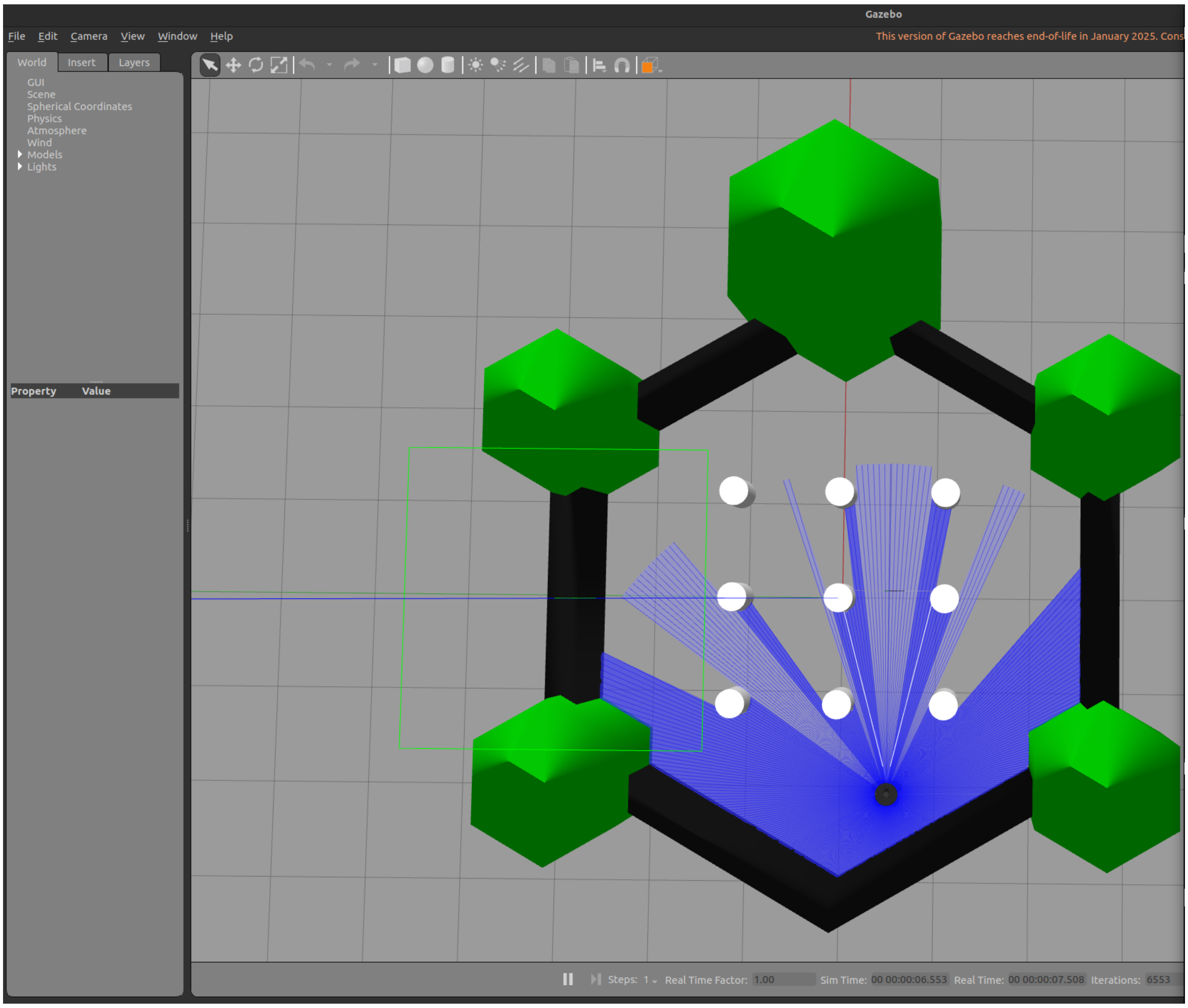
https://roboticsbackend.com/ros2-nav2-generate-a-map-with-slam_toolbox/

Basic example

In a terminal, launch the following:

```
export TURTLEBOT3_MODEL=burger
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

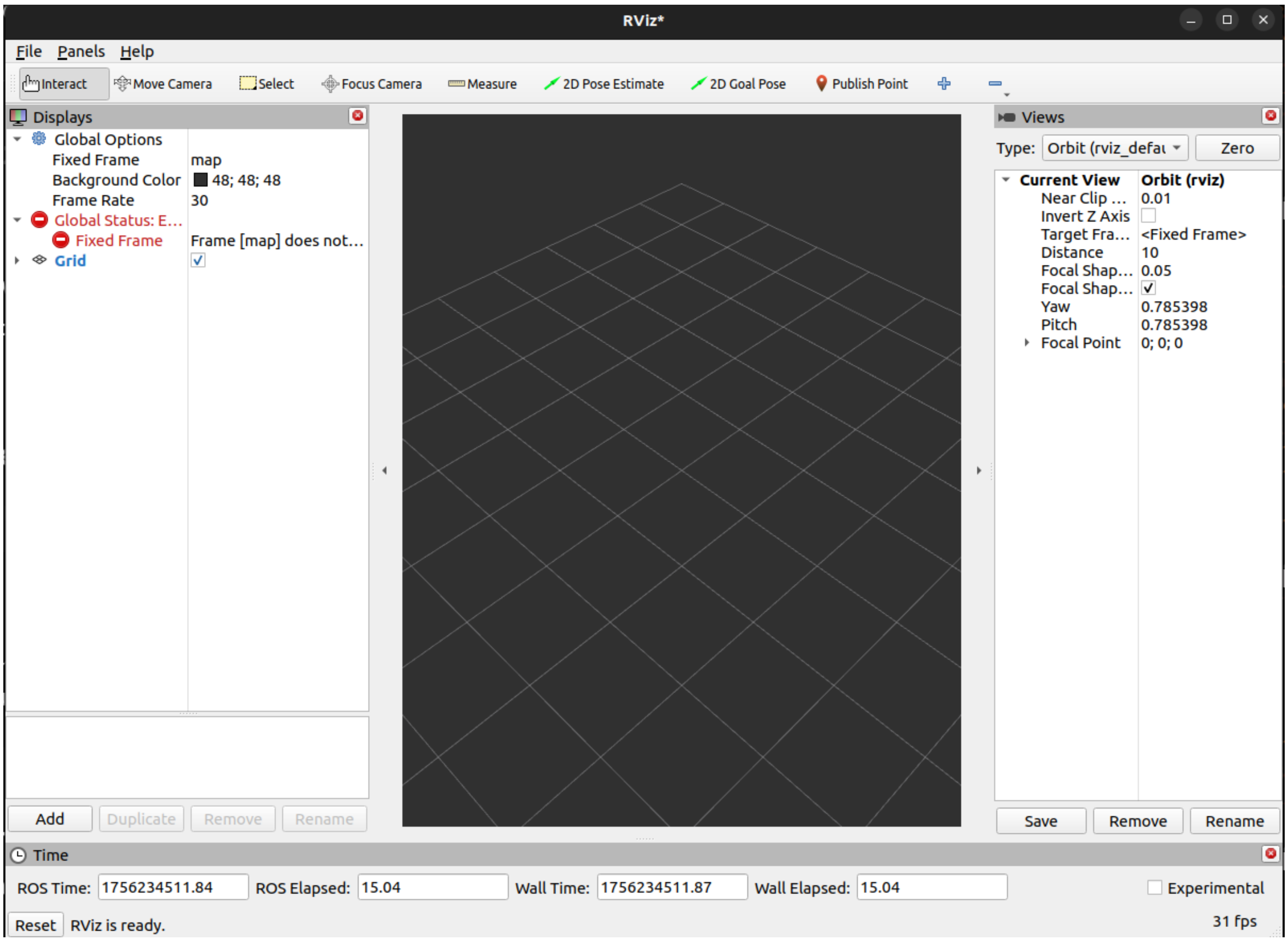
You should get the Gazebo 3d visualization with a map and a turtlebot3 robot as the image below shows. The model loaded for the turtlebot3 is going to be *burger*, other options are *waffle* and *waffle_pi*.


Gazebo
File Edit Camera View Window Help
This version of Gazebo reaches end-of-life in January 2025. Cons
World Insert Layers
GUI
Scene
Spherical Coordinates
Physics
Atmosphere
Wind
Models
Lights
Property Value
Steps: 1 Real Time Factor: 1.00 Sim Time: 00 00:00:06.553 Real Time: 00 00:00:07.508 Iterations: 6553

In a second terminal, launch

