Tinkerers' Lab
IIT Hyderabad

Mechatronics Summer School 2025

Prepared by:

The Senior Mechatronics Team

Jaideep Nirmal AJ, *Head*
Abhijith Raj, *Senior Core*
Satyavada Sri Harini, *Senior Core*
Taha Mohiuddin, *Senior Core*

Tinkerers' Lab, IIT Hyderabad
Between A, B, G, H Hostel Block, IIT Hyderabad
IITH Main Road, Near NH-65, Sangareddy, Telangana 502285

Tinkerers' Lab
IIT Hyderabad

.

Module 3
ROS 201: Robot Description

Contents

1 Getting to the Point **2**

2 The Bigger Picture **3**
2.1 Robot State Publisher . . . 3
2.2 URDF . . . 4
2.3 xacro files . . . 6
2.4 RViz . . . 8
2.4.1 Displays:- . . . 9
2.4.2 Views Panel:- . . . 10
2.4.3 Coordinate Frames:- . . . 11
2.5 Transforms . . . 12

3 (More than) Hello, URDF World! **15**
3.1 Deciding What to Build . . . 15
3.2 Deciding How to Build . . . 15
3.3 The Build . . . 16
3.4 Gazebo Plugins . . . 21

4 Gazebo **23**
4.1 Installing Gazebo . . . 24
4.2 Creating a World . . . 24
4.3 Launch File . . . 30
4.4 World through the Bot's eyes . . . 32
4.5 Moving the Bot around . . . 34

5 DIY Tasks **38**

1 Getting to the Point

Having learnt the basics of ROS and understood the concepts with turtlesim, we are now ready to get to the actual point - building robots. As mentioned in the first module, we first simulate robots before testing them out in the real world, in order to minimise monetary damages. But how do we simulate the robots?

We simulate a robot in way quite similar to building a webpage. Yes, you read it right. We *describe* the robot using XML files. We use URDF - *Unified Robot Description Format* to describe a robot model in ROS.

More technically, URDF is an XML-based file format used to describe the kinematic and visual aspects of a robot. It also supports including 3D models as links within the description. Essentially, you can use URDF to reference a CAD model's geometry and visual appearance, making it a part of the robot's description in the URDF file.

Simulating a robot sounds very easy, but is in fact the most challenging part. Keep in mind that this is more-or-less a *multiphysics* simulation. We will have to account for last thing that could alter the kinematics of the robot. By multiphysics, we essentially mean take $g = 9.81$ and not 10, do NOT ignore friction, do NOT treat everything to be a sphere - things we always did in JEE times.

Thankfully, all of this hardwork is already done by the Gazebo team and we only have to give them the data they need. And we do it through the URDF files. A URDF file usually has 3 aspects

1. **Geometrical Description** - basically, size and shape
2. **Visual Details** - color and texture
3. **Physics Variables** - Mass and Moment(s) of Inertia ($\boldsymbol{I}$ *is a tensor*)

We mention all of these systematically in a neatly written XML URDF file.

2 The Bigger Picture

2.1 Robot State Publisher

In ROS 2, the `robot_state_publisher` is a crucial node used to publish the coordinate transforms (TFs) of a robot to the `/tf` topic. It reads a URDF (Unified Robot Description Format) model of the robot and a stream of joint states (typically published by `joint_state_publisher` or a hardware driver), and from these it computes and broadcasts the 3D transforms between each link in the robot.

This is essential for RViz and other ROS tools to visualize how different parts of the robot move in relation to each other, and to know where everything is located in space. For example, if a robot has a movable arm, `robot_state_publisher` will continuously update the pose of the arm's links as the joint angles change.

How It Works:

- The node takes in the robot's description (usually a URDF or a processed `xacro` file) as a parameter named `robot_description`.
- It subscribes to the `/joint_states` topic for the latest joint values.
- Using the joint values and the kinematic tree defined in the URDF, it calculates the transformation between links.
- It publishes the resulting transforms to the `/tf` topic.

You can test the robot state publisher by:

- Writing a URDF or Xacro model of your robot.
- Running a launch file that starts both `robot_state_publisher` and `joint_state_publisher_gui`.
- Opening RViz and adding the `RobotModel` and `TF` displays.
- Interacting with the joints in the GUI to observe live transformation changes.

Understanding Joints and Links:

To describe a robot's physical structure, URDF models use two core elements: **links** and **joints**.

- **Links** represent the rigid parts of the robot — such as wheels, body frames, arms, etc. Each link can have its own geometry, material, and visual/inertial properties.
- **Joints** connect two links and describe how one link moves relative to another. Joints define the type of motion allowed, such as:

- **fixed**: no motion, e.g., chassis to base_link.
- **revolute**: rotational movement, e.g., robot arms or wheels.
- **prismatic**: linear movement, e.g., a sliding joint.
- **continuous**: like revolute but allows unlimited rotation.
- **floating**, **planar**: advanced joint types (less common).

Each joint connects a `parent` and a `child` link and may have properties like origin (position and orientation offset), axis of rotation/translation, and joint limits.

Example:
Below is a simplified example of a wheel joint in URDF:

```
<joint name="left_wheel_joint" type="continuous">
    <parent link="base_link"/>
    <child link="left_wheel"/>
    <origin xyz="0 0.175 0" rpy="0 0 0"/>
    <axis xyz="0 0 1"/>
</joint>
```

This joint connects `base_link` to `left_wheel` and allows unlimited rotation around the z-axis — a perfect fit for a differential drive robot's wheel.

By defining a chain of links and joints and connecting them appropriately, a complete robot model is formed. The `robot_state_publisher` then uses these definitions along with joint states to compute how each part of the robot moves in real-time.

2.2 URDF

The Unified Robot Description Format (URDF) is an XML-based language used to describe the physical configuration of a robot. It defines a robot as a collection of rigid bodies, called **links**, connected via **joints** that specify the kinematic relationships.

1. Basic Structure

Every URDF file starts with a declaration of the robot's name and wraps all descriptions within the `<robot>` tag:

```
<?xml version="1.0"?>
<robot name="my_robot">
  ...
</robot>
```

2. Links

A link represents a single rigid body. Each link may include:

- `<visual>`: Visual representation for RViz or simulators.
- `<collision>`: Simplified shape used for collision detection.
- `<inertial>`: Inertia properties for physical simulation.

Example:

```
<link name="base_link">
  <visual>
    <origin xyz="0 0 0.1" rpy="0 0 0"/>
    <geometry>
      <box size="0.4 0.2 0.1"/>
    </geometry>
    <material name="grey"/>
  </visual>
</link>
```

3. Joints

Joints connect two links and define their relative motion. Each joint must specify:

- Type: `fixed`, `revolute`, `prismatic`, `continuous`, etc.
- `<parent>` and `<child>` link names.
- `<origin>`: The joint's position and orientation relative to the parent.
- `<axis>`: The axis along which motion occurs (for movable joints).

Example of a continuous joint (like a wheel):

```
<joint name="left_wheel_joint" type="continuous">
  <parent link="base_link"/>
  <child link="left_wheel"/>
  <origin xyz="0 0.175 0" rpy="-1.5708 0 0"/>
  <axis xyz="0 0 1"/>
</joint>
```

4. Materials

Materials can be reused across multiple links by defining them once. Example:

```
<material name="blue">
  <color rgba="0.2 0.2 1 1"/>
</material>
```

5. Summary

URDF provides a clear, modular way to represent a robot's physical structure for visualization, simulation, and kinematics. It acts as the backbone for tools like `robot_state_publisher`, RViz, and Gazebo, making it a fundamental component of any ROS 2 robot project.

2.3 xacro files

Xacro (XML Macros) is a powerful extension for URDF files that greatly simplifies the process of describing robots in ROS2. It provides features such as file includes, property definitions, and macros. Xacro enables developers to create modular, reusable, and maintainable robot models.

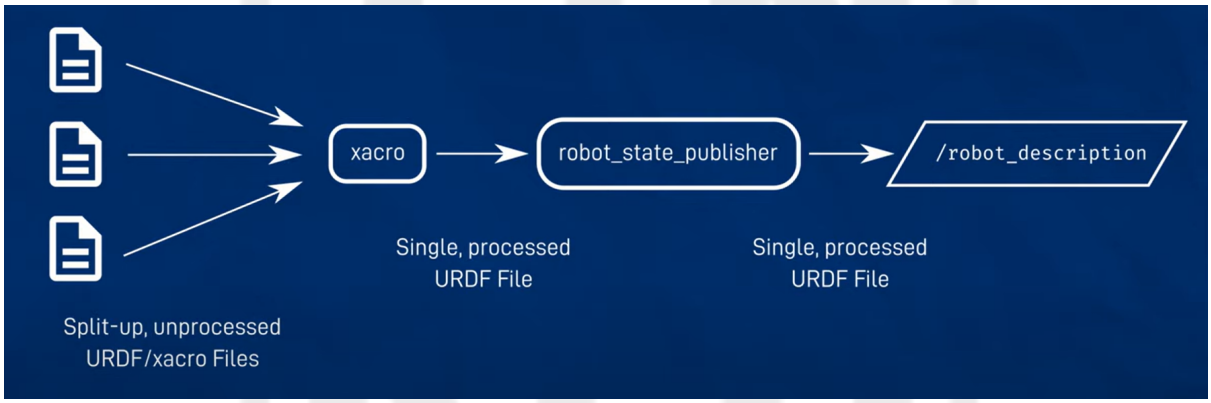

Figure 1: Working of Xacro

In order to use Xacro we need to add the following line to the robot tags:

```
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
```

Key Features

- **File Includes:** Xacro supports modularity by allowing one file to include others. This is useful for organizing a robot into logical components like base, arm, or sensors. For example:

```
<xacro:include filename="camera_sensor.xacro" />
```

- **Property Definitions:** Developers can define reusable properties to represent dimensions, masses, or other parameters. These properties streamline updates and ensure consistency across the robot model. For instance:

```
<xacro:property name="arm_length" value="1.5" />
```

- **Macros for Repetitive Structures:** Xacro macros encapsulate repetitive XML structures, reducing redundancy and improving readability. A typical macro for defining the inertial properties of a box might look like:

```
<xacro:macro name="inertial_box" params="mass x y z">
  <inertial>
    <origin xyz="0 0 0" />
    <mass value="${mass}" />
    <inertia ixx="${mass*(y*y + z*z)/12}" ... />
  </inertial>
</xacro:macro>
```

.

2.4 RViz

In this section we shall look at what is RViz? and some basic configurations, settings and controls involved in using it. In short RViz is a 3D visualizer for the Robot Operating System (ROS) framework. RViz is already intsalled when we do a full installation of ROS2. To start first source the setup file:

```
Terminal
source /opt/ros/humble/setup.bash
```

then start the visualizer:

```
Terminal
ros2 run rviz2 rviz2
```

When RViz starts for the first time, you will see this window:

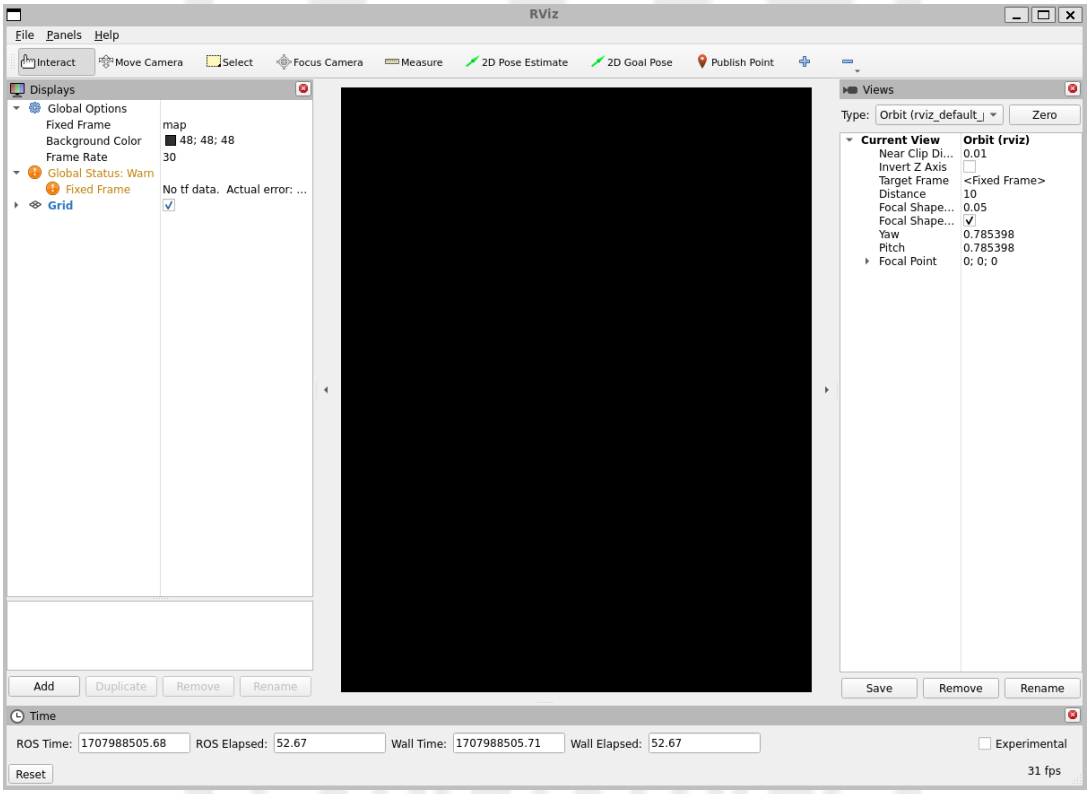

Figure 2: Initial Startup

The big black window in the middle is the 3D view (empty because there is nothing to see). On the left is the Displays list, which will show any displays you have loaded. Right now it just contains the global options and a Grid, which we'll get to later. On the right are some of the other panels.

