Robotics Lab

Report Homework 1

Giulio Acampora P38000258
Alessandra Del Sorbo P38000289
Matteo Russo P38000247
Federico Trenti P38000263

2024/2025

Contents

1 **3**

1.1 Create the description of your robot and visualize it in Rviz 3

2 **7**

2.1 Add sensors and controllers to your robot and spawn it in Gazebo 7

3 **14**

3.1 Add a camera sensor to your robot 14

4 **18**

4.1 Create a ROS publisher node that reads the joint state and sends joint position commands to your robot 18

Chapter 1

1.1 Create the description of your robot and visualize it in Rviz

After we downloaded the arm_description package from the repository we focused on setting up a launch configuration for the arm_description package to facilitate the visualization and simulation of the robot model in RViz.

```
$ git clone https://github.com/RoboticsLab2024/
   arm_description.git
```

To begin, we created a new folder named launch within the arm_description package directory. This folder serves as the designated location for our launch file.
Next, we developed a launch file named display.launch.py within the launch folder. This file was configured to load the URDF model of the robot as a ROS parameter called robot_description (also, according to the following point, we already modified it to work as a xacro). Additionally, we specified the initialization of the nodes: robot_state_publisher, which publishes the state of the robot's joints; the joint_state_publisher, responsible for providing the current state of the joints; and the rviz2 node,to visualize the robot model.

```
# Path to the URDF file
    urdf_file = os.path.join(
        get_package_share_directory('arm_description'),
        'urdf',
        'arm.urdf.xacro'
    )

    robot_description_xacro = {"robot_description":
       Command(['xacro ', urdf_file])}

    joint_state_publisher_node = Node(
```

```
        package="joint_state_publisher_gui",
        executable="joint_state_publisher_gui",
        name="joint_state_publisher_gui",
        output="both"
    )

    robot_state_publisher_node = Node(
        package="robot_state_publisher",
        executable="robot_state_publisher",
        output="both",
        parameters=[robot_description_xacro,
                    {"use_sim_time": True},
            ],
    )

    rviz_node = Node(
        package="rviz2",
        executable="rviz2",
        name="rviz2",
        output="log",
        arguments=["-d", LaunchConfiguration("
           rviz_config_file")],  # OPTIONAL
    )
```

To launch the file, we used:

```
$ ros2 launch arm_description display.launch.py
```

The RViz results are displayed in Fig.1.1

As an optional step, we created a .rviz configuration file to save our RViz setup. This configuration ensures that the desired settings are automatically loaded each time we launch the visualization. We also modified the display.launch.py file to include this configuration file as an argument for the RViz node.

```
    declared_arguments = []

    declared_arguments.append(
        DeclareLaunchArgument(
            "rviz_config_file",
            default_value=PathJoinSubstitution(
                [FindPackageShare("arm_description"), "
                   config", "rviz", "arm.rviz"]
            ),
            description="RViz config file (absolute
               path) to use when launching rviz.",
```