```
ros2 run rviz2 rviz2
```

You should get an empty rviz window like:



Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
Global Status: E…	
Fixed Frame	Frame [map] does not…
Grid	✓

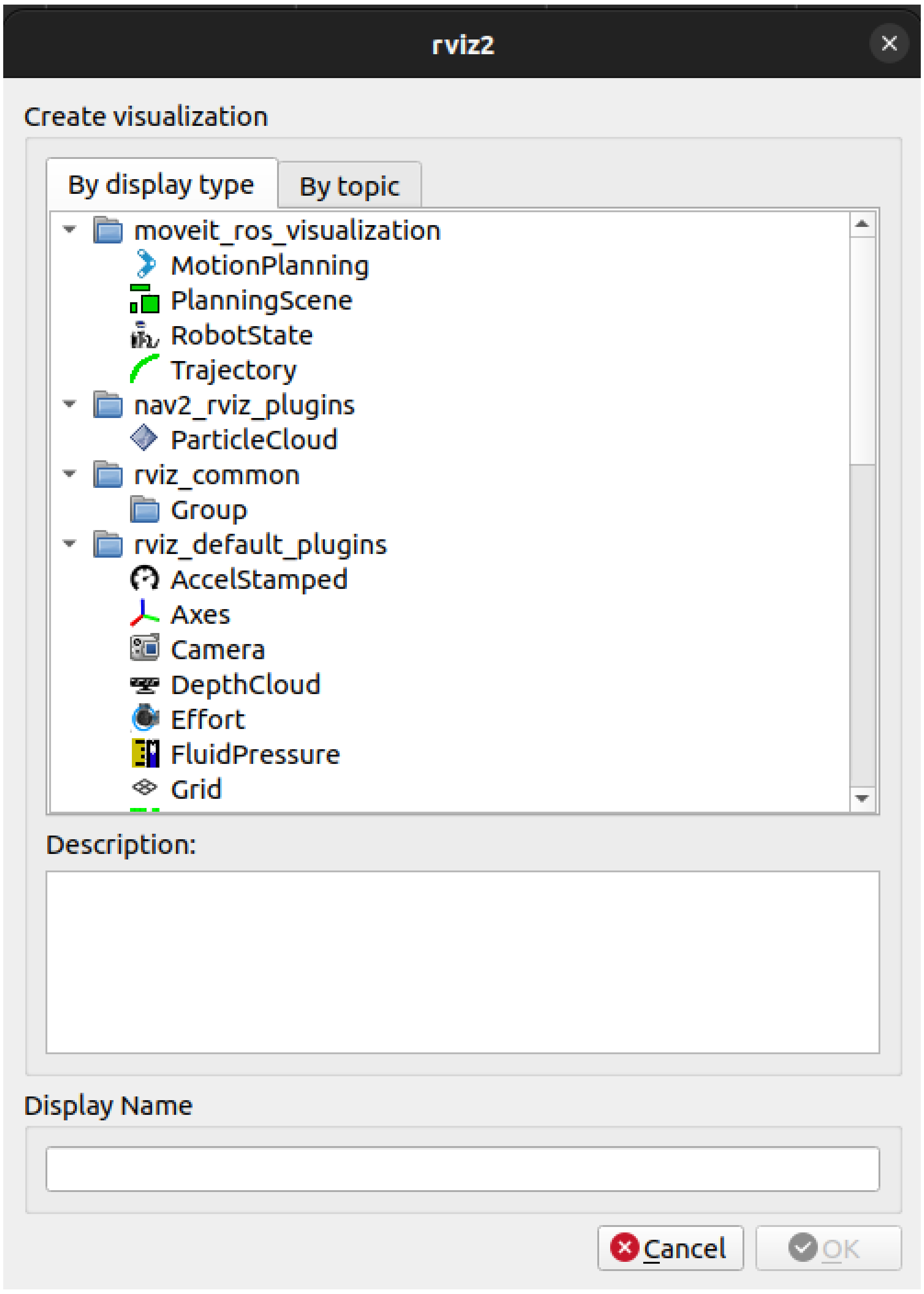
Type: Orbit (rviz_defau

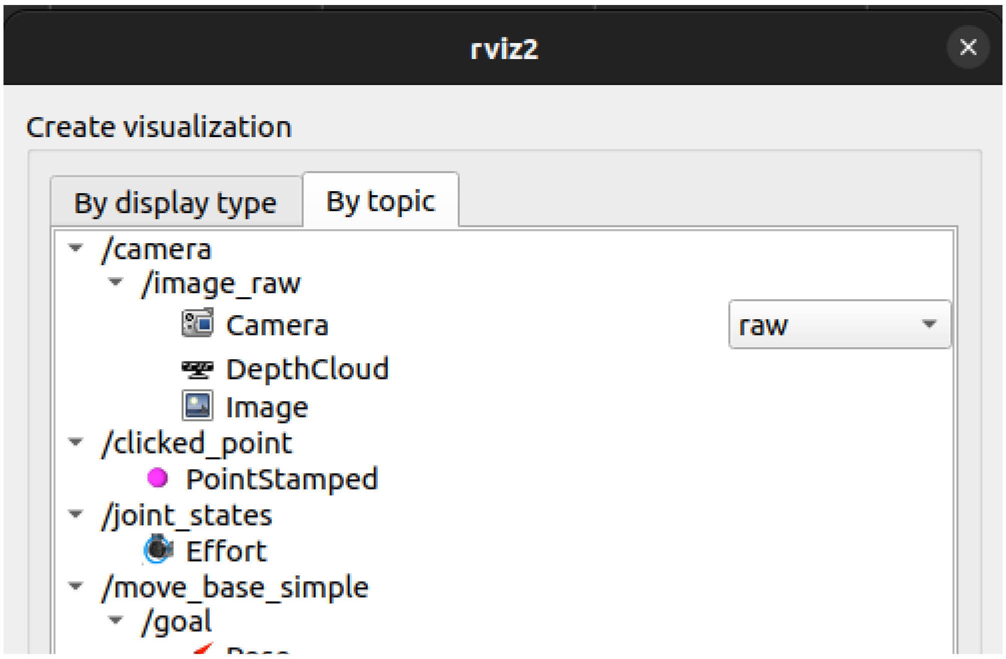
Current View	Orbit (rviz)
Near Clip …	0.01
Invert Z Axis	☐
Target Fra…	<Fixed Frame>
Distance	10
Focal Shap…	0.05
Focal Shap…	✓
Yaw	0.785398
Pitch	0.785398
Focal Point	0; 0; 0

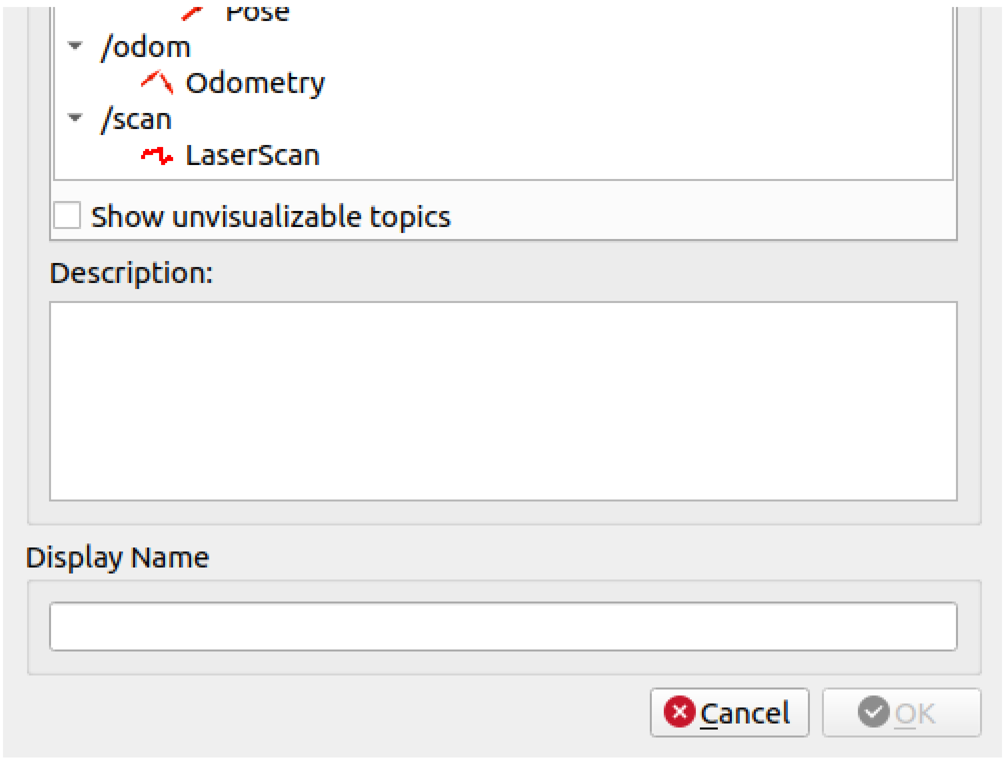
ROS Time: 1756234511.84 ROS Elapsed: 15.04 Wall Time: 1756234511.87 Wall Elapsed: 15.04

RViz is ready. 31 fps

Here you can add information by display type or by topic. You just need to press add and a window should allow you to select by the type or topic needed.


rviz2
Create visualization
By display type
By topic
moveit_ros_visualization
MotionPlanning
PlanningScene
RobotState
Trajectory
nav2_rviz_plugins
ParticleCloud
rviz_common
Group
rviz_default_plugins
AccelStamped
Axes
Camera
DepthCloud
Effort
FluidPressure
Grid
Description:
Display Name
Cancel
OK


rviz2
Create visualization
By display type
By topic
/camera
/image_raw
Camera
raw
DepthCloud
Image
/clicked_point
PointStamped
/joint_states
Effort
/move_base_simple
/goal



Choose which information you would like to visualize. You might need to match the required parameters and names for each case.

You can add as much information as you like. In order for you not to have to add it each time you launch rviz2 you can save the configuration as *[configuration_name].rviz*. To launch rviz with your configuration, just do:

```
ros2 run rviz2 rviz2 -d [path_to_file].rviz
```

To move the robot in this example you can use:

```
ros2 run turtlebot3_teleop teleop_keyboard
```

TASK 2

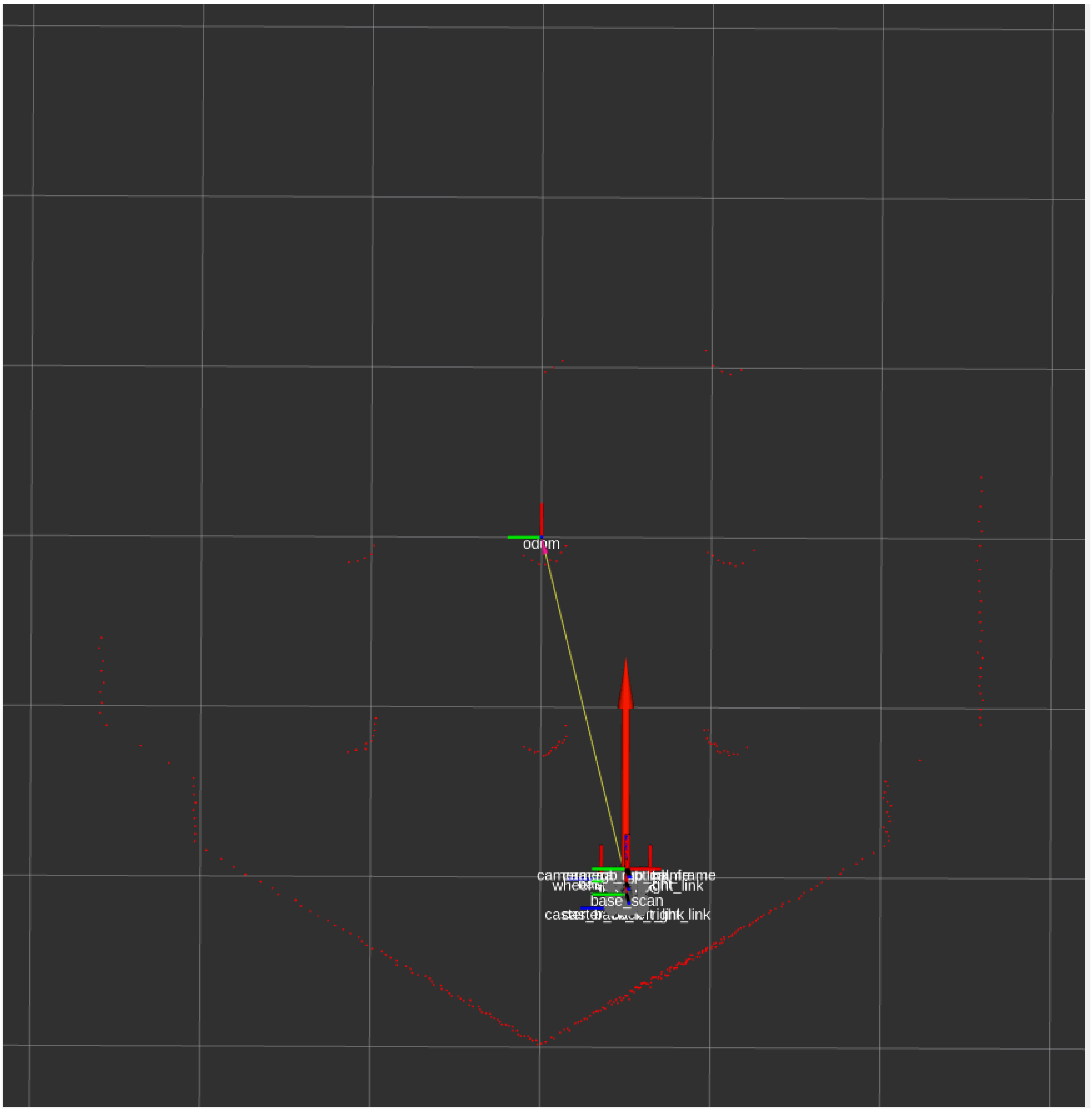
From the previous example use rviz to visualize the following information:

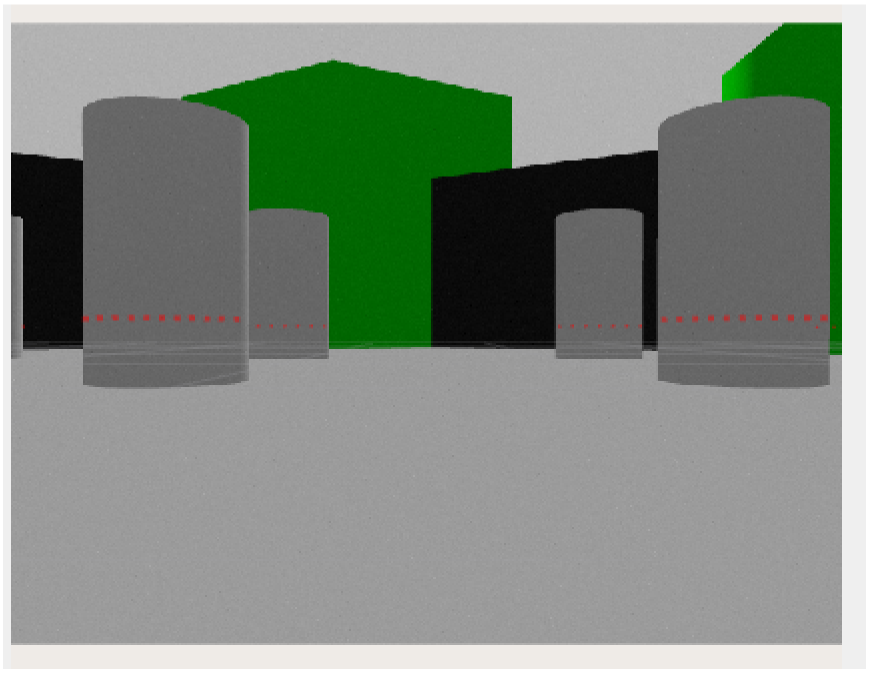
- First you need to select your fixed frame for the global options. We suggest either *base_link* or *odometry*.
- Visualize the following information
  - **Odometry:** the odometry data from the turtlebot. **Tip:** turn the covariance visualization off for a clearer view.
  - **LaserScan:** will show you the lidar data from the Turtlebot3 lidar sensor (the blue lines from Gazebo).
  - **TF:** the TF transformation of all the frames from the turtlebot model.

  - **Camera:** show the image that you get from the camera.
- Toggle the different views, play with some of the parameters and explain what each part shows when you add it and modify it.
- Move the robot around and show different views from the map.
- Did you have to change or set any specific parameters?.
- What happens if you change the *burger* model to *waffle* or *waffle_pi*?

Save your rviz configuration file for presentation.

In the end you should have something similar to this.


odom
base_scan



TF2 Transforms - Coordinates frames

TF2 is a fundamental ROS2 tool that handles coordinate frames and their relationships. In robotics, a robot has many parts—like its base, sensors, and arm—each with its own coordinate frame. TF2 helps a robot understand where all these frames are in relation to each other and to the world around it. It's essential for tasks like combining data from different sensors (sensor fusion) or controlling a robot's movements.

How It Works

To manage these relationships, TF2 uses a few key mathematical concepts:

- **Homogeneous Transformations:** Think of these as special 4x4 matrices that hold information about both rotation and translation, allowing TF2 to describe how one frame is positioned relative to another.
- **Quaternions:** These are a more efficient way to represent 3D rotations than traditional yaw, pitch, and roll, which helps avoid a problem called gimbal lock.
- **Transform Trees:** TF2 organizes all the coordinate frames into a hierarchy, or a tree-like structure, to keep track of their connections.
- **Interpolation:** This allows TF2 to calculate a frame's position at any point in time, even between the moments when new data is received.

When using TF2, it's a good idea to follow these practices:

- Keep your transform trees simple and well-organized.
- Use clear, descriptive names for your frames, like *base_link* or laser_frame.
- Publish transforms at a consistent and appropriate rate (e.g., between 10 and 100 times per second) so the robot always has up-to-date information.
- Always include timestamps so TF2 knows when each transform was published, which is crucial for dynamic, or changing, relationships.

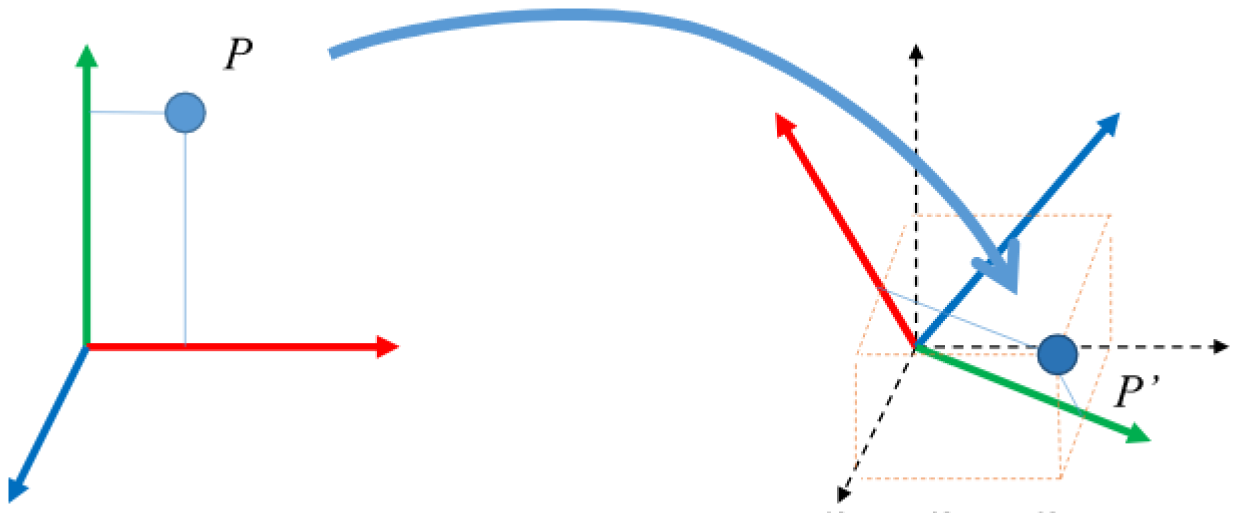
In short, TF2 is the central library for managing coordinate frames and their transforms (the translations and rotations between them) over time, whether they're static (fixed) or dynamic (changing).

Example from: https://husarion.com/tutorials/ros2-tutorials/7-transformation/

Transformation frames

Coordinate transformation is a method used in various fields to describe the same object or event from a different viewpoint. This is helpful for things like simplifying complex equations in physics or creating maps in cartography.

A transformation essentially involves translation (moving from one point to another) and rotation (turning or reorienting). For instance, imagine two different coordinate systems that are rotated relative to each other. To find the position of a point in the second system, you can use a rotation matrix to transform the point's coordinates from the first system to the second.



$$\boldsymbol{p} = (x, y, z) \qquad \boldsymbol{p}' = R\boldsymbol{p} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

https://husarion.com/tutorials/ros2-tutorials/7-transformation/

The translation boils down to a simple addition of the coordinates of the second reference point.

In general, transformations involve simple matrix operations, and coordinate transformation itself is a powerful tool that can help us understand and work with complex systems and phenomena. Fortunately, the tf package has a lot of useful functions that do all these calculations for us.

Basic Transformation

ROS2 helps with tracking coordinate system using the package tf2. For that, we use the message type *geometry_msgs/TransformStamped*. The format of this message is the following:

- **Header header:** the header message that consist of the time stamp for the current transformation and information of the parent ID frame for the message.
- **string child_frame_id:** string containing the child frame ID
- **Transform transform:** the transformation from the coordinate frame in the header (parent) to the child_frame ID coordinate (child).

Static transformation.

A static transform in ROS2's TF2 library is a fixed, unchanging relationship between two coordinate frames. Unlike dynamic transforms, which are constantly updated for things like a robot's moving base, static transforms are only published once and are assumed to remain constant.

For this tutorial let assume we have a robot fixed at *(x,y,z)* relative to the map and slightly rotated. We can now use the static_transform_publisher to get the transformation.