2.4.1 Displays:-

A display is something that draws something in the 3D world, and likely has some options available in the displays list. An example is a point cloud, the robot state, etc. Now we shall look at some common display types and how to add a display:

- **Adding a Display:** To add a display, click the Add button at the bottom. This will pop up the new display dialog:

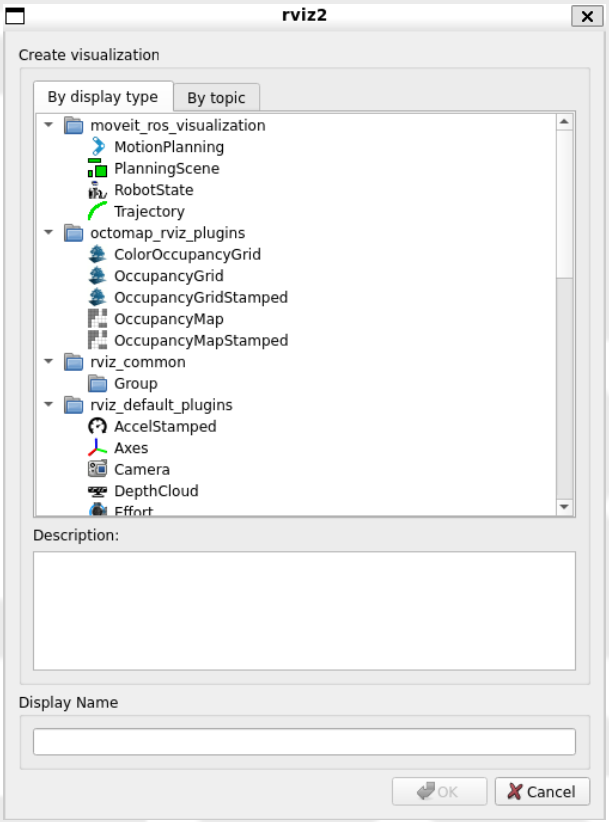

Figure 3: Add Display

The list at the top contains the display type. The type details what kind of data this display will visualize. The text box in the middle gives a description of the selected display type. Finally give the display a unique name. Also each display gets its own list of properties. For example:

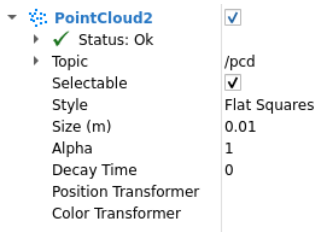

Figure 4: Display Properties

Each display gets its own status which let you know if everything is OK or not. The status can be one of: OK, Warning, Error, or Disabled. The status is indicated in the display's title, as well as in the Status category that you can see if the display is expanded:

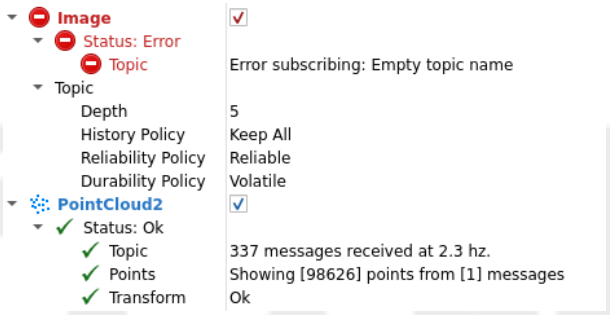

Figure 5: Display Status

You can refer to https://docs.ros.org/en/humble/Tutorials/Intermediate/RViz/RViz-User-Guide/RViz-User-Guide.html for an extended list of display types available.

- **Configurations:-** A configuration consists of displays and their properties, tool properties and viewpoints. Different configurations can be made for better visualization in different circumstances.

2.4.2 Views Panel:-

There are a number of different camera types available in the visualizer. Camera types consist both of different ways of controlling the camera and different types of projection (Orthographic vs. Perspective). The orbital camera view is the one set by default. It simply rotates around a focal point, while always looking at that point. The focal point is visualized as a small disc while you're moving the camera:

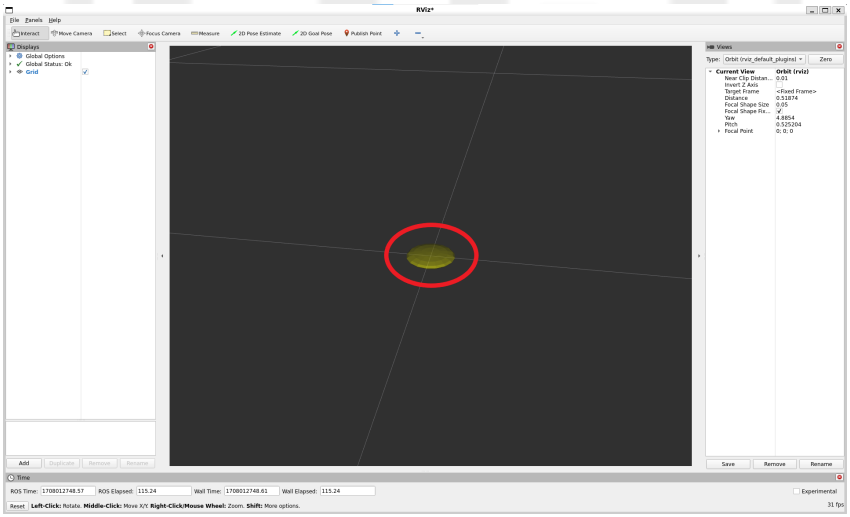

Figure 6: Focal Point

Controls:

- **Left mouse button:** Click and drag to rotate around the focal point.
- **Middle mouse button:** Click and drag to move the focal point in the plane formed by the camera's up and right vectors. The distance moved depends on the focal point – if there is an object on the focal point, and you click on top of it, it will stay under your mouse.
- **Right mouse button:** Click and drag to zoom in/out of the focal point. Dragging up zooms in, down zooms out.
- **Scrollwheel:** Zoom in/out of the focal point

Other than the orbital view there are a few other built in viewtypes like: FPS Camera, Top-Down Orthographic, X-Y Orbit and Third Person follower. You can also design your own custom view through the viewpanel. Refer to the previous link for the steps and controls of other viewtypes.

2.4.3 Coordinate Frames:-

RViz uses the tf transform system for transforming data from the coordinate frame it arrives in into a global reference frame. There are two coordinate frames that are important to know about in the visualizer, the target frame and the fixed frame.

The Fixed Frame: The more-important of the two frames is the fixed frame. The fixed frame is the reference frame used to denote the world frame. This is usually the map, or world, or something similar, but can also be, for example, your odometry frame.

If the fixed frame is erroneously set to, say, the base of the robot, then all the objects the robot has ever seen will appear in front of the robot, at the position relative to the robot at which they were detected. For correct results, the fixed frame should not be moving relative to the world. If you change the fixed frame, all data currently being shown is cleared rather than re-transformed.

The Target Frame: The target frame is the reference frame for the camera view. For example, if your target frame is the map, you'll see the robot driving around the map. If your target frame is the base of the robot, the robot will stay in the same place while everything else moves relative to it.

The transforms are further explained in the next section. Also RViz has tools like Publish point (to publish coordinates of a point), Time (to show simulation time), Measure (to measure distance between two displayed points) get used to them and keep exploring!

2.5 Transforms

Transforms (coordinate transformations) help convert positions and orientations between different frames of reference. Suppose robot 1 detects an object — to tell robot 2 where it is, we must transform coordinates between the respective frames. It is almost essential and very useful in 3D space where manual trigonometry becomes complex.

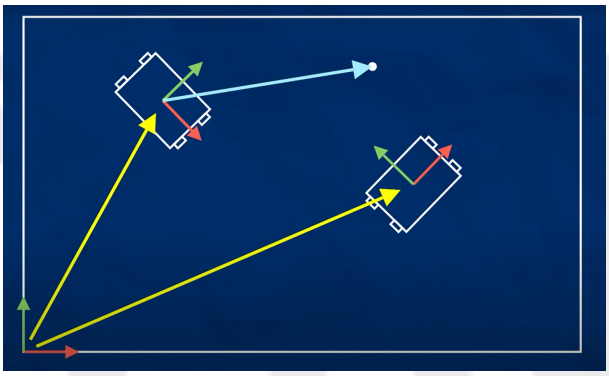

Figure 7: Example

Now as discussed in RViz frames (like world, robot1, robot2) are coordinate reference points. Transforms define how to convert between these frames. All transforms should form a tree (no loops, one parent per frame). Once a transform tree is defined, positions, vectors, and shapes can be converted between frames. Let's look at an example to better understand it a camera above a robotic gripper sees an object — transforms allow determining the object's position relative to the gripper. The transform tree would look like this:

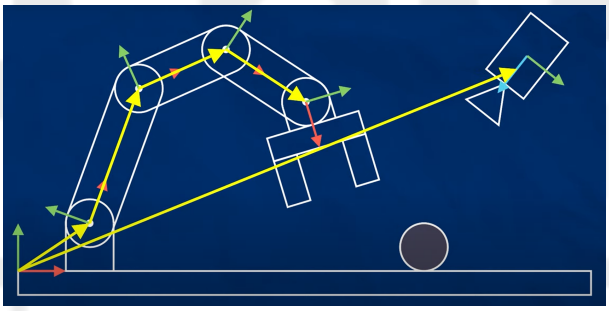

Figure 8: Example

As mentioned TF2 is the library in ROS used for managing transforms. The nodes in ROS which take care of this are called Broadcasters and listeners synonyms for our publishers and subscribers. There are 2 basic types of transforms:

- **Static transforms:** don't change over time.
- **Dynamic transforms:** update continuously; listeners check for freshness.

The static_transform_publisher is used to publish static transforms i.e the fixed relationships between frames like the base of a robot and camera.

Coming to RViz to visualize the transforms add TF to the display. Another reminder to always set the world as the fixed frame. Transforms show up as arrows (from child to parent in RViz). You can modify transform values live and see the effect in RViz.

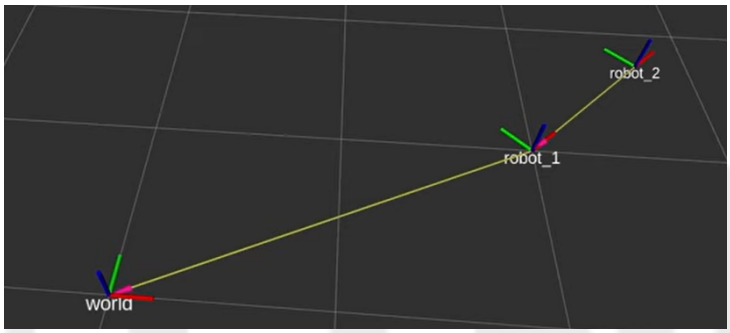

Figure 9: Static in RViz

Now to broadcast and run a dynamic transform we require:

- A URDF file of the robot. (Covered in next section).
- A Robot State Publishing node: It parses the URDF and has the static transforms in it while computing dynamic ones in real time.
- A joint state publisher: because latter needs joint positions to compute dynamic state which are recieved from sensors.

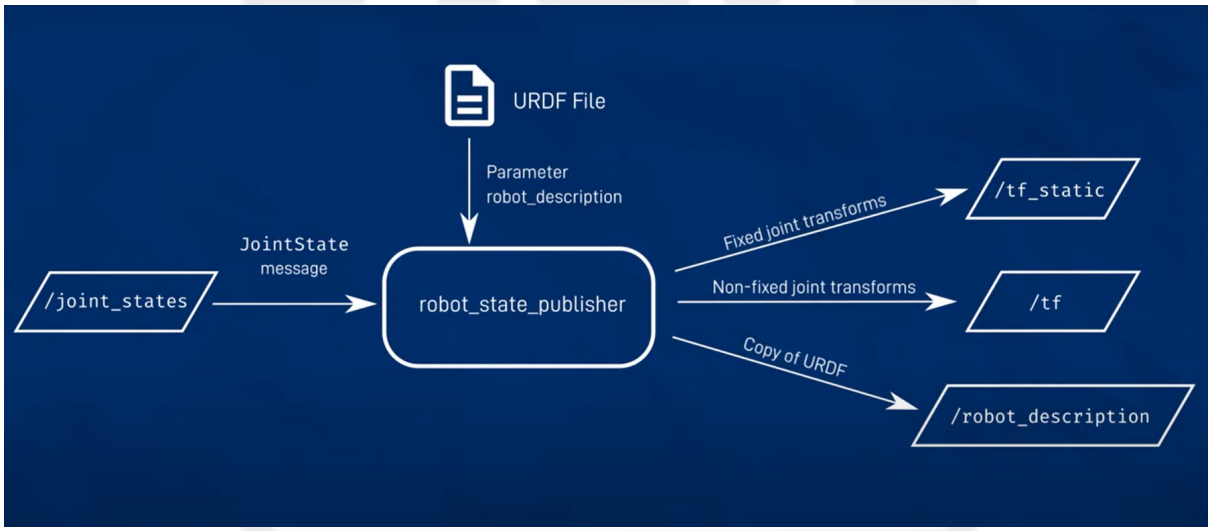

Figure 10: Dynamic transform 1

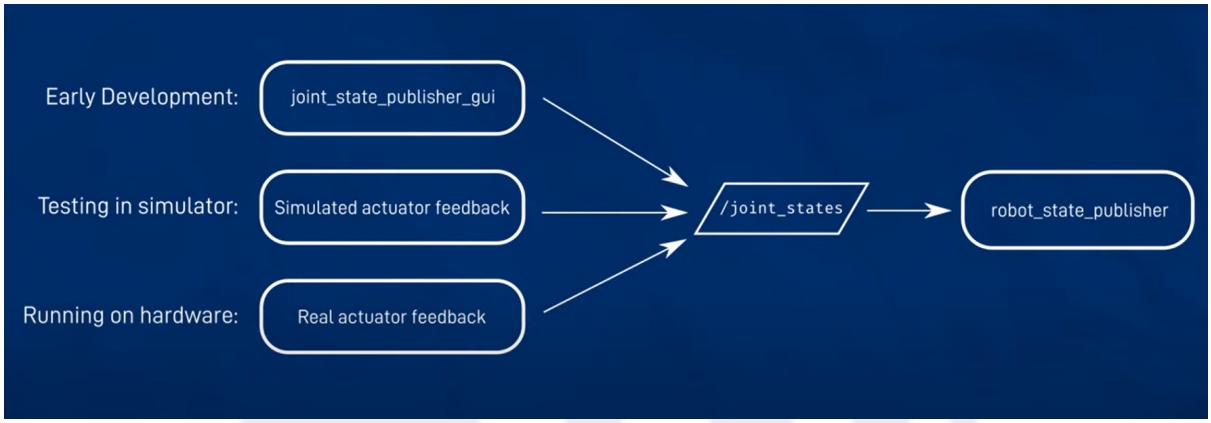

Figure 11: Dynamic Transform 2

PRO TIP

ROS 2 doesn't support rqt_tf_tree, so use:
ros2 run tf2_tools view_frames
This generates a frames.pdf showing the TF tree structure and transform timestamps.