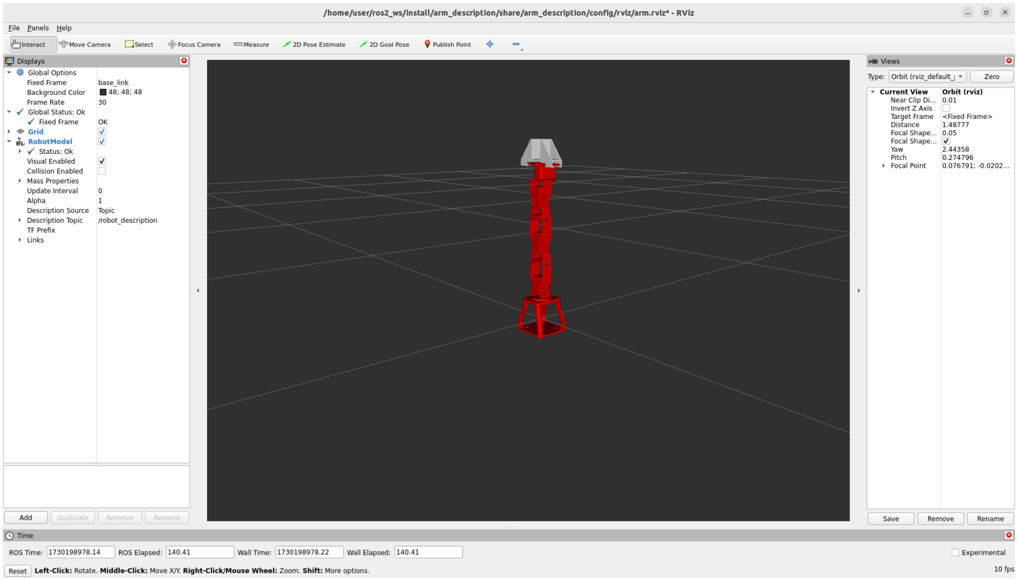

Figure 1.1: Manipulator

```
            )
        )
```

Listing 1.1: Code

In order for the launch file to work in the workspace we modified the CMakeLists.txt adding:

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

Next, we replaced the collision meshes with simpler primitive shapes, specifically using `<box>` geometries. These box shapes were carefully sized to approximate the dimensions of the robot's links. This simplification is expected to improve the computational efficiency during collision checks in simulation.
Something like this:

```
    <collision>
      <geometry>
        <box size="0.09 0.09 0.09"/> <!-- Dimensioni
            della box -->
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0.003"/>
    </collision>
```

You can see what the collisions look like in Fig.1.2.

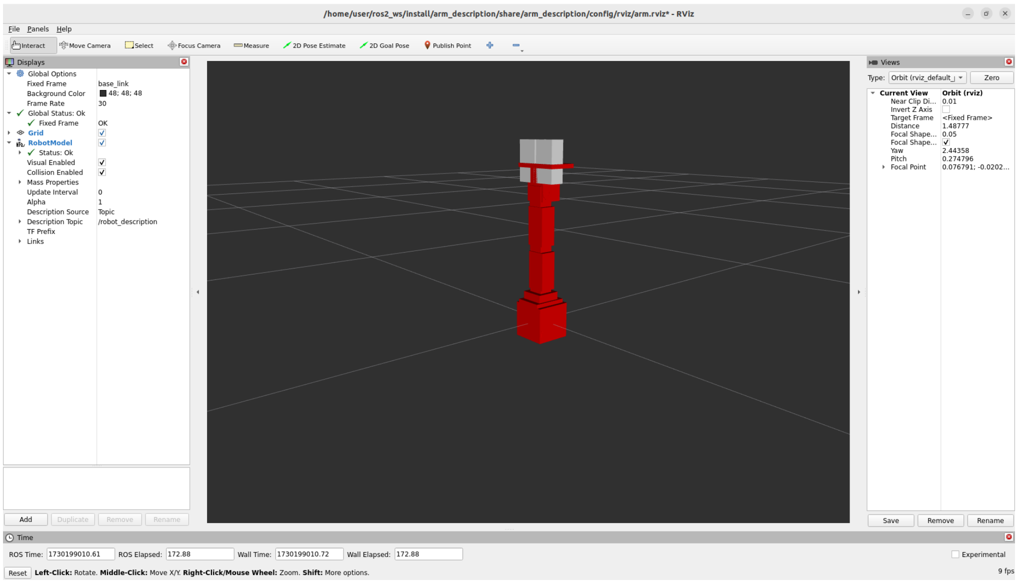

Figure 1.2: Manipulator with collisions.

Chapter 2

2.1 Add sensors and controllers to your robot and spawn it in Gazebo

We began by creating a new package named arm_gazebo. This package will serve as the foundation for our simulation setup.

```
$ ros2 pkg create --build-type ament_cmake arm_gazebo
```

Within the arm_gazebo package, we created a folder called launch. This folder is containing our launch files. Inside this folder, we created a launch file named arm_world.launch.