```
ros2 run tf2_ros static_transform_publisher --frame-id map --child-frame-id
robot <optional arguments>
```

The *optional arguments* are the following (for the rotation you can use either quaternion or euler):

- --x - x component of translation
- --y - y component of translation
- --z - z component of translation
- --qx - x component of quaternion rotation
- --qy - y component of quaternion rotation
- --qz - z component of quaternion rotation
- --qw - w component of quaternion rotation
- --roll - roll component Euler rotation
- --pitch - pitch component Euler rotation
- --yaw - yaw component Euler rotation

In a terminal run:

```
ros2 run tf2_ros static_transform_publisher --frame-id map --child-frame-id
robot --x -2 --y 2 --yaw 1.2
```

The output should be:

```
[INFO] [1756321761.242871956]
[static_transform_publisher_voSpcoe61yrif4zT]: Spinning until stopped -
publishing transform
translation: ('-2.000000', '2.000000', '0.000000')
rotation: ('0.000000', '0.000000', '0.564642', '0.825336')
from 'map' to 'robot'
```

We can check the transformation by running the command below, which will show the constantly published transformation

```
ros2 run tf2_ros tf2_echo map robot
```

Output:

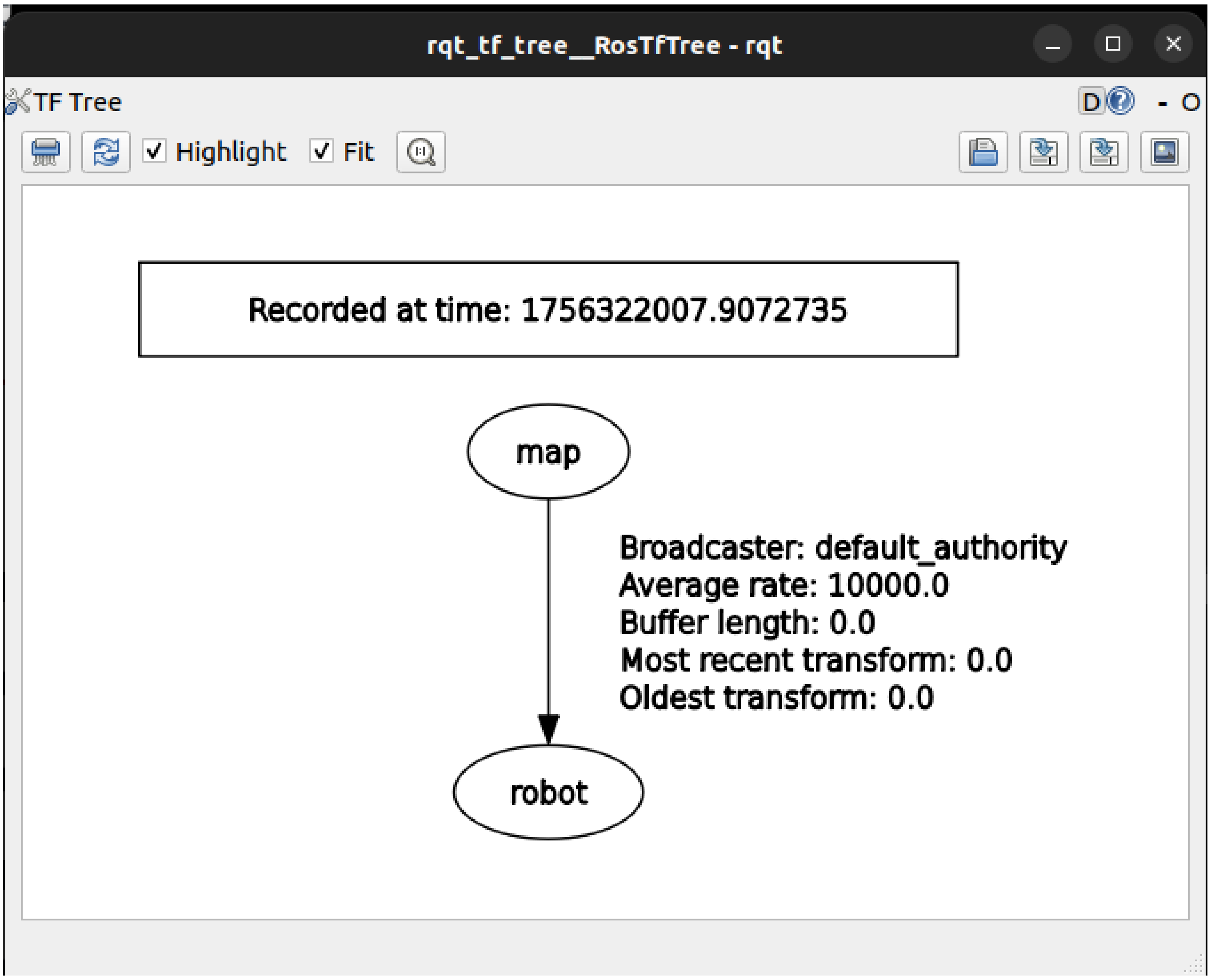
```
[INFO] [1756321905.576049773] [tf2_echo]: Waiting for transform map ->
robot: Invalid frame ID "map" passed to canTransform argument target_frame
- frame does not exist
At time 0.0
- Translation: [-2.000, 2.000, 0.000]
- Rotation: in Quaternion (xyzw) [0.000, 0.000, 0.565, 0.825]
- Rotation: in RPY (radian) [0.000, -0.000, 1.200]
- Rotation: in RPY (degree) [0.000, -0.000, 68.755]
- Matrix:
0.362 -0.932  0.000 -2.000
0.932  0.362  0.000  2.000
```

```
    0.000  0.000  1.000  0.000
    0.000  0.000  0.000  1.000
```

Here you can check if your input information has been read correctly. You have the translation vector and rotation alongside the transformation matrix. Rotation is shown both in quaternion and euler in this step.

A visual inspection of the relationship between the frames can be done using *rqt* tools.

```
ros2 run rqt_tf_tree rqt_tf_tree
```

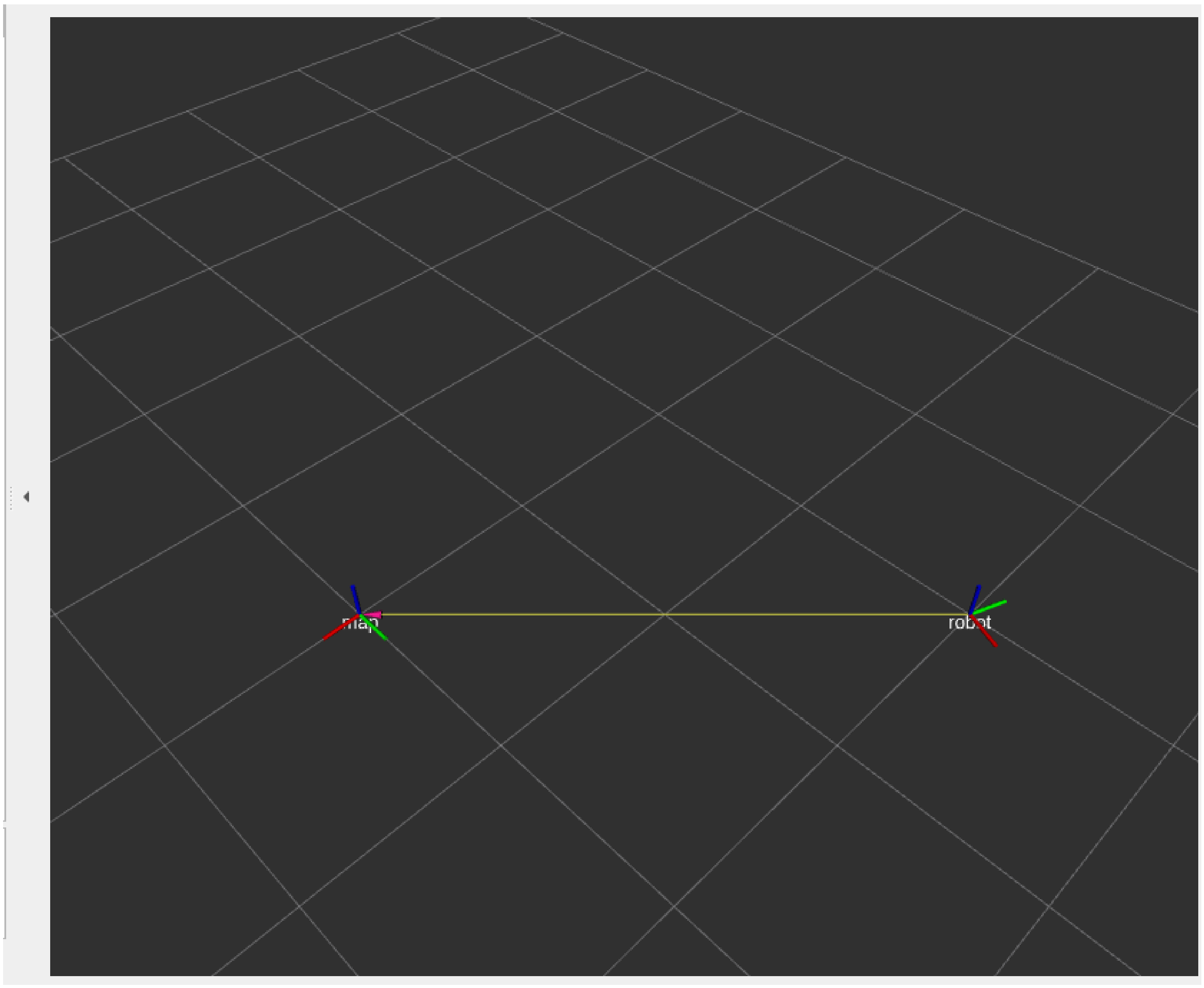


Recorded at time: 1756322007.9072735

Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0

In the image above we can see that *robot* is a child frame from *map*.

The best way to visualize this static transformation is using rviz. Launch rviz2 and add the transformation frames TF. You should get:

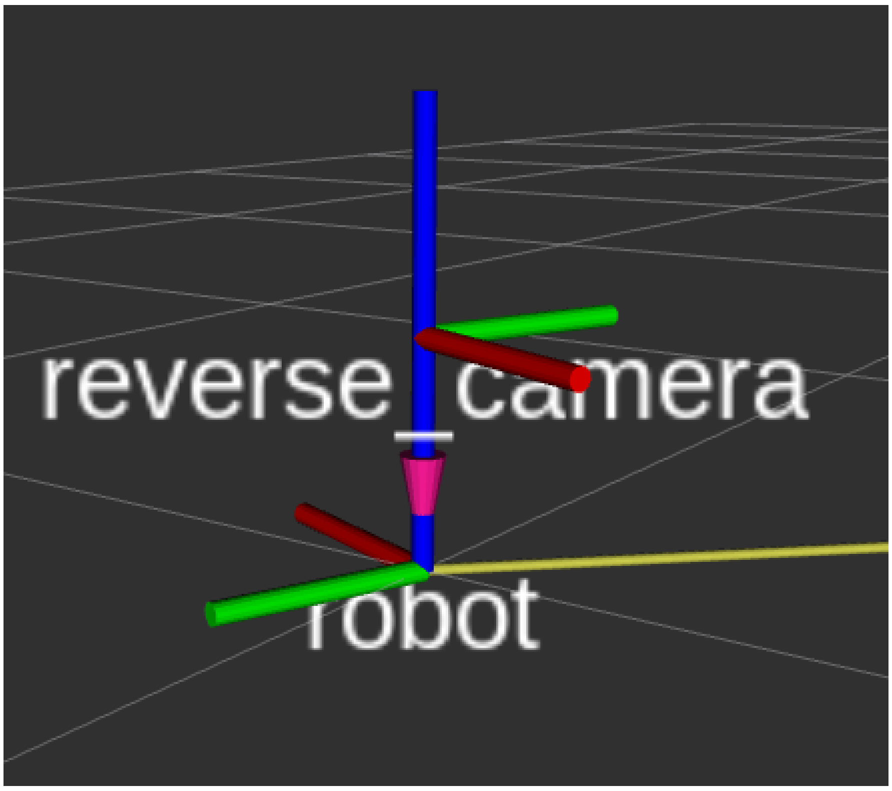


Example

We can add a reverse_camera frame 20 cm above the robot by using the static transformation publisher. Don't forget to rotate the frame. The camera should be facing the opposite way from the robot.