Some differences between URDF and TF's for a better understanding are:

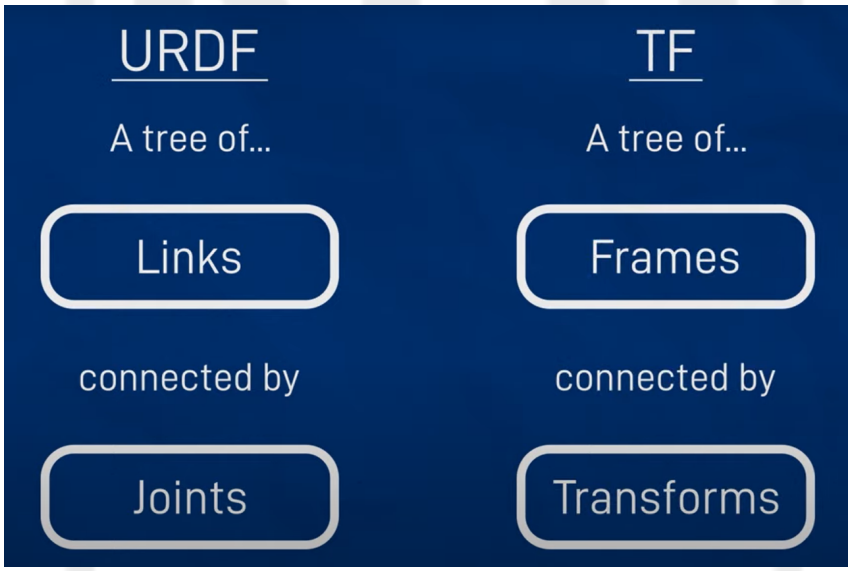

Figure 12: URDF vs TF

The theory of forward and inverse kinematics will be covered in upcoming modules!

3 (More than) Hello, URDF World!

3.1 Deciding What to Build

As a common starting point, we may build a *Differential drive* bot here. By Differential drive, we mean that the bot is mobile with two or more independently controlled motors.

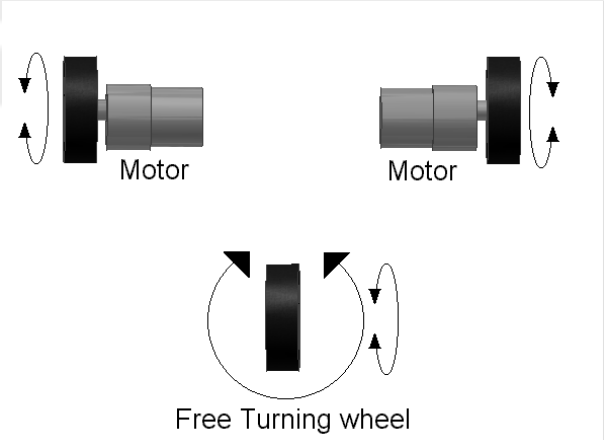

Figure 13: A Diff-Drive bot Schematic

3.2 Deciding How to Build

The next now is to break up the larger model into joints and links that we can describe in a URDF file. The idea here is to keep it as minimal as possible. For a Diff-drive bot, we can probably break it up into 4 parts- 2 diff-drive wheels, 1 caster wheel and a cuboidal body. For the , we don't have to spend a lot of time thinking how to model and can

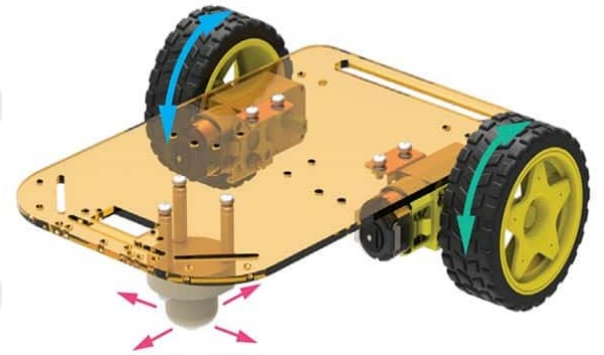

Figure 14: A Diff-Drive Bot

start right away.

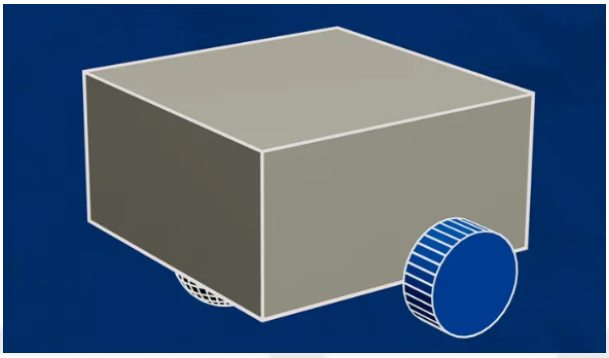

Figure 15: A representation of the bot

3.3 The Build

Where to Build?

Create a package named `bot`. Now within the `bot` subfolder, create another subfolder called `description`. We usually keep all the URDF files organised in a `description` subfolder. Notice that we have altered the usual folder structure by adding another subfolder. And yes, you guessed it right- we need to add this folder also to the install paths- look up the previous modules and figure out. Make sure the files being included are xacro files.

File Setup

In the `desciption` folder in the project, create a file called `robot.urdf.xacro`, and add the following initial content:

```
<?xml version="1.0"?>
<robot xmlns="https://www.ros.org/wiki/xacro" name="robot">

</robot>
```

To make the code more modular, we split the robot description into different URDF files and include everything in a single master file. This way it is also easier to keep track of things. Now, create another file called `robot_core.xacro` and add the same initial content to that file. Then add these lines to the `robot.urdf.xacro`

```
<xacro:include filename="robot_core.xacro"/>
```

Before we go ahead and build the bot, let us describe the colors we will be using in the robot. In `robot_core.xacro`, add these lines

```
<material name="white">
    <color rgba="1 1 1 1"/>
</material>

<material name="blue">
    <color rgba="0.2 0.2 1 1"/>
</material>

<material name="orange">
    <color rgba="1 0.3 0.1 1"/>
</material>

<material name="black">
    <color rgba="0 0 0 1"/>
</material>
```

The base_link

Every URDF file needs to have a base_link. This will be a dummy and will serve as a reference to every other link. We usually leave this blank. Now, add these lines to the robot_core.xacro file:

```
<link name="base_link"></link>

<joint name="chassis_joint" type="fixed">
    <parent link="base_link"/>
    <child link="chassis"/>
    <origin xyz="-0.1, 0, 0"/>
</joint>

<link name="chassis">
    <visual>
        <origin xyz="0.15 0 0.075"/>
        <geometry>
            <box size="0.3 0.3 0.15"/>
        </geometry>
        <material name="white"/>
    </visual>
</link>
```

Now lets see what the robot looks like so far by building it. To make things easier, we will write a launch file for the bot

Launch file:

```
import os

from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch.substitutions import LaunchConfiguration
from launch.actions import DeclareLaunchArgument
from launch_ros.actions import Node

import xacro


def generate_launch_description():

    # Process the URDF file
    pkg_path = os.path.join(get_package_share_directory('bot'))
    xacro_file = os.path.join(pkg_path,'description','robot.urdf.xacro')
    robot_description_config = xacro.process_file(xacro_file)

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

    # Launch!
    return LaunchDescription([
        DeclareLaunchArgument(
            'use_sim_time',
            default_value='false',
            description='Use sim time if true'),

        node_robot_state_publisher
    ])
```

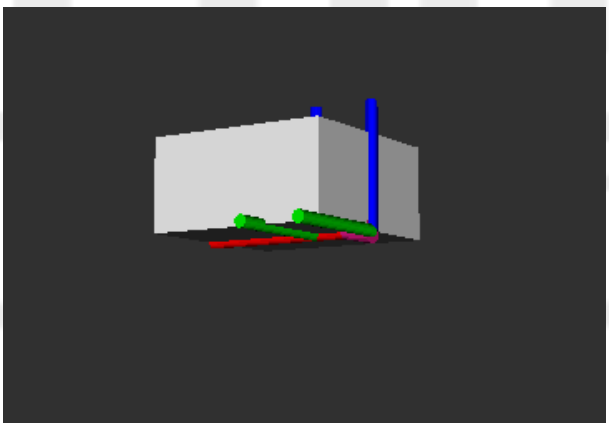

Figure 16: Rviz2 preview

Wheels

The Wheels will be a revolute joint with continuous rotation. Add these lines to the `robot_core.xacro` for the left wheel. Also, copy paste the same and make necessary changes for the right wheel.

```
<link name="left_wheel">
    <visual>
        <origin xyz="0.15 0 0.075"/>
        <geometry>
            <cylinder radius="0.05" length="0.04"/>
        </geometry>
        <material name="blue"/>
    </visual>
</link>

<joint name="left_wheel_joint">
    <parent link="base_link"/>
    <child link="left_wheel"/>
    <origin xyz="0 0.175 0" rpy="-${pi/2} 0 0"/>
    <axis xyz="0 0 1"/>
</joint>
```

Next we need to add the caster wheel to the robot. The following lines will do the same-

```
<link name="caster_wheel">
    <visual>
        <origin xyz="0.15 0 0.075"/>
        <geometry>
            <sphere radius="0.05"/>
        </geometry>
        <material name="black"/>
    </visual>
</link>

<joint name="caster_wheel_joint">
    <parent link="chassis"/>
    <child link="caster_wheel"/>
    <origin xyz="0.24 0 0"/>
</joint>
```

Let's see how the robot looks so far. Use joint state publisher to change the angle of the wheels and see how it looks.

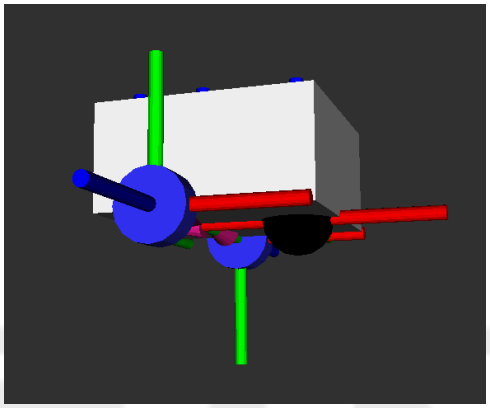

Figure 17: Rviz2 preview

Adding a camera

Create a new xacro file and name it camera.xacro. Do not forget to mention the xacro in the robot.urdf.xacro file.

```
<xacro:include filename="camera.xacro"/>
```

camera.xacro:

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" >

    <joint name="camera_joint" type="fixed">
        <parent link="chassis"/>
        <child link="camera_link"/>
        <origin xyz="0.305 0 0.08" rpy="0 0 0"/>
    </joint>

    <link name="camera_link">
        <visual>
            <geometry>
                <box size="0.010 0.03 0.03"/>
            </geometry>
            <material name="black"/>
        </visual>
    </link>

    <joint name="camera_optical_joint" type="fixed">
        <parent link="camera_link"/>
        <child link="camera_link_optical"/>
        <origin xyz="0 0 0" rpy="${-pi/2} 0 ${-pi/2}"/>
    </joint>

    <link name="camera_link_optical"></link>


</robot>
```

3.4 Gazebo Plugins

In order for gazebo to understand what part of the URDF does what, we add what is called plugins in the URDF. This is done within `<gazebo>` tags. What information we specify within the tag may be pretty self explanatory. Go ahead and append these at the end of your xacro file.

Camera Plugin

[1]

```
<gazebo reference="camera_link">
  <sensor type="camera" name="camera1">
    <update_rate>30.0</update_rate>
    <camera>
      <horizontal_fov>1.3962634</horizontal_fov>
      <image>
        <width>800</width>
        <height>800</height>
        <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.02</near>
        <far>300</far>
      </clip>
    </camera>
    <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
      <alwaysOn>true</alwaysOn>
      <frameName>camera_rgb_optical_frame</frameName>
      <imageTopicName>image_raw</imageTopicName>
      <cameraInfoTopicName>camera_info</cameraInfoTopicName>
    </plugin>
    <visualize>true</visualize>
  </sensor>
</gazebo>
```

[1] Additional Debugging maybe required

Differential Drive Plugin

[2] This enables the robot movement to be controlled by teleop.