```
ros2 run tf2_ros static_transform_publisher --frame-id robot --child-frame-id reverse_camera --z 0.2 --yaw 3.14
```

Below you can check your implementation.



Dynamic transformations

A dynamic transform in ROS2's TF2 is a relationship between two coordinate frames that changes over time. Instead of being published once, these transforms are continuously broadcast at a specific rate (e.g., 10-100 Hz). This is essential for tracking moving parts of a robot, such as the position of a moving base relative to a fixed map, or the location of a robot arm's end-effector.

Up until now all the frames are static with reference to each other. However, robots move so the transformation can't be static and have to be dynamic, meaning they need to move. ROS2 can also allow this scenario by describing the movement of one frame *(robot)* relative to another *(map)*. For this example we will be creating an object that moves around one point of the map.

We need to use a *TransformBroadcaster*, which sends *TransformStamped* messages, working very similarly to a publisher.

First let's create a package to work with the tf_transforms

```
ros2 pkg create --build-type ament_python tutorial_pkg
```

Add to *setup.py* and *package.xml* the needed dependencies and entry point.

setup.py

```
import os
from glob import glob

data_files=[
    ('share/ament_index/resource_index/packages',
        ['resource/' + package_name]),
    ('share/' + package_name, ['package.xml']),
    (os.path.join('share', package_name, 'launch'),
glob(os.path.join('launch', '*launch.[pxy][y|a]ml]'))),
    (os.path.join('share', package_name, 'img_data'),
glob(os.path.join('img_data', '*.*'))),
],
```

package.xml

```
<depend>rclpy</depend>
<depend>geometry_msgs</depend>
<depend>tf2_ros</depend>
<depend>tf_transformations</depend>
<depend>python3-opencv</depend>
```

TF Broadcaster

Python script: *tf_broadcaster.py*

Create a tf_broadcaster.py and copy the following.

```
import math
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import TransformStamped
from tf2_ros import TransformBroadcaster
from tf_transformations import quaternion_from_euler

class TFBroadcaster(Node):
    """
    Broadcasts a transform from the 'map' frame to the 'robot' frame.
    The 'robot' frame moves in a circular path.
    """
    def __init__(self):
        super().__init__('tf_broadcaster')
        self.tf_broadcaster = TransformBroadcaster(self)
        self.timer = self.create_timer(0.01, self.timer_callback)
        self.get_logger().info('Node started!')

    def timer_callback(self):
        """
        Periodically broadcasts the transform.
        """
        t = self.get_clock().now().seconds_nanoseconds()[0] + \
            self.get_clock().now().seconds_nanoseconds()[1] / 1e9

        transform_stamped = TransformStamped()
        transform_stamped.header.stamp = self.get_clock().now().to_msg()
        transform_stamped.header.frame_id = '' # TODO: set the parent frame id
        transform_stamped.child_frame_id = '' # TODO: set the child frame id

        # Set translation
        transform_stamped.transform.translation.x = math.cos(t)
        transform_stamped.transform.translation.y = math.sin(t)
        transform_stamped.transform.translation.z = 0.0

        # Set rotation using roll, pitch, and yaw
        roll = 0.0
        pitch = 0.0
        yaw = math.fmod(t, 2 * math.pi) + math.pi / 2
        q = quaternion_from_euler(roll, pitch, yaw)

        transform_stamped.transform.rotation.x = q[0]
        transform_stamped.transform.rotation.y = q[1]
        transform_stamped.transform.rotation.z = q[2]
        transform_stamped.transform.rotation.w = q[3]

        self.tf_broadcaster.sendTransform(transform_stamped)

def main(args=None):
    rclpy.init(args=args)
    tf_broadcaster = TFBroadcaster()
    rclpy.spin(tf_broadcaster)
    tf_broadcaster.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Code explanation

```
self.tf_broadcaster = TransformBroadcaster(self)
self.timer = self.create_timer(0.01, self.timer_callback)
```

Declares the specific publisher type and the timer.

```
transform_stamped = TransformStamped()
transform_stamped.header.stamp = self.get_clock().now().to_msg()
transform_stamped.header.frame_id = # TODO: set the parent frame id
transform_stamped.child_frame_id = # TODO: set the child frame id
```

Declares the transformation message, set the timestamp and declares the parent frame and child frame.

```
t = self.get_clock().now().seconds_nanoseconds()[0] + \
self.get_clock().now().seconds_nanoseconds()[1] / 1e9
```

Get the current time in seconds

```
# Set translation
transform_stamped.transform.translation.x = math.cos(t)
transform_stamped.transform.translation.y = math.sin(t)
transform_stamped.transform.translation.z = 0.0
```

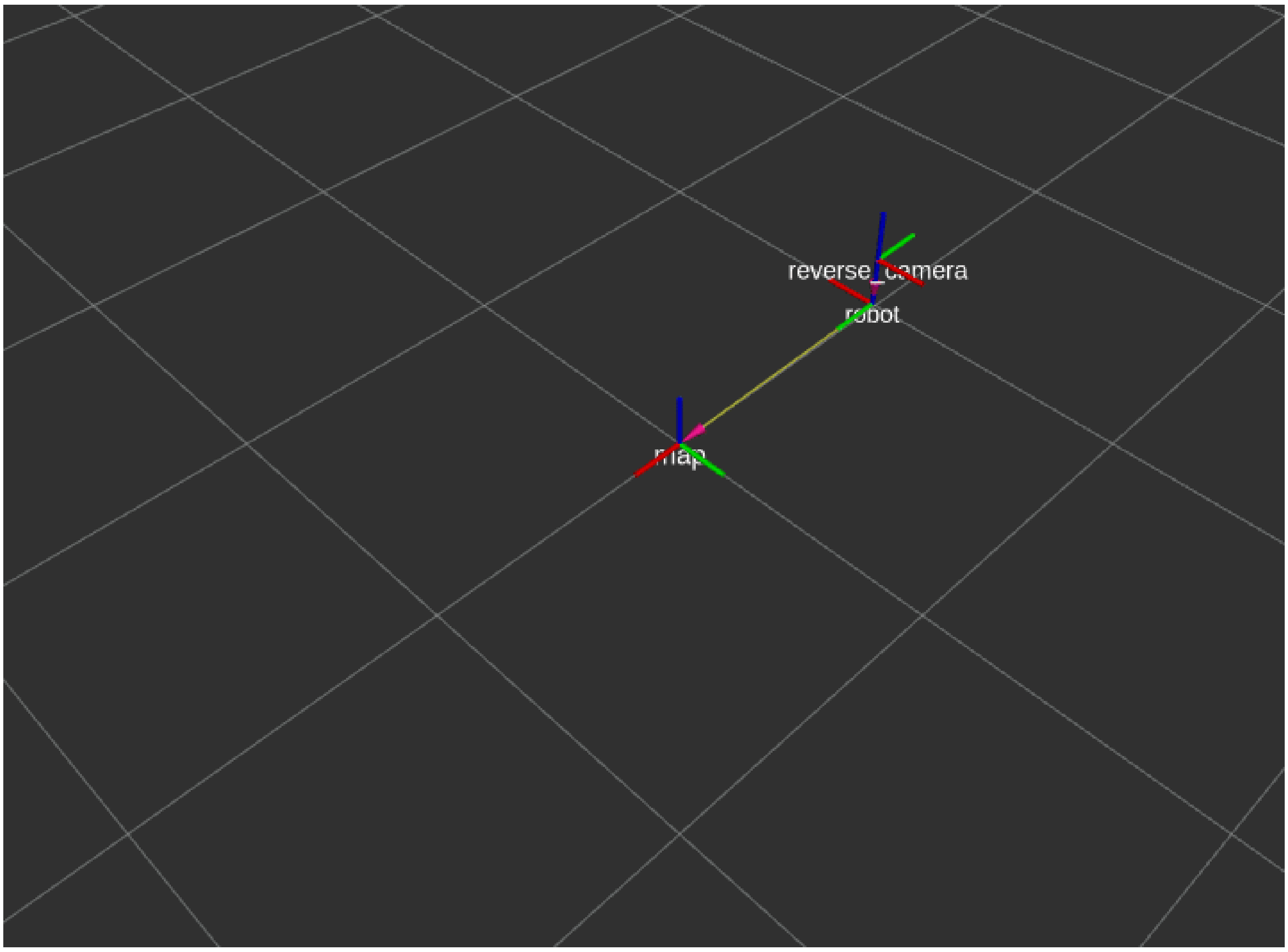
Computes the translation of the frame, in this case for an object to rotate around the center of the map.

```
    # Set rotation using roll, pitch, and yaw
    roll = 0.0
    pitch = 0.0
    yaw = math.fmod(t, 2 * math.pi) + math.pi / 2
    q = quaternion_from_euler(roll, pitch, yaw)

    transform_stamped.transform.rotation.x = q[0]
    transform_stamped.transform.rotation.y = q[1]
    transform_stamped.transform.rotation.z = q[2]
    transform_stamped.transform.rotation.w = q[3]
```

Sets the rotation of the frame.

Run

Finally, build and run the code. You should get the child frame moving around the parent frame.



https://husarion.com/tutorials/ros2-tutorials/7-transformation/

Bonus: You can use the same static transformation to get the reverse camera in the moving robot. Since the frame from the camera is linked to the robot, it should also be moving. You should see now the benefit of using the TF library.

TF Listener

Complementing the TF broadcaster, we also have a TF subscriber. The *TransformListener* is used to receive data, which will automatically determine the transformation between two frames, even if they are not directly connected to each other.

Python script: *tf_listener.py*

Create a tf_listener.py and copy

```
import rclpy
from rclpy.node import Node
from tf2_ros import TransformException
from tf2_ros.buffer import Buffer
from tf2_ros.transform_listener import TransformListener
from tf_transformations import euler_from_quaternion

class TFListener(Node):
    """
    Listens for a transform from 'map' to 'robot' and logs the yaw angle.
    """
    def __init__(self):
        super().__init__('tf_listener')
        self.tf_buffer = Buffer()
        self.tf_listener = TransformListener(self.tf_buffer, self)
        self.timer = self.create_timer(0.01, self.timer_callback)
        self.get_logger().info('Node started!')

    def timer_callback(self):
        """
        Periodically looks up the transform and prints the yaw.
        """
        try:
            transform = self.tf_buffer.lookup_transform(
                '', # TODO: set the parent frame id
                '', # TODO: set the child frame id
                rclpy.time.Time()
            )

            q = (
                transform.transform.rotation.x,
                transform.transform.rotation.y,
                transform.transform.rotation.z,
                transform.transform.rotation.w
            )
            roll, pitch, yaw = euler_from_quaternion(q)
            self.get_logger().info(f'Yaw: {yaw}')

        except TransformException as ex:
            self.get_logger().info(f'Could not transform: {ex}')

def main(args=None):
    rclpy.init(args=args)
    tf_listener = TFListener()
    rclpy.spin(tf_listener)
    tf_listener.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Code explanation

```
self.tf_buffer = Buffer()
self.tf_listener = TransformListener(self.tf_buffer, self)
self.timer = self.create_timer(0.01, self.timer_callback)
```

Declares the buffer, specifies the subscriber type and set the timer callback.

```
transform = self.tf_buffer.lookup_transform(
    '', # TODO: set the parent frame id
    '', # TODO: set the child frame id
    rclpy.time.Time()
)
```

Finds the last transformation from parent frame id to child frame id. The third argument specifies the time which we want to transform. Right now it is set at the latest available transformation.

```
q = (
    transform.transform.rotation.x,
    transform.transform.rotation.y,
    transform.transform.rotation.z,
    transform.transform.rotation.w
)
roll, pitch, yaw = euler_from_quaternion(q)
self.get_logger().info(f'Yaw: {yaw}')
```

Converts quaternion to euler roll, pitch and yaw.

Run

Now build, source and run. Don't forget that the broadcaster also needs to be running just like topic publisher and subscriber.

You should get the yaw read as the robot moves.

```
[INFO] [1756325890.119616909] [tf_listener]: Yaw: 2.14796682697016
[INFO] [1756325890.129657306] [tf_listener]: Yaw: 2.157992328221381
[INFO] [1756325890.139765400] [tf_listener]: Yaw: 2.167974437291205
[INFO] [1756325890.149454167] [tf_listener]: Yaw: 2.1779846797533633
```

TASK 3

- Add a static transformation in python
  - Create a python script that publishes a static transform from the moving *robot* frame to a *lidar* frame.
  - The position should be *X* forward and *Z* up from the robot center.
  - Run and verify that the transform appears correctly using rviz2.
  - **Tip:** Same structure as the dynamic broadcaster.
- Create a python broadcaster with the moving robot as a parent
  - Write a python node that broadcasts a *scanner* frame as a child of the moving robot frame.
  - The scanner should oscillate to left and right about 30cm at a given frequency.

    - Position it above the the robot with an z-offset and a x-offset.
    - You can use the same moving robot from the tutorial
    - Run an verify that the transform appears correctly using rviz2.
- Create simple python listener to verify your transforms.
    - Write a python node that listens to the transform from map to either *lidar* or *scanner*
    - Print the distance from map origin to your frame every 2 seconds
    - Calculate and display the frame's speed relative to the map
- Create a simple python listener to count the number of revolutions of the *robot* frame around the origin in the example scenario
- Show the *ros2 run rqt_tf_tree rqt_tf_tree* output for the final scenario.
- **OPTIONAL:** make the initial moving *robot* frame have a more complex movement and check if all the transformations work correctly

Gazebo simulator

Gazebo is a sophisticated 3D robot simulator that provides realistic physics simulation, sensor modeling, and environmental interactions. It is particularly valuable for testing algorithms before deploying on real hardware, conducting experiments in controlled conditions, and developing in scenarios that might be dangerous or expensive in the real world.

Gazebo engine supports multiple physics engines such as ODE, Bullet, Simbody and DART. It can render high fidelity graphics with shadows, textures and lighting. Sensors such as cameras, LiDARs, IMUs and GPS are modeled with realistic noise. Finally the architecture can have custom behaviors and interfaces.

Some applications of Gazebo are:

- Algorithm development and testing
- Hardware-in-the-loop simulation
- Multi-robot system development
- Training scenarios that are difficult to replicate in real world
- Performance benchmarking under controlled conditions

Gazebo-ROS2 Integration

Gazebo-ROS2 integration enables a robot simulation environment to communicate and be controlled by ROS2.

Some of the features for integration are:

- **Launch Files:** Python-based launch files (.launch.py) which replace the XML integration from ROS. This change gives developers more control and flexibility, since now logic can be used to create more dynamic and conditional launch configurations.
- **Gazebo Plugins:** Bridge between Gazebo physics and ROS2 topics/services, which is the key to the communication between Gazebo and ROS2. This is what allows the simulated sensor in Gazebo to publish data to a ROS2 topic, or what lets a ROS2 control node send a command to a simulated robot in the simulation.
- **URDF/Xacro:** Robot description formats with ROS2 integration, which include the links and joints of the robot's physical structure. This file allows the Gazebo to understand the robot's physical shape,

mass and inertia to apply physics, and ROS2 to understand the robot's joint limits, sensor locations and other properties.
- **tf2:** Transform library for coordinate frames, which is the core ROS2 library that keeps track of the relationship between different coordinate frames over time.

Gazebo basic testing

Set Turtlebot3 Model

You can run these commands in each terminal that you are working with gazebo, **but** it is easier if you add them to your .bashrc. This basically loads the model of the Turtlebot you are working with, and tells the path of where the models are in your system.

```
export TURTLEBOT3_MODEL=waffle_pi
export
GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:/opt/ros/humble/share/turtlebot3_gazeb
o/models
```

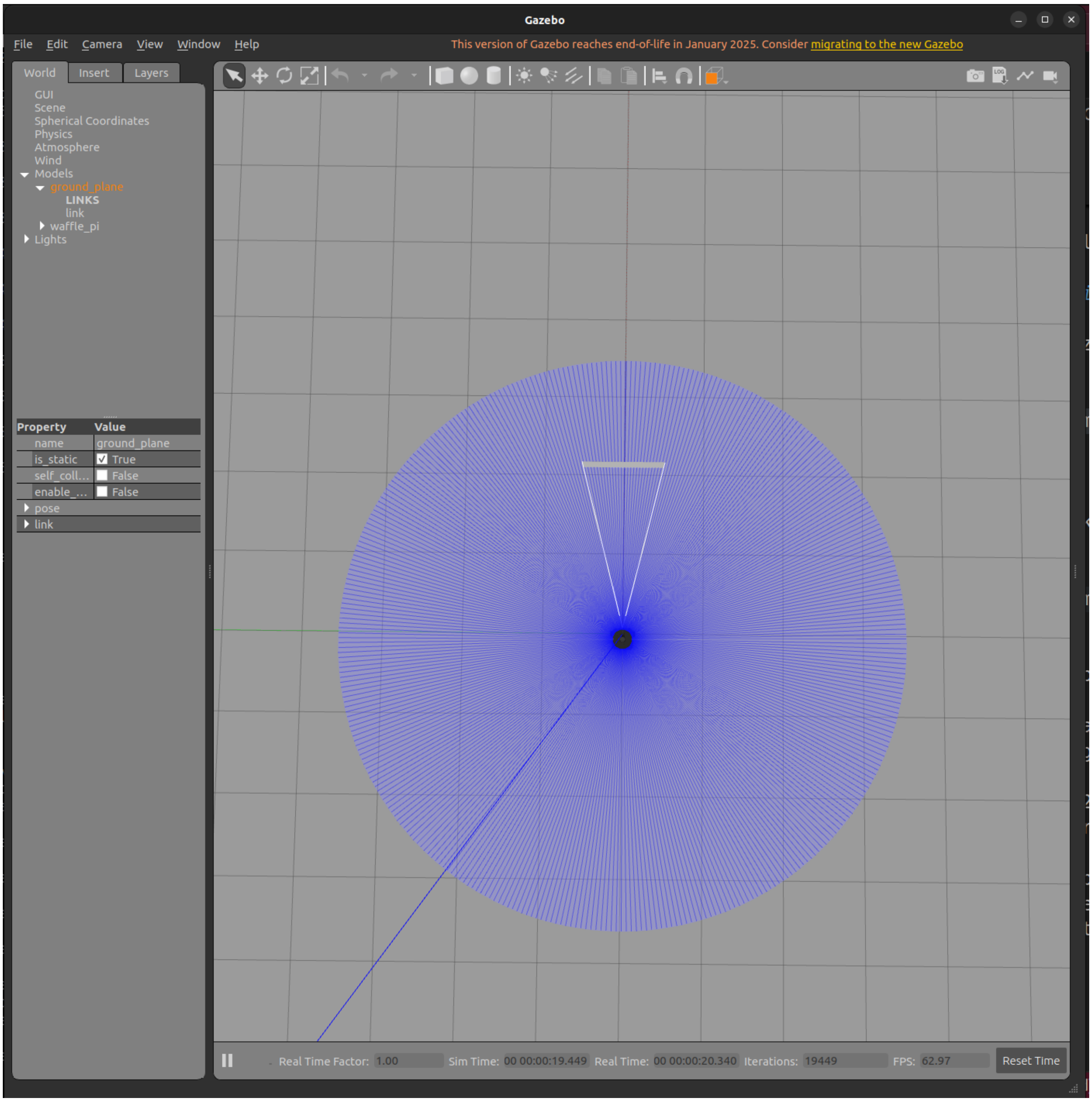
Available models:

- **burger:** Smallest, basic sensors
- **waffle:** Medium size, camera
- **waffle_pi:** Largest, camera + lidar

We will be working with *waffle_pi* for this section

The following command launches a gazebo program with a turtlebot in an empty world