```
<gazebo>
    <plugin name="diff_drive_controller" filename="libgazebo_ros_diff_drive
        .so">
        <!-- Update rate in Hz -->
        <update_rate>50</update_rate>
        <!-- wheels -->
        <left_joint>robot_0_base_left_wheel_joint</left_joint>
        <right_joint>"robot_0_base_right_wheel_joint</right_joint>
        <!-- kinematics -->
        <wheel_separation>0.45</wheel_separation>
        <wheel_diameter>0.2</wheel_diameter>
        <!-- output -->
        <publish_odom>true</publish_odom>
        <publish_odom_tf>true</publish_odom_tf>
        <publish_wheel_tf>true</publish_wheel_tf>
        <odometry_topic>odom</odometry_topic>
        <odometry_frame>odom</odometry_frame>
        <robot_base_frame>robot_0_base_footprint</robot_base_frame>
    </plugin>
</gazebo>
```

[2] Additional Debugging maybe required

4 Gazebo

Gazebo is a powerful, open-source 3D robotics simulator that provides a realistic virtual environment for developing, testing, and validating robotics projects. It simulates the dynamics of robots, their sensors, and their interactions with complex indoor and outdoor environments using advanced physics engines. Gazebo supports a wide range of sensors (like cameras, LiDAR, GPS), actuators, and robot models, and allows for the creation of custom robots and environments

Simulating robots is a complex task in itself and is very compute-intensive. You *might* face an issue with simulation or face high jitter or frame drops if you are on a virtual machine. You might also face hiccups simulating larger virual worlds if you do not have a discrete GPU.

There are two variants of Gazebo- the classic gazebo and the newer ignition gazebo. While ignition gazebo had some issues working with ROS humble on Ubuntu 22.04, the classic gazebo has reached its end of life. However as the saying goes- if it works, let it work. You should still be fine with gazebo classic, which is what the module will follow. Gazebo Classic, as of writing this document remains the mature version and supports

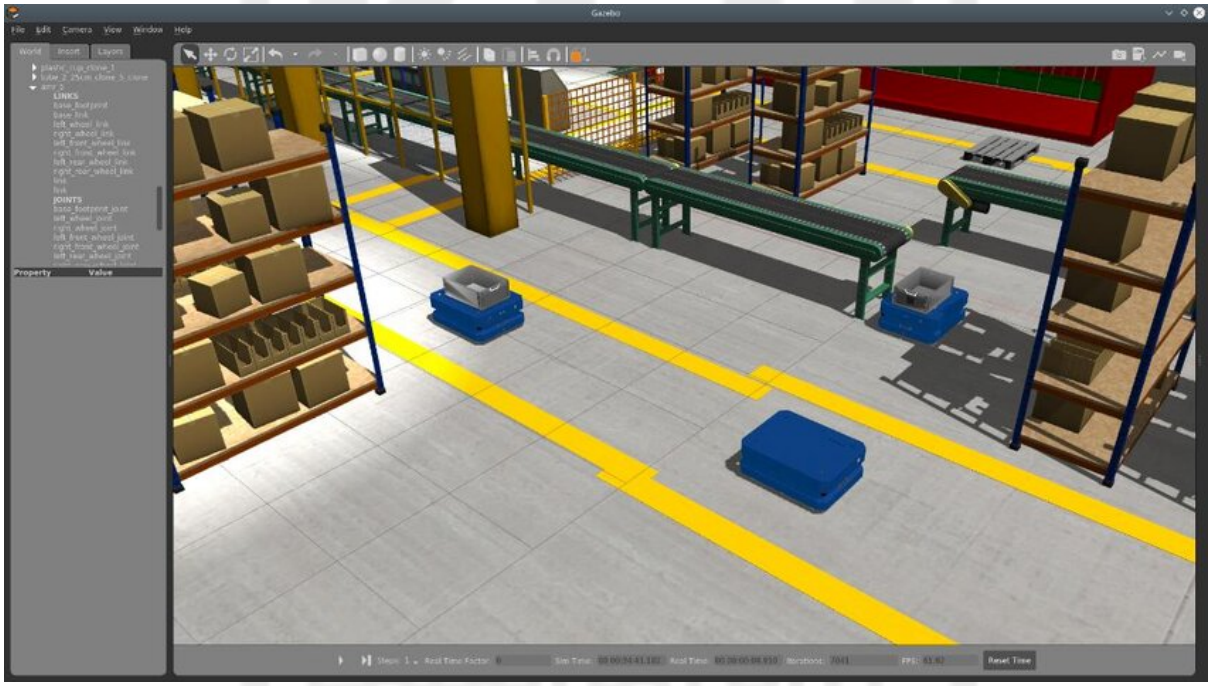

Figure 18: Classic Gazebo Interface

a wide range of sensors and plugins. The classic version was built when ROS-1 was mainstream. Now that ROS1 has seen its sunset, Gazebo has also come up with their update. Ignition Gazebo is the latest version with a modular looking UI. It has several levels of simulation, which supports better rendering with slower or limited hardware.

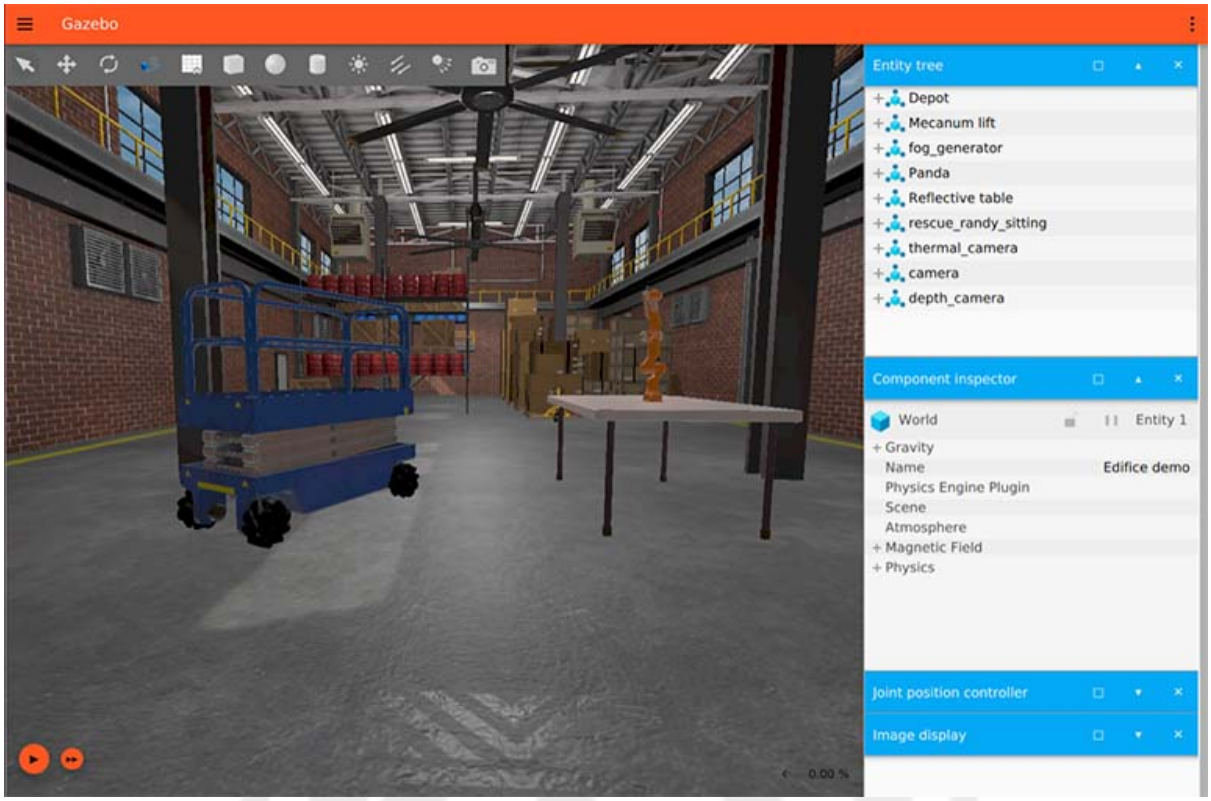

Figure 19: New Gazebo Interface

4.1 Installing Gazebo

Gazebo requires intel i5 or higher, with atleast 4GB of RAM and a disk space of 500MB. Here are the steps to install the classic gazebo:

```
Terminal
$ sudo apt update
$ sudo apt upgrade -y
$ sudo apt install gazebo
$ gazebo
```

As simple as that, but things that might go wrong are just infinite. The last line should open the gazebo window that looks all-grey. If you don't then your system is probably missing some driver. The errors that gazebo spits out into the terminal are pretty verbose and you should be able to debug them yourself with the help of Stack Exchange (the Classic Generative Human Intelligence) or vibe with ChatGPT (Modern Generative Artificial Intelligence).

4.2 Creating a World

Now we need a world for the robot to spawn into. It is extremely simple to create one. We do so by placing 3D models readily available from open source libraries and placing

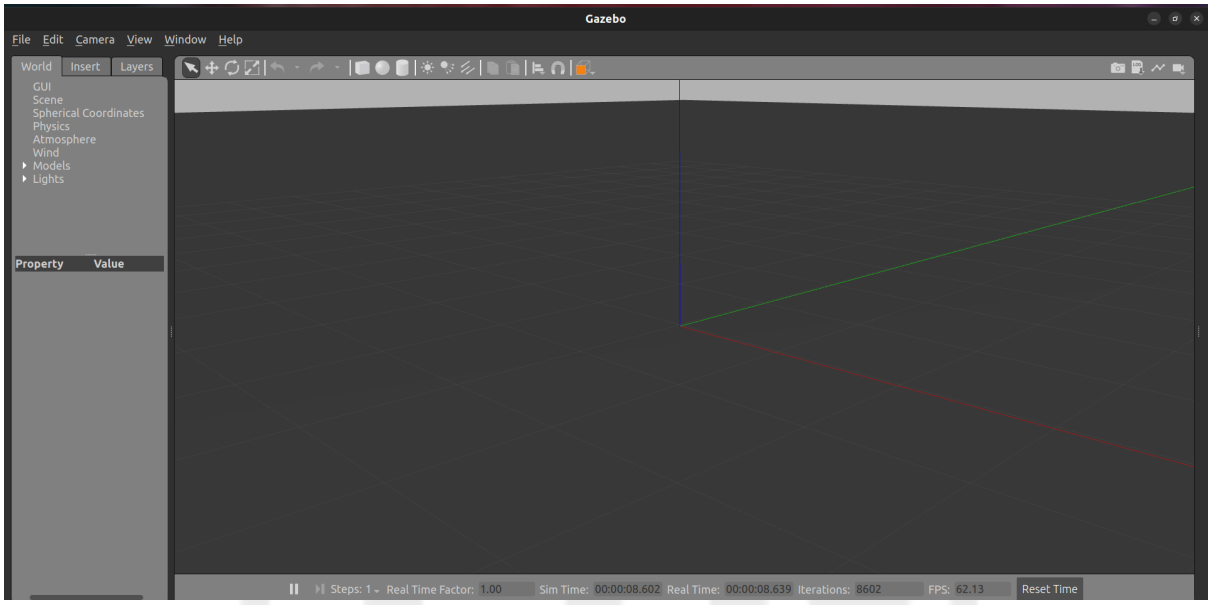

Figure 20: Launch Screen of Gazebo

them where we desire. Here's a step-wise walk through.

Step 1: Insert Tab

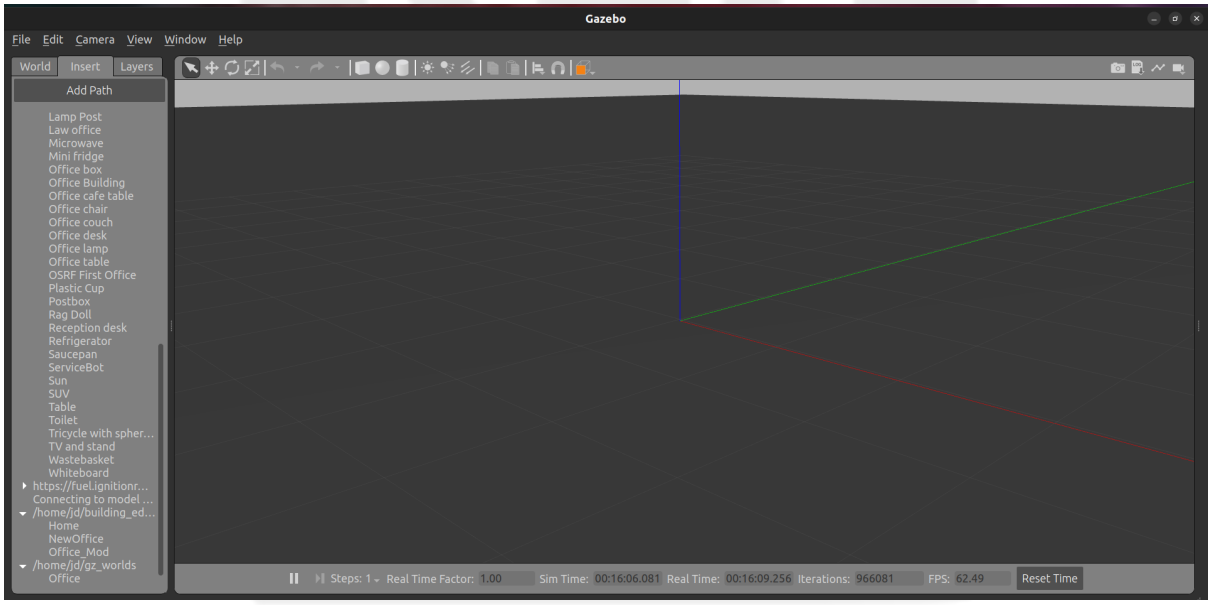

Figure 21: Step 1: Click Insert tab

First, go to Insert tab in the main screen. While doing this, ensure you have a strong internet connection. It might take a while to download the models depending on your internet speed. The screen might appear frozen with some messages like *Connecting to model database...* under the expandable menus. While it is extremely difficult to tell

apart a crash from a download loop based on the pane, you must check if the sim time progressing. If it is, then wait patiently for gazebo, else good luck figuring out why! If Gazebo does crash, the window will be closed immediately and errors would be printed in the terminal if you had launched gazebo through the terminal.

Step 2: Place & Orient the models

Begin by selecting a model you want from the list of available models.
Once placed, you can move a model by using the translation mode option.

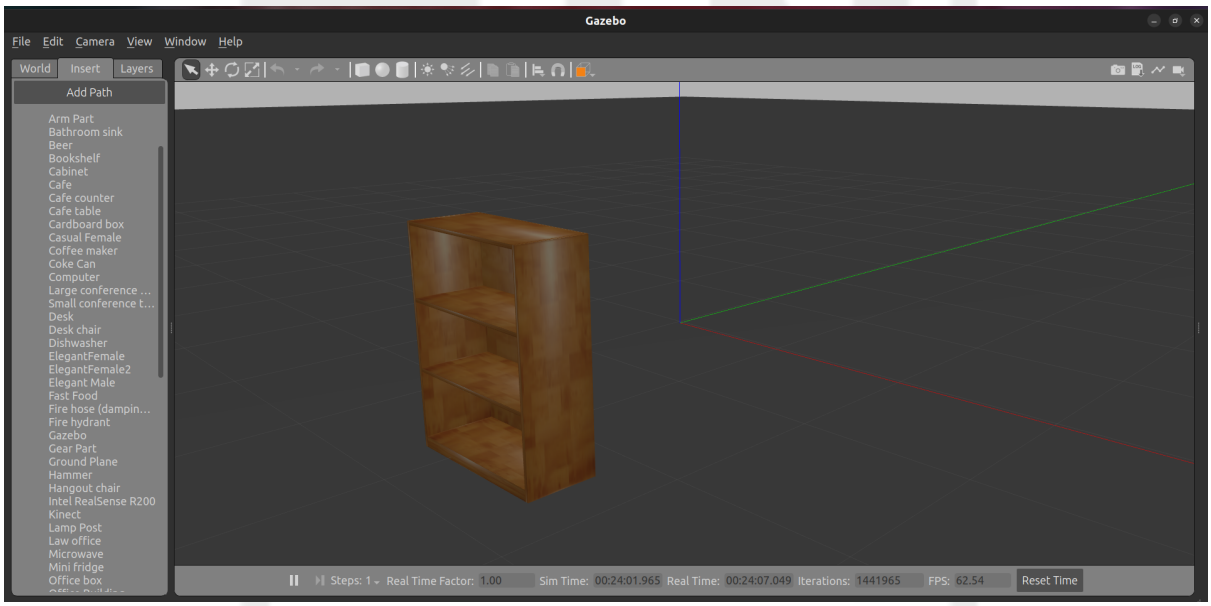

Figure 22: Step 2a: Select a model

You can also rotate the model using rotation mode.
Go on and place a couple of more models to make an interesting world. To move around in the world, the mouse actions are same as any CAD software:

- Scroll wheel to zoom in or out
- Ctrl+Move mouse to *translate* around in the world
- Click scroll wheel and move to *look* around the world

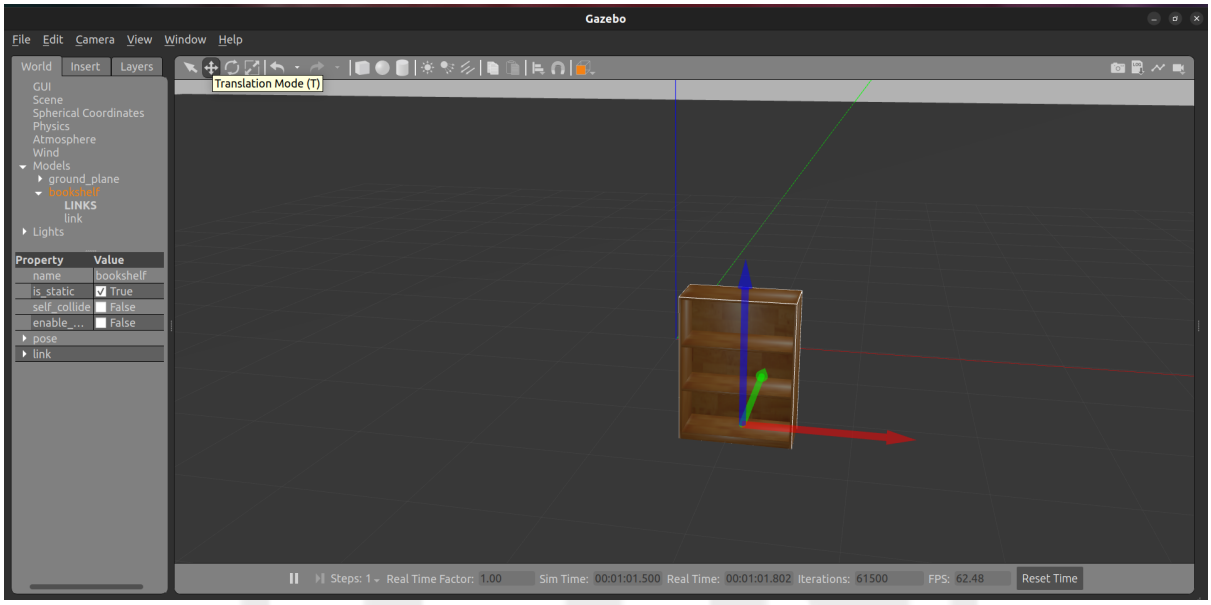

Figure 23: Step 2b: Move the model

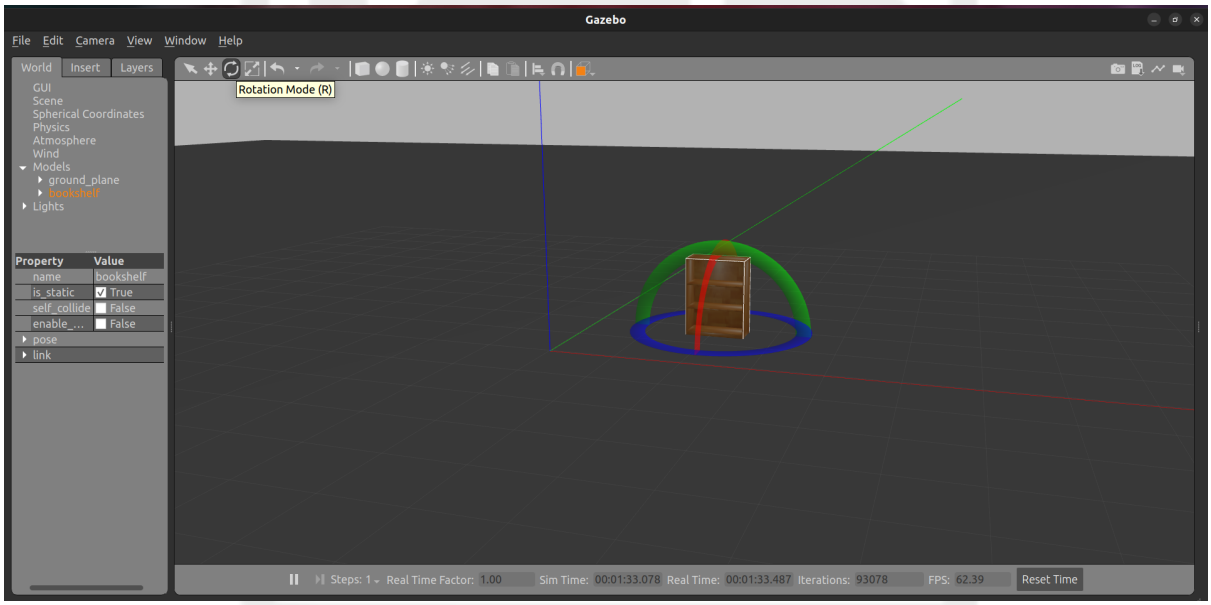

Figure 24: Step 2c: Rotate the model

Step 3: Create a Building

Its meaningful to draw walls and add storeys in order to make a building. Gazebo allows these seemingly complicated tasks with ease. Now you'll see a 2D grid at the top and the 3D world at the bottom. It might be unclear how the 2D plane is mapped to the 3D world's base (location of origin). So, select wall option and draw a line anywhere in the 2D grid. You will see the wall updated into the 3D world as you draw something on the grid. Add walls all around and place doors and/or windows. There are options to choose the texture of the walls too. You can also add storeys to your building. Next

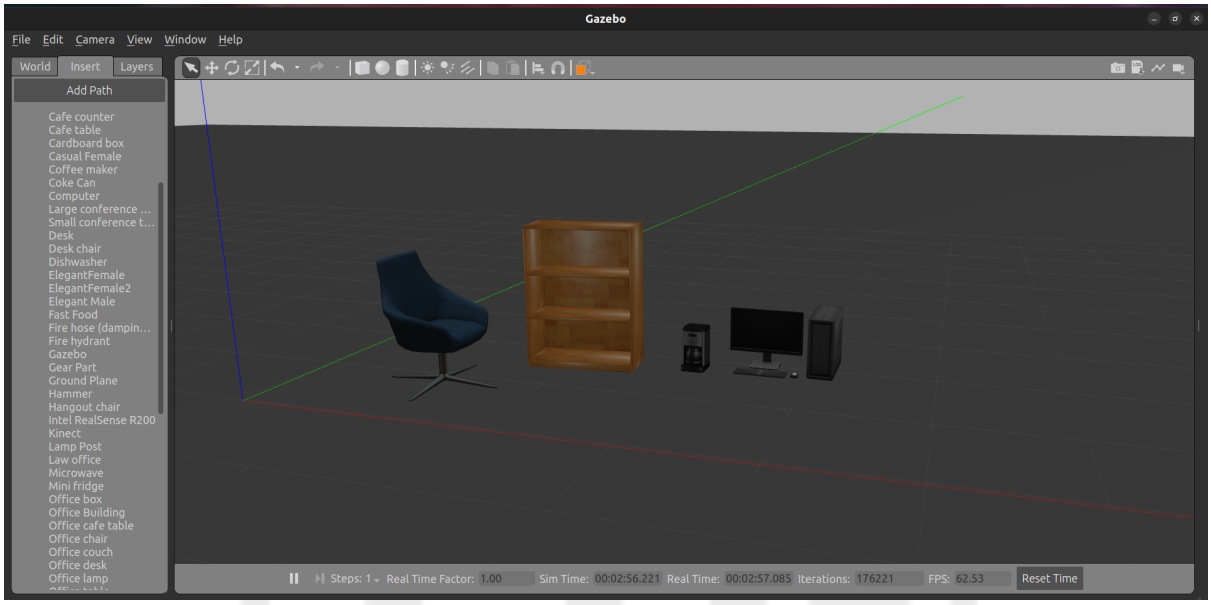

Figure 25: Step 2d: Add stuff

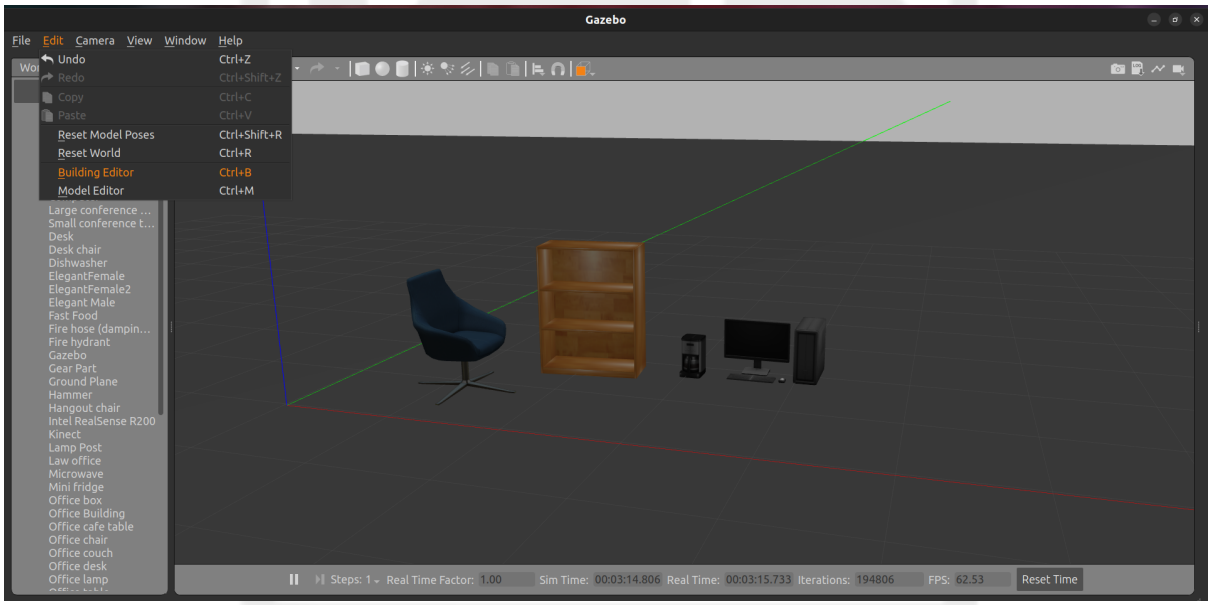

Figure 26: Step 3a: Select Building Editor

step is to save the building. You can either go to Save As option or just click exit to be prompted whether you want to save- and select save option. Once done, the changes will be reflected in the world. You may have a better look at the world with your mouse actions.

Step 4: Save the world

Explore the different options in the gazebo and perhaps add some more models to the world and save it. Go to File > Save World As and give a suitable name to your world.

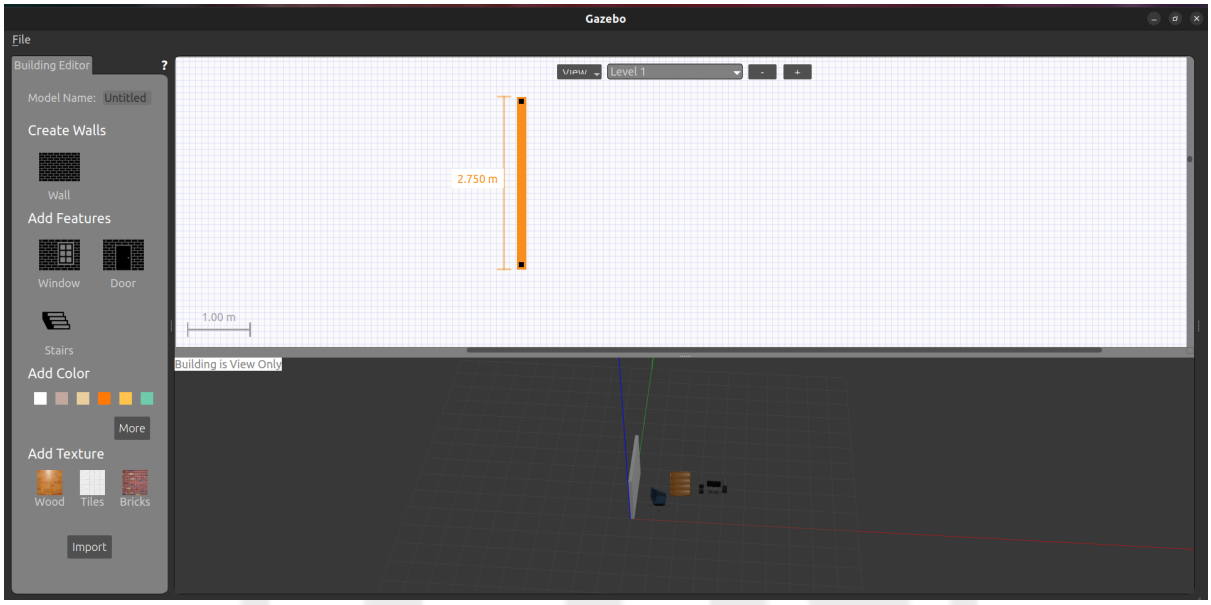

Figure 27: Step 3b: Select wall option and draw a wall

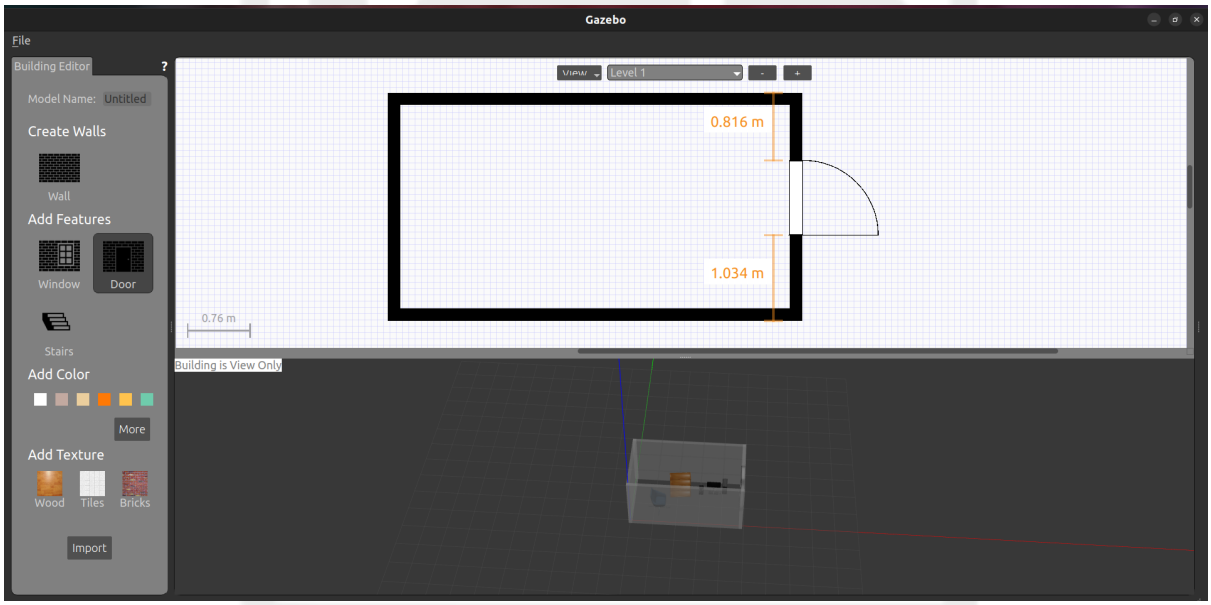

Figure 28: Step 3c: Complete a Room

This world was named *hello* and the file created is `hello.world`.

PRO TIP

While saving a world, save it under a `worlds` subfolder in your project directory

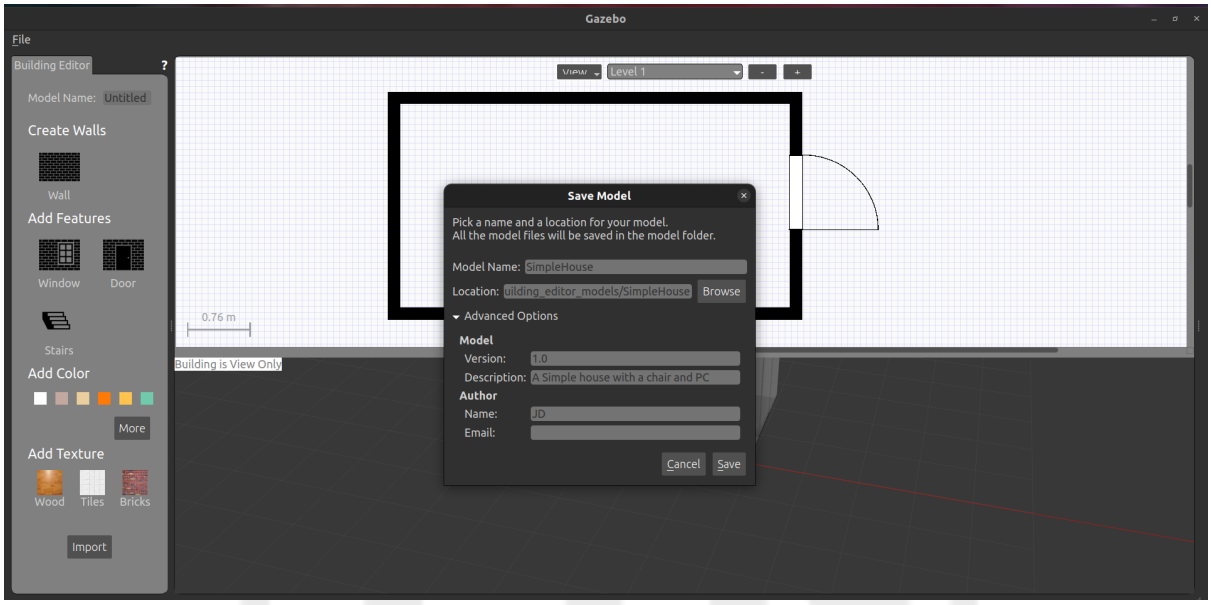

Figure 29: Step 3d: Save with details (optional)

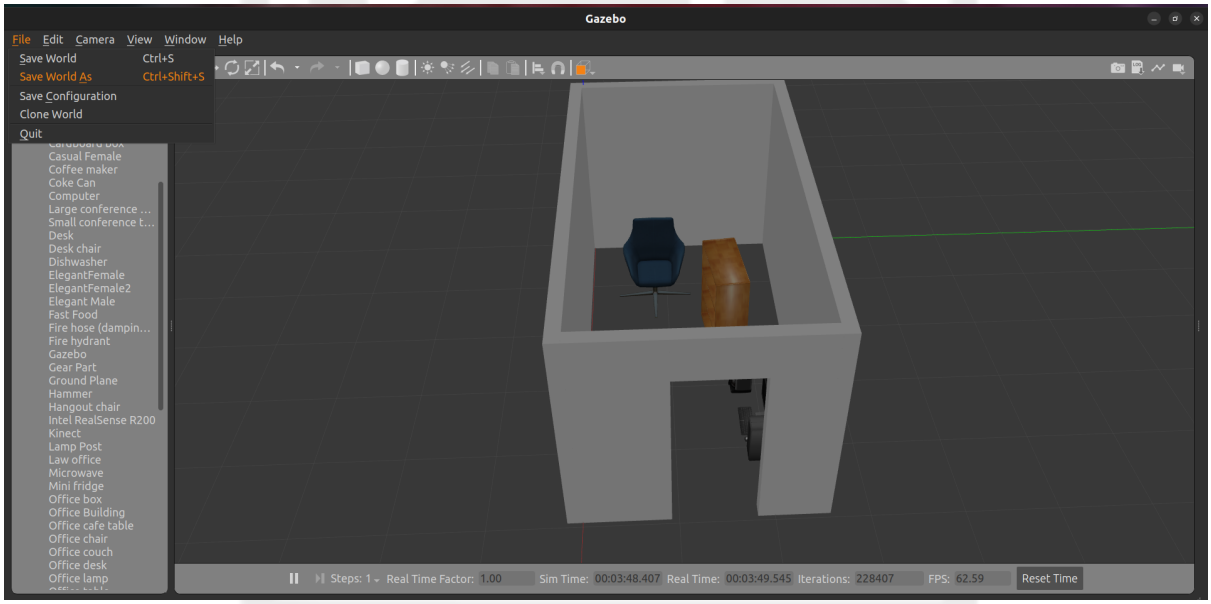

Figure 30: Step 3e: Ensure correct modelling

4.3 Launch File

As the drill has been, we shall be writing a launch file to make it easiier to launch all things at once. Now, here are the key changes you'll have to make to your existing launch file to launch gazebo.

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, IncludeLaunchDescription, ExecuteProcess
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import Node
from launch_ros.actions import SetParameter
```

```
from launch.substitutions import Command, LaunchConfiguration, PathJoinSubstitution
from launch_ros.substitutions import FindPackageShare

pkg_name = "bot"

def generate_launch_description():
    # the xacro file
    urdf_path = PathJoinSubstitution(
        [FindPackageShare(pkg_name), "description", "my_robot.urdf.xacro"]
    )

    # CHECK THIS OUT LATER! What are RViz configs and how to use them?
    '''
    rviz_config_path = PathJoinSubstitution(
        [FindPackageShare(pkg_name), "rviz", "robot_rviz_config.rviz"]
    )
    '''

    return LaunchDescription([

        # CHECK THIS OUT LATER! use gazebo time instead of system time
        SetParameter(name='use_sim_time', value=True),

        DeclareLaunchArgument(
            'urdf_path',
            default_value=urdf_path,
            description='Path to the URDF file'
        ),

        #DeclareLaunchArgument(
        #   'rviz_config_path',
        #   default_value=rviz_config_path,
        #   description='Path to the RViz config file'
        #),

        Node(
            package='rviz2',
            executable='rviz2',
            output='screen'
            #arguments=['-d', LaunchConfiguration('rviz_config_path')]
        ),

        Node(
            package='joint_state_publisher_gui',
            executable='joint_state_publisher_gui'
        ),

        ## IMPORTANT! mind the spaces in args passed to Command()
        ## they NEED to be present
        #  Change the robot name to what you want
        Node(
            package='robot_state_publisher',
            executable='robot_state_publisher',
            parameters=[{
                'robot_description': Command(['xacro ', LaunchConfiguration('urdf_path'),
                                              ' robot_name:=robot_0'])
            }]
        ),

        # Change hello.world to <your world's name>.world
        IncludeLaunchDescription(
            PythonLaunchDescriptionSource([
                PathJoinSubstitution([FindPackageShare('gazebo_ros'), 'launch', 'gazebo.launch
    .py'])
```

```
            ]),
            launch_arguments={'world': PathJoinSubstitution([FindPackageShare(pkg_name), '
    worlds', 'hello.world'])}.items()
        ),

        # Specify spawn params if you require
        Node(
            package='gazebo_ros',
            executable='spawn_entity.py',
            arguments=['-topic', 'robot_description', '-entity', 'my_robot', '-x', '1.0', '-y
    ', '1.0', '-z', '0.0']
        )
    ])
```