```
ros2 launch turtlebot3_gazebo empty_world.launch.py
```



Testing ROS2 Integration

You can test that everything is working as it should be by:

```
# Terminal 1: Launch simulation
ros2 launch turtlebot3_gazebo empty_world.launch.py
```

This terminal launches the Turtlebot simulator in an empty world

---

```
# Terminal 2: Control robot
ros2 run turtlebot3_teleop teleop_keyboard
```

This terminal allows you to control the movement of the robot by sending Twist command

---

```
# Terminal 3: View camera (if waffle/waffle_pi)
ros2 run rqt_image_view rqt_image_view
```

This terminal launches the *rqt_image_viewer*. Just choose the correct topic and you are going to see what the camera captures. **Tip:** add a model or an object to visualize

---

```
# Terminal 4: Visualize in RViz2
ros2 run rviz2 rviz2
```

This terminal will launch an rviz window, add the relevant data to it.

From here you can manually add an object to the world and see if it appears in the lidar data and the camera. To add you can just use the options that appear at the top toolbar in Gazebo with drag and drop basic models.

Built-in worlds

This package already comes with some built-in worlds you can use for testing. Launch ANY of these and see the difference.

```
# Empty world (OPTION 1)
ros2 launch turtlebot3_gazebo empty_world.launch.py

# TurtleBot3 World (obstacles course) (OPTION 2)
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py

# House environment (resource-intensive) (OPTION 3)
ros2 launch turtlebot3_gazebo turtlebot3_house.launch.py
```

To check which launch files you have available you can do the following:

```
ros2 launch turtlebot3_gazebo turtlebot3_ [AND PRESS TAB]
```

You should get all the options available to you. Keep in mind that some might need to load and can take a long time. You should get something like this:

```
turtlebot3_autorace_2020.launch.py
turtlebot3_dqn_stage1.launch.py
turtlebot3_dqn_stage2.launch.py
turtlebot3_dqn_stage3.launch.py
turtlebot3_dqn_stage4.launch.py
turtlebot3_house.launch.py
turtlebot3_world.launch.py
```

Launch file parameters

You can change some parameters while launching the worlds. Some examples are shown below

Changing the robot initial position:

```
# Specify robot position
ros2 launch turtlebot3_gazebo empty_world.launch.py x_pose:=2.0 y_pose:=1.0
```

That launch changes the model of the Turtlebot used.

```
# Use different TurtleBot3 model
TURTLEBOT3_MODEL=burger ros2 launch turtlebot3_gazebo empty_world.launch.py
```

The *use_sim_time:=true* parameter tells a ROS2 node to get its time from the /clock topic instead of the computer's system clock. **This is particularly useful when working with rosbags**. It is crucial when synchronizing all nodes in a simulation.

-**True:** the node working with the /clock topic. -**False:** the node uses the computer's real-time

```
# Launch paused
ros2 launch turtlebot3_gazebo empty_world.launch.py use_sim_time:=true
```

Creating a custom World

We are going to create a new package to spawn a turtlebot3 and launch our own custom worlds.

```
ros2 pkg create --build-type ament_python gazebo_example --license Apache-2.0 --dependencies launch launch_ros gazebo_ros
```

Inside this package create a *launch* folder and a *worlds* folder.

We also need set up our package. In setup.py in *data_files* add:

```
# Install launch files
(os.path.join('share', package_name, 'launch'), glob(os.path.join('launch',
'*.py'))),
# Install world files
(os.path.join('share', package_name, 'worlds'), glob(os.path.join('worlds',
'*.world'))),
```

In the *package.xml* add the dependencies:

```
<!-- Dependencies -->
<depend>rclpy</depend>
<depend>gazebo_ros_pkgs</depend>
<depend>turtlebot3_gazebo</depend>
<depend>robot_state_publisher</depend>
<depend>joint_state_publisher</depend>
```

To create a custom world there are three methods available. The example for all three is below, but I suggest using the first one for this, since it is the easiest.

Method 1: Using Gazebo Building Editor (RECOMMENDED)

First start a gazebo

```
gazebo
```

Go to edit -> building editor. From here you can add walls, doors and windows. Just create a basic room with some windows and doors.

Then go to File -> save as, choose the folder and model name. Then exit building editor.

Here you can insert models and add some furniture. To do so you have two options.

- For some basic models you can insert from the top toolbar some basic solids.
- Go to Insert -> http://models.gazebosim.... (online model repository)
    - Models from here will be downloaded from the internet so connection is necessary and depending on the selection it might take some time.
    - You can play with the properties to change size, position, and orientation.

Finally, do File -> save world as, and save your creation.

Method 2: Creating world files manually

Create ~/my_turtlebot3_world.world:

```
<?xml version="1.0" ?>
<sdf version="1.6">
<world name="my_turtlebot3_world">

    <!-- Physics -->
    <physics type="ode">
    <max_step_size>0.001</max_step_size>
    <real_time_factor>1</real_time_factor>
    <real_time_update_rate>1000</real_time_update_rate>
    </physics>

    <!-- Lighting -->
    <include>
    <uri>model://sun</uri>
    </include>

    <!-- Ground -->
    <include>
    <uri>model://ground_plane</uri>
    </include>

    <!-- Obstacles for navigation -->
    <model name="obstacle_1">
    <pose>3 2 0.5 0 0 0</pose>
    <static>true</static>
    <link name="link">
        <collision name="collision">
        <geometry>
            <box>
            <size>1 1 1</size>
            </box>
        </geometry>
        </collision>
        <visual name="visual">
        <geometry>
            <box>
            <size>1 1 1</size>
            </box>
        </geometry>
        <material>
            <ambient>0.8 0.2 0.2 1</ambient>
        </material>
        </visual>
    </link>
    </model>

    <!-- Walls -->
    <model name="wall_1">
    <pose>5 0 1 0 0 0</pose>
    <static>true</static>
    <link name="link">
        <collision name="collision">
        <geometry>
            <box>
            <size>0.2 10 2</size>
            </box>
        </geometry>
        </collision>
        <visual name="visual">
        <geometry>
            <box>
            <size>0.2 10 2</size>
            </box>
        </geometry>
        <material>
            <ambient>0.5 0.5 0.5 1</ambient>
        </material>
        </visual>
    </link>
    </model>

</world>
</sdf>
```

Method 3: Programmatically Generate Worlds

Create a Python script to generate world files:

```
#!/usr/bin/env python3

def create_maze_world():
    world_content = '''<?xml version="1.0" ?>
<sdf version="1.6">
<world name="maze_world">
    <physics type="ode">
    <max_step_size>0.001</max_step_size>
    <real_time_factor>1</real_time_factor>
    <real_time_update_rate>1000</real_time_update_rate>
    </physics>

    <include><uri>model://sun</uri></include>
    <include><uri>model://ground_plane</uri></include>
'''

    # Generate maze walls
    walls = [
        (2, 0, 4, 0.2), (4, 2, 0.2, 4), (0, 4, 8, 0.2),
        (6, 1, 0.2, 2), (1, 2, 2, 0.2), (7, 3, 2, 0.2)
    ]

    for i, (x, y, sx, sy) in enumerate(walls):
        world_content += f'''
    <model name="wall_{i}">
    <pose>{x} {y} 1 0 0 0</pose>
    <static>true</static>
    <link name="link">
        <collision name="collision">
        <geometry><box><size>{sx} {sy} 2</size></box></geometry>
        </collision>
        <visual name="visual">
        <geometry><box><size>{sx} {sy} 2</size></box></geometry>
        <material><ambient>0.3 0.3 0.3 1</ambient></material>
        </visual>
    </link>
    </model>'''