Make changes to the launch file or setup.py file if required and make sure the following are done:

- xacro files are in description subfolder
- world file is in world subfolder
- launch file is in launch subfolder
- these subfolders are added to the install directory in setup.py

Once this is done, build the package.

4.4 World through the Bot's eyes

Once built, run the package through the launch file. This might take significantly longer because of the number of nodes the launch file is launching. The launch file summons the robot state publisher, joint state publisher, RViz and gazebo all at once. Again, gazebo may take some time to download the models. Wait patiently till that is done. The screenshot show a bot spawned in a larger world, different from what was walked through. The goal here is to look at the world through the eyes of the robot, i.e, accessing the camera feed.

First, RViz needs to set up. There might be some global errors. RViz expects a base_link to always be present in the urdf. However, through the launch file, we have prefixed the robot name to every joint and link in the URDF file (find which line!). Therefore, we need to tell RViz which link it needs to treat as the base link.

Now, to add a camera, go to Add option under fixed frame and select Camera option

Again, you might get an error in the camera tab. This is because - as you might have guessed - we have not specified which topic the camera feed it coming from. List out the topics being published using the ros commands in a separate terminal and select the (most appropriate) camera topic. If you are not the developer of the URDF file of the robot you just spawned, there is a high chance you do not know which topic is that of the camera feed. You might have to tinker around and find it. It is therefore essential to name the topics appropriately. Look into the URDF file to see what the camera topic was named as.

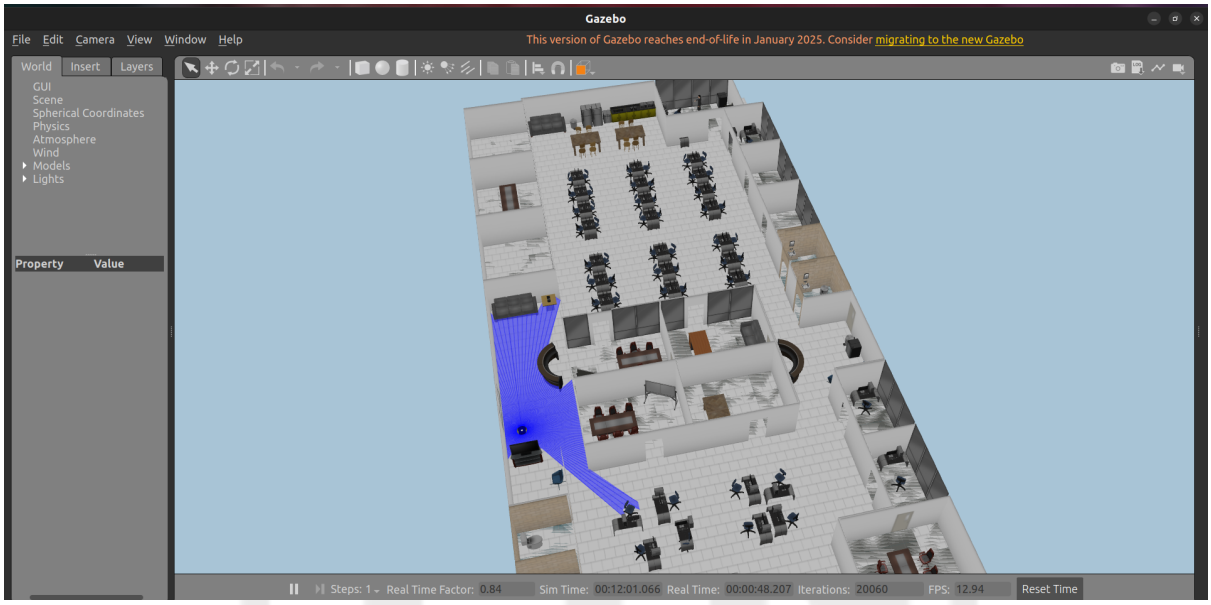

Figure 31: A bot spawned into a larger world

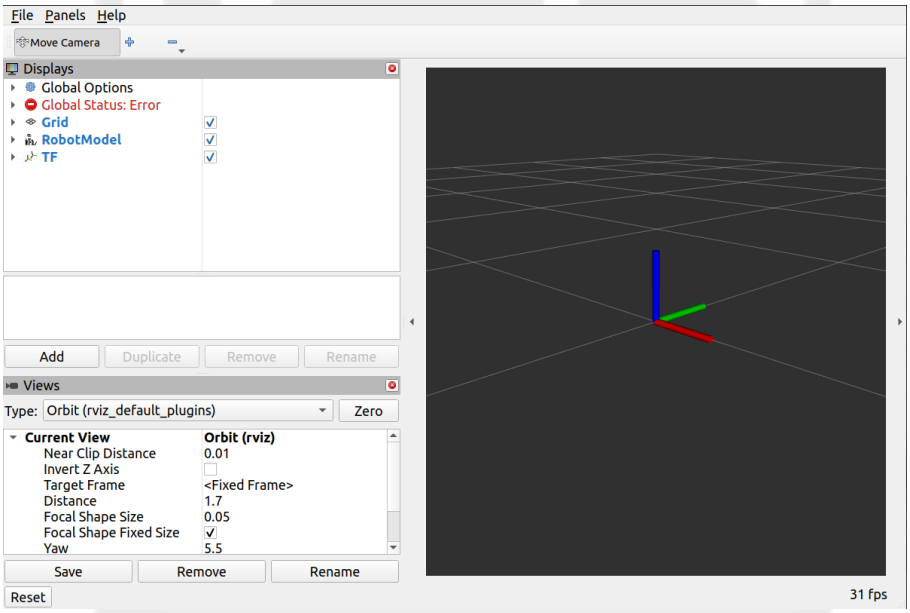

Figure 32: RViz Initial Window

Finally, select the camera topic.

Tada! The camera feed is now visible.

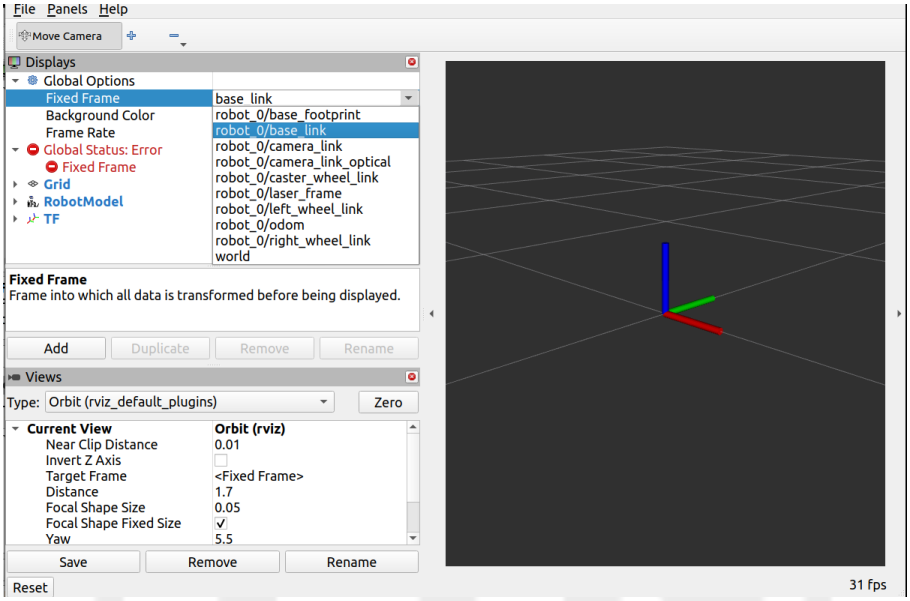

Figure 33: Selection of base_link

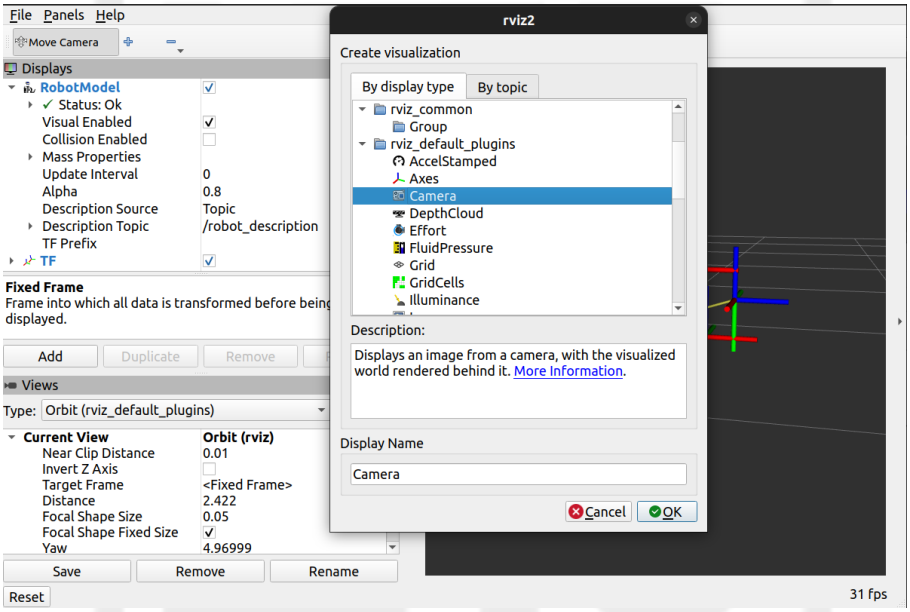

Figure 34: Adding camera

4.5 Moving the Bot around

The differential drive plugin listens to commands from `teleop_twist_keyboard` topic. We can therefore easily control the movement of the robot. Thing to keep in mind is that teleop publishes the data to `/cmd_vel` by default. However, as seen from the topic list, the topic of our robot is `/robot_0/cmd_vel`. Therefore, this has to be specifed to teleop and we do so using ros args. In another window, type the following command

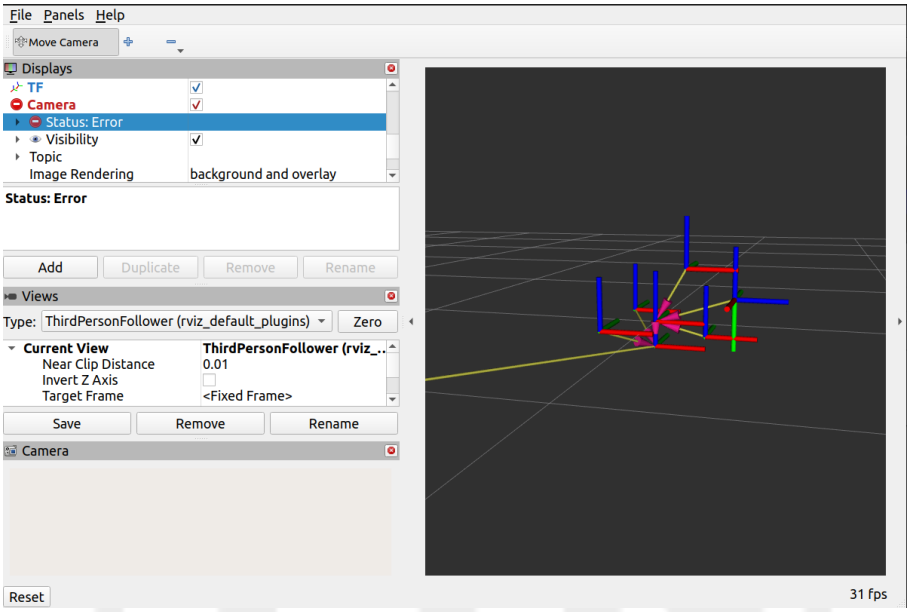

Figure 35: Select Camera topic

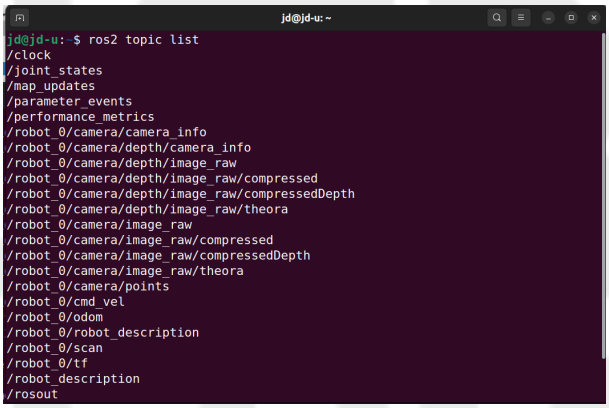

Figure 36: Listing topics to select Camera topic

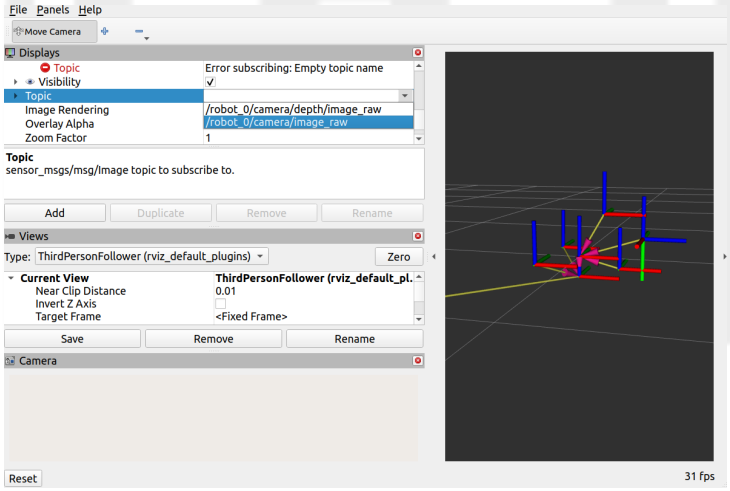

Figure 37: Select the Camera topic

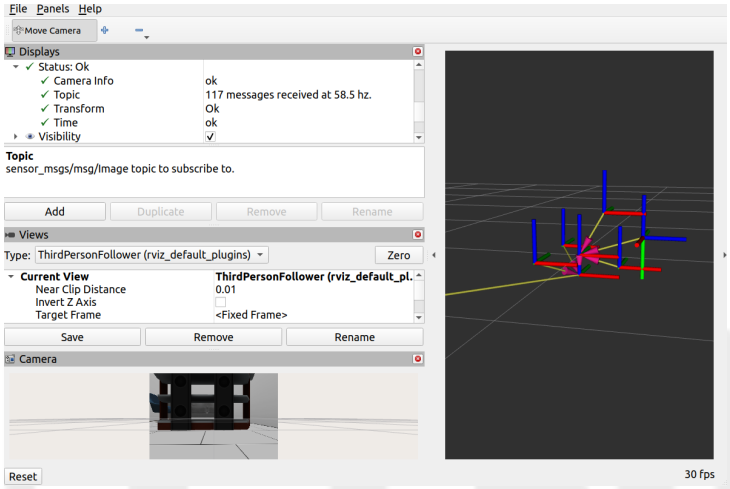

Figure 38: The camera feed appears!

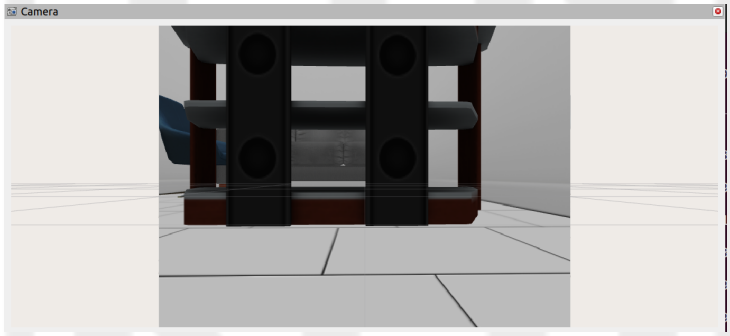

Figure 39: Camera feed can be popped out into a separate window

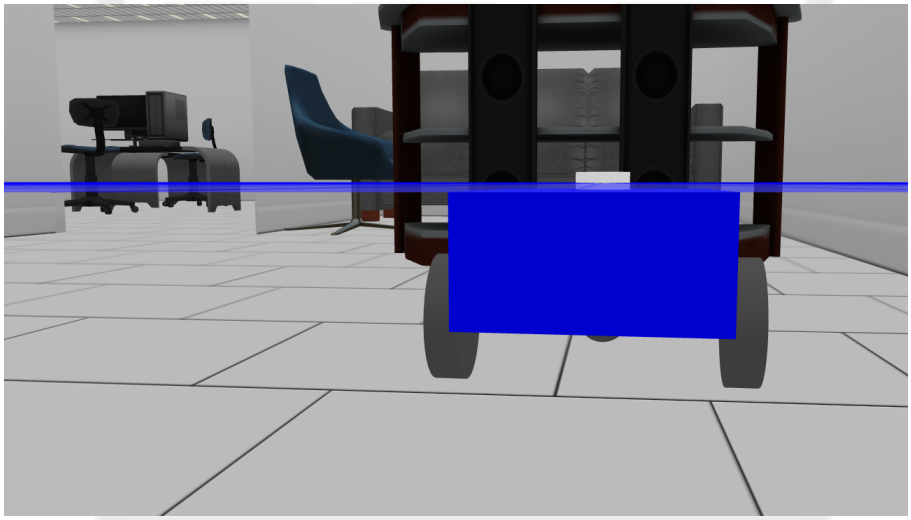

Figure 40: Robot's point of view in the world

```
Terminal                                                    - □ X
$ ros2 run teleop_twist_keyboard teleop_twist_keyboard --ros-args --remap
cmd_vel:=robot_0/cmd_vel
```

Now, move the bot around in the world and the camera feed will update in real time.

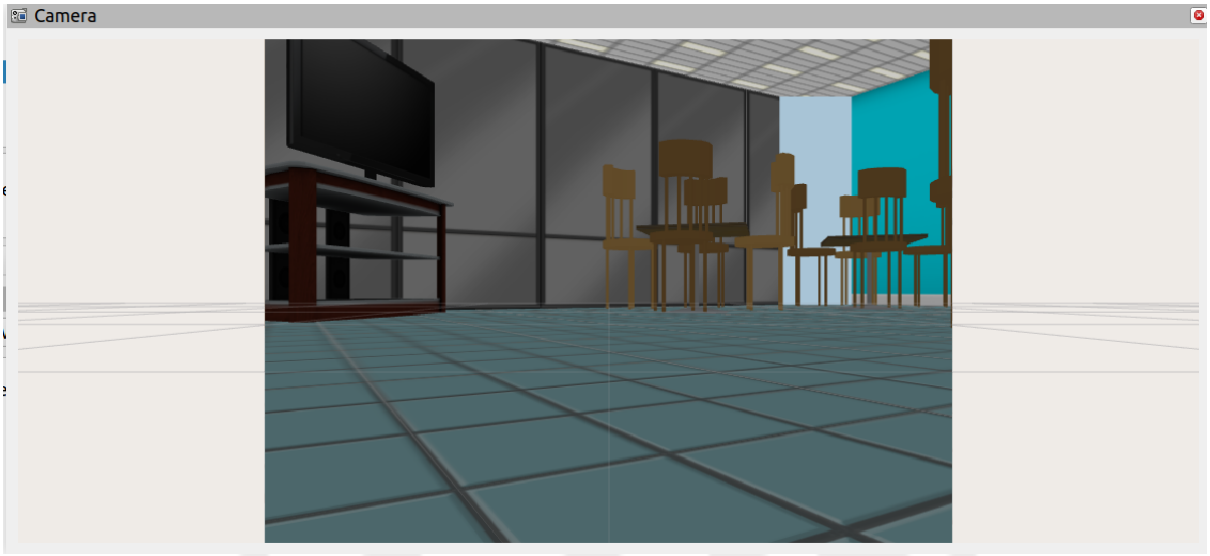

Figure 41: Camera feed from a different position of the bot

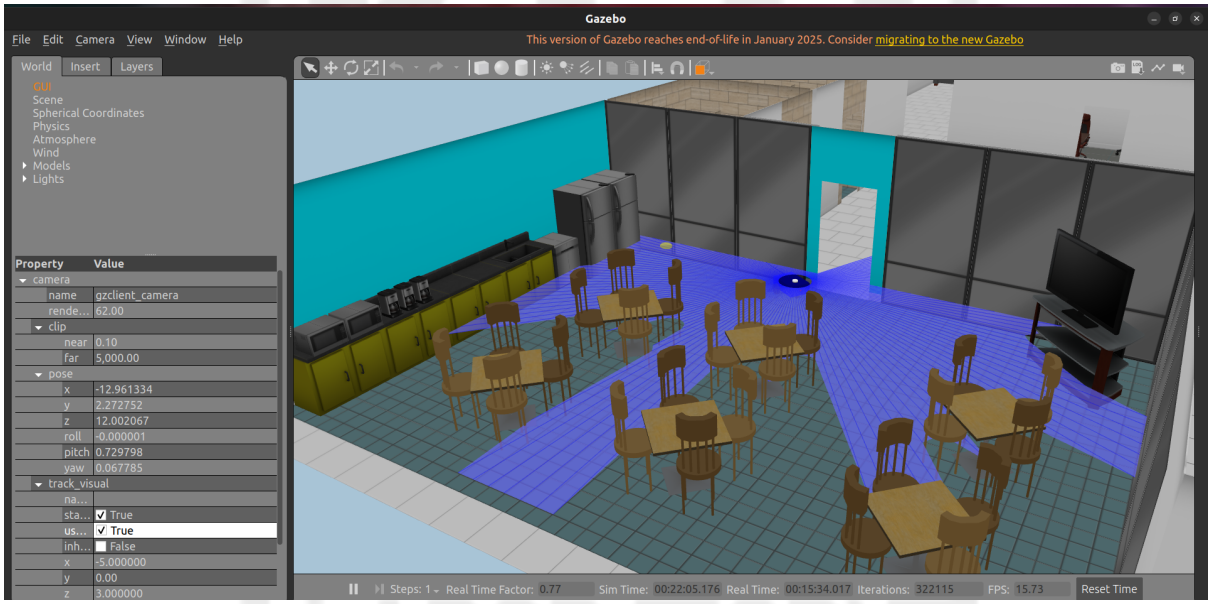

Figure 42: The bot's position in the world

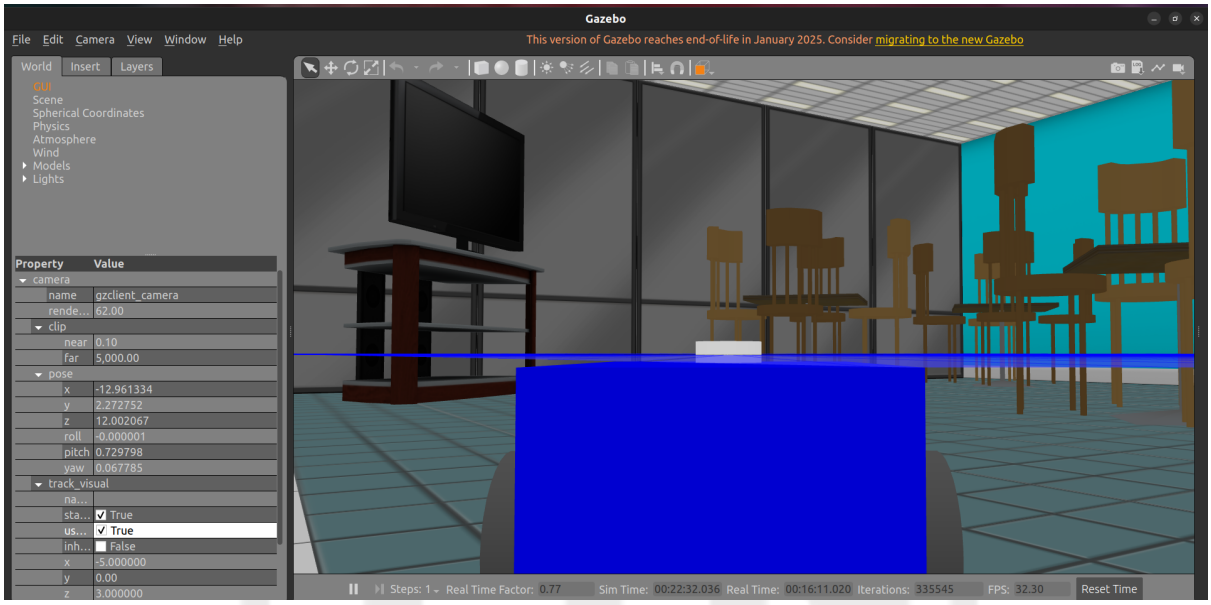

Figure 43: A larger view of the world

5 DIY Tasks

1. **Follow Along**
   Closely follow the instructions in this module to create a differential drive bot and spawn it in a custom world.

2. **Artoo's Day Out!**
   Describe Artoo-Deetoo (R2-D2) as a URDF file, ensuring atleast some details on its body along with appropriate colors. Artoo must have the following:

   - A Camera on the head
   - A Movable head, controlled with a custom topic
   - 4 wheels at the base, controlled with teleop

   Create a world with atleast two buildings and 10 different objects. The set up has to be realistic and meaningful. Navigate Artoo around the world.

Deliverables:
A **single pdf** file containing the following details for each of the tasks:

- Different views of the bot and the world
- Camera feed from at least two different locations in the world
- Architecture of the ROS code
- For the second task- mechanism used to rotate the head (top hemisphere)

Figure 44: Artoo-Deetoo (R2-D2)

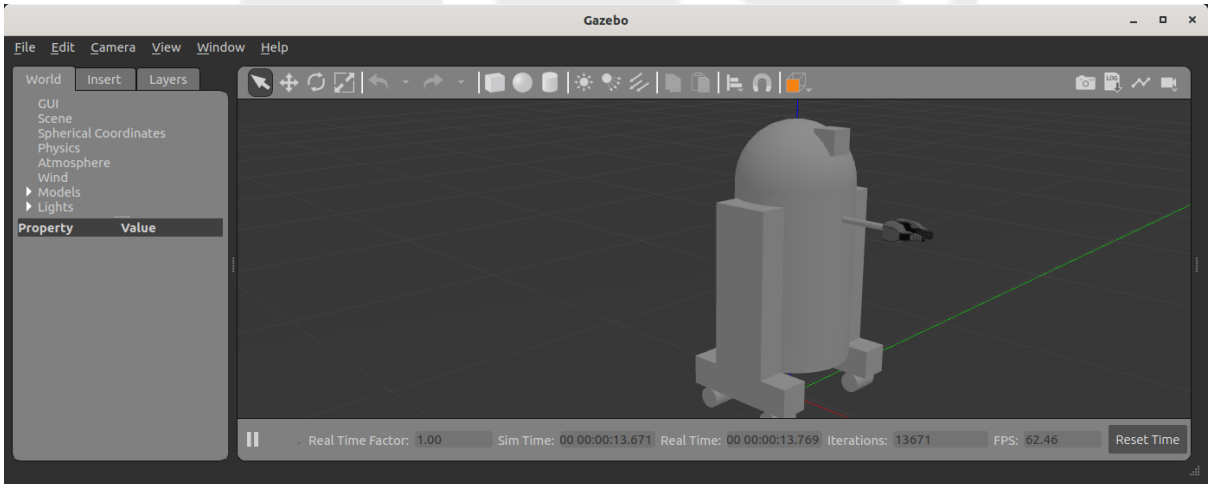

Figure 45: Expected URDF outcome for task 2- do include colors