    world_content += '\n  </world>\n</sdf>'
    return world_content

# Generate and save world
with open('maze_world.world', 'w') as f:
    f.write(create_maze_world())

print("Maze world generated: maze_world.world")
```

Launch files

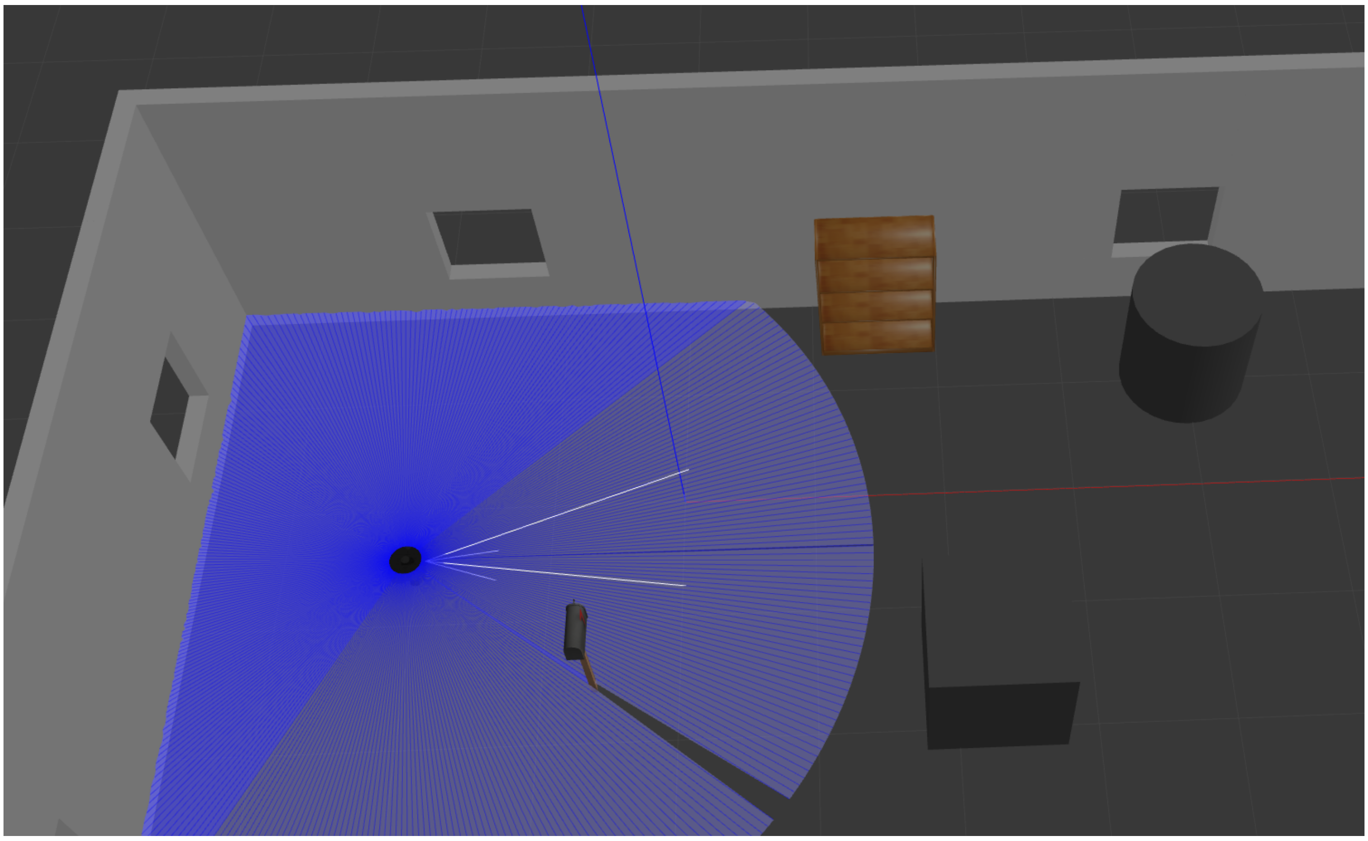
Python script: *gazebo_example.launch.py*

This is the blueprint for the setup file you need to use to launch both the custom world and the turtlebot

```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription, DeclareLaunchArgument
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node

def generate_launch_description():
    pkg_gazebo_ros = get_package_share_directory('gazebo_ros')
    pkg_turtlebot3_gazebo =
get_package_share_directory('turtlebot3_gazebo')
    pkg_my_world = get_package_share_directory('') # TODO: your package
name
    pkg_turtlebot3_description =
get_package_share_directory('turtlebot3_description')

    # Launch configuration variables
    use_sim_time = LaunchConfiguration('use_sim_time', default='true')
    x_pose = LaunchConfiguration('x_pose', default='') # TODO: choose spawn
position
    y_pose = LaunchConfiguration('y_pose', default='') # TODO: choose spawn
position
```

```
    # World file path
    world = os.path.join(pkg_my_world, 'worlds', '.world') # TODO: fill the
name of your custom world

    # Gazebo server
    gzserver_cmd = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(pkg_gazebo_ros, 'launch', 'gzserver.launch.py')
        ),
        launch_arguments={'world': world}.items()
    )

    # Gazebo client
    gzclient_cmd = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(pkg_gazebo_ros, 'launch', 'gzclient.launch.py')
        )
    )

    # Robot State Publisher - this will publish the TF tree from the URDF
    urdf_file_name = 'turtlebot3_waffle.urdf'
    urdf = os.path.join(pkg_turtlebot3_description, 'urdf', urdf_file_name)
    with open(urdf, 'r') as infp:
        robot_desc = infp.read()
        robot_desc = robot_desc.replace('${namespace}', '')

    robot_state_publisher_cmd = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='screen',
        parameters=[{
            'use_sim_time': use_sim_time,
            'robot_description': robot_desc,
            'frame_prefix': ''
        }]
    )

    # Spawn TurtleBot3 - this will spawn the robot model and the associated
plugins
    turtlebot3_model_path = os.path.join(pkg_turtlebot3_gazebo, 'models',
'turtlebot3_waffle_pi', 'model.sdf') # TODO: check the model used

    spawn_turtlebot_cmd = Node(
        package='gazebo_ros',
        executable='spawn_entity.py',
        arguments=[
            '-entity', 'turtlebot3_waffle',
            '-file', turtlebot3_model_path,
            '-x', x_pose,
            '-y', y_pose,
            '-z', '0.1'
        ],
        output='screen'
    )

    ld = LaunchDescription()
    ld.add_action(DeclareLaunchArgument('use_sim_time',
default_value='true', description='Use sim time if true'))
    ld.add_action(DeclareLaunchArgument('x_pose', default_value='',
description='Initial x position of the robot')) # TODO: choose spawn
position
    ld.add_action(DeclareLaunchArgument('y_pose', default_value='',
description='Initial y position of the robot')) # TODO: choose spawn
position

    ld.add_action(gzserver_cmd)
    ld.add_action(gzclient_cmd)
    ld.add_action(robot_state_publisher_cmd)
    ld.add_action(spawn_turtlebot_cmd) # Remove the extra comma here

    return ld
```

Finally launch your file and see what you get. You should get something like this, depending on your world.



TASK 4

- Create your own world.
  - Create a closed room with windows and a door.
  - Include a basic model from the top tool bar (cylinder, box, sphere)
  - Include a model from the predesigned one like a bookshelf, chair, table. You can choose.

  - You are completely free to choose how much things you want to add, just take into account computational power of your pc.
- Adapt your launch file to work with the custom world you just created
- Test your setup with different spawn positions
  - Move around the robot with

    ```
    ros2 run turtlebot3_teleop teleop_keyboard
    ```

  - Launch rviz2 and:
    - select the reference frame
    - add the tf transforms
    - the laser scan data
    - the image data
    - save your rviz configuration

Rosbags

ROS2 bag files are a fundamental tool for data collection, analysis, and system debugging. They allow you to record all or specific topics during system operation and replay them later for analysis, algorithm development, or testing.

ROS2 bags allow you to **record** live data streams and save them in storage. You can then **playback** the information, reproducing the streams exactly as they occurred. The bag can also be **inspected** without having to play it. Finally, you can **filter** either while recording or playing, by only including specific topic or excluding some.

ROS2 bags are normally used for algorithmic development by recording sensor data and the using it for offline testing. It is also very helpful for systems debugging with problematic scenarios. They help in the sharing of datasets between team members or research groups or to help recreate specific scenarios. With rosbags you can create standard datasets that help in benchmarking algorithms or sensors. Finally, it helps to generate datasets for machine learning applications.

Basic recordings

```
# Record all topics
ros2 bag record -a

# Record with specific name
ros2 bag record -a -o [my_bag_name]

# Record specific topics
ros2 bag record /turtle1/cmd_vel /turtle1/pose

# Record specific topics from file
ros2 bag record $(cat topics.txt)
```

```
# Record with custom name and compression
ros2 bag record -a -o my_experiment_data --compression-mode file --
compression-format zstd /scan /odom /camera/image_raw

# Record for specific duration in seconds (on linux)
timeout 60 ros2 bag record -a

# Record excluding certain topics
ros2 bag record -a -x /imu  # Exclude IMU topic for example

# Record with custom storage format
ros2 bag record -a --storage sqlite3 (for now the only out of the box)
```

BagFile Analysis

```
# Get basic info
ros2 bag info my_bag_name

#Using rqt tools, you need to load the folder with your rosbag
rqt_bag my_bag_name/
```

Playback options

```
# Basic playback
ros2 bag play my_bag

# Play at different speeds
ros2 bag play my_bag --rate 0.5   # Half speed
ros2 bag play my_bag --rate 2.0   # Double speed
# You can also change this value when the bag is playing with the up/down
keys

# Play only specific topics
ros2 bag play my_bag --topics /scan /odom

# Loop playback until stopped
ros2 bag play my_bag --loop

# Skip first N seconds
ros2 bag play my_bag --start-offset 10

# Remap topics during playback (use ROS2 remap syntax, no --remap flag)
ros2 bag play my_bag /scan:=/backup_scan
```

Example

Terminal 1

```
ros2 launch  gazebo_example tb3_custom_world.launch.py
```

Terminal 2

```
ros2 run turtlebot3_teleop teleop_keyboard
```

Terminal 3

```
ros2 bag record -a -o test
```

Now move around the robot while you are recording. After you finish the recording you can check your rosbag info. Try to play with the parameters and see what changes in the rosbag info.

TASK 5

Launch your custom world scenario with a the Turtlebot3, laser data and image data.

- Record two rosbags one with the robot moving around for 10 seconds, then do it for 30 seconds (can be an approximation). (group<#>_rosbag_10sec, group<#>_rosbag_60sec)
    - Check the rosbag information. How do the amount of messages compare between the two? Does it make sense? why? What about the size of the file?
- In a fresh gazebo launch record a rosbag with the name group<#>_rosbag including all topics while you move your robot in a set pattern, let's say a rectangle.
    - Play your rosbag with:

      ```
      ros2 bag play [my_rosbag]/
      ```

    - What happens in the gazebo? Do you get any warnings? Try to give an explanation for these warnings.
    - Play your rosbag with:

      ```
      ros2 bag play [my_rosbag]/ --topics /cmd_vel
      ```

    - What happens now? Do you get any warning? Try to give an explanation for this.

    - What happens if you start the rosbag in the middle (more or less)?
    - Play the rosbag at different speeds and now record new rosbags. Compare the rosbags and explain the differences between them.

- Run your group<#>_rosbag **without** gazebo running, and launch Rviz2
    - Can you get all the data visualized? (TF, image, laser)
    - Show your final setup.
    - Ideally you should not have any warning or errors in Rviz2

Use proper names for your rosbags, and all starting with your group<#>. The rosbags need to be included in your final submission