Next, we filled the arm_world.launch file with specific commands that would load the robot's URDF model into the robot_description topic.

```
    urdf_path = get_package_share_directory('arm_description')

    urdf_file = os.path.join(urdf_path, "urdf", "arm.urdf.xacro")

    with open(urdf_file, 'r') as infp:
        link_desc = infp.read()

    robot_description_xacro = {"robot_description": Command(['xacro ', urdf_file])}

    robot_state_publisher_node = Node(
        package="robot_state_publisher",
        executable="robot_state_publisher",
        output="both",
        parameters=[robot_description_xacro,
                    {"use_sim_time": True},
```

```
            ],
        remappings=[('/robot_description', '/robot_description')]
    )
```

Listing 2.1: Code

Additionally, we set up the file to spawn the robot using the create node from the ros_gz_sim package (also notice that we specified the position in such a way to spawn the robot above the ground).

```
    declared_arguments.append(DeclareLaunchArgument('gz_args', default_value='-r -v 1 empty.sdf',
                                  description='Arguments for gz_sim'),)

    gazebo_ignition = IncludeLaunchDescription(
            PythonLaunchDescriptionSource(
                [PathJoinSubstitution([FindPackageShare('ros_gz_sim'),
                                      'launch',
                                      'gz_sim.launch.py'])]),
            launch_arguments={'gz_args': LaunchConfiguration('gz_args')}.items()
    )

    position = [0.0, 0.0, 1.5]

    gz_spawn_entity = Node(
        package='ros_gz_sim',
        executable='create',
        output='screen',
        arguments=['-topic', 'robot_description',
                   '-name', 'arm',
                   '-allow_renaming', 'true',
                    "-x", str(position[0]),
                    "-y", str(position[1]),
                    "-z", str(position[2]),],
    )

    ign = [gazebo_ignition, gz_spawn_entity]
```

Listing 2.2: Code

In order to allow Gazebo to know where the meshes are, we added inside the package.xml file:

```
  <export>
    <build_type>ament_cmake</build_type>
    <gazebo_ros gazebo_model_path="${prefix}/.." />
  </export>
```

and also we moved the meshes folder from the arm_description package into this one, modifying opportunely the link path inside the URDF of arm_description:

```
        <geometry>
          <mesh filename="package://arm_gazebo/meshes/base_link.stl" scale="0.001 0.001 0.001"/>
        </geometry>
```

We implemented a PositionJointInterface as a hardware interface using ros2_control. To achieve this, we created a file named arm_hardware_interface.xacro within the arm_description/urdf folder. This file contains a macro that defines the hardware interface for each joint.

```
<?xml version="1.0" encoding="utf-8"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

<xacro:macro name="joint_ros2_control" params="name initial_pos">

    <joint name="${name}">
        <command_interface name="position"/>
        <state_interface name="position">
            <param name="initial_value">${initial_pos}</param>
        </state_interface>
        <state_interface name="velocity">
            <param name="initial_value">0.0</param>
        </state_interface>
        <state_interface name="effort">
            <param name="initial_value">0.0</param>
        </state_interface>
    </joint>

</xacro:macro>

</robot>
```

We then included this macro in our main arm.urdf.xacro file using the xacro:include directive. Furthermore, we defined each joint using ros2_control, specifying the hardware interface as PositionJointInterface.

```xml
<xacro:include filename="$(find arm_description)/urdf/arm_hardware_interface.xacro"/>
<ros2_control name="HardwareInterface_Ignition" type="system">

  <hardware>
  <plugin>ign_ros2_control/IgnitionSystem</plugin>
  </hardware>

  <xacro:joint_ros2_control name="base_1_2" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="j0" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="base_turn_rot_dyn2" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="j1" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="dyn2_f4" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="d4_dyn3" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="j2" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="dyn3_f5" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="f5_dyn4" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="j3" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="f5_wrist" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="wrist_dyn5" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="dyn5_dyn5_r" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="wrist_rotate_dyn5" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="crawer_base_crawer_left" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="crawer_base_crawer_right" initial_pos="0.0"/>

</ros2_control>
```

We updated the arm.urdf.xacro file to include commands for loading joint controller configurations from a YAML file.

```
<gazebo>
  <plugin filename="ign_ros2_control-system" name="
     ign_ros2_control::IgnitionROS2ControlPlugin">
  <parameters>$(find arm_control)/config/arm_control.
     yaml</parameters>
  <controller_manager_prefix_node_name>
     controller_manager</
     controller_manager_prefix_node_name>
  </plugin>
</gazebo>
```

As one can see in Fig.2.1 where the robot doesn't fall down, we ensured that the controllers were correctly spawned using the controller_manager package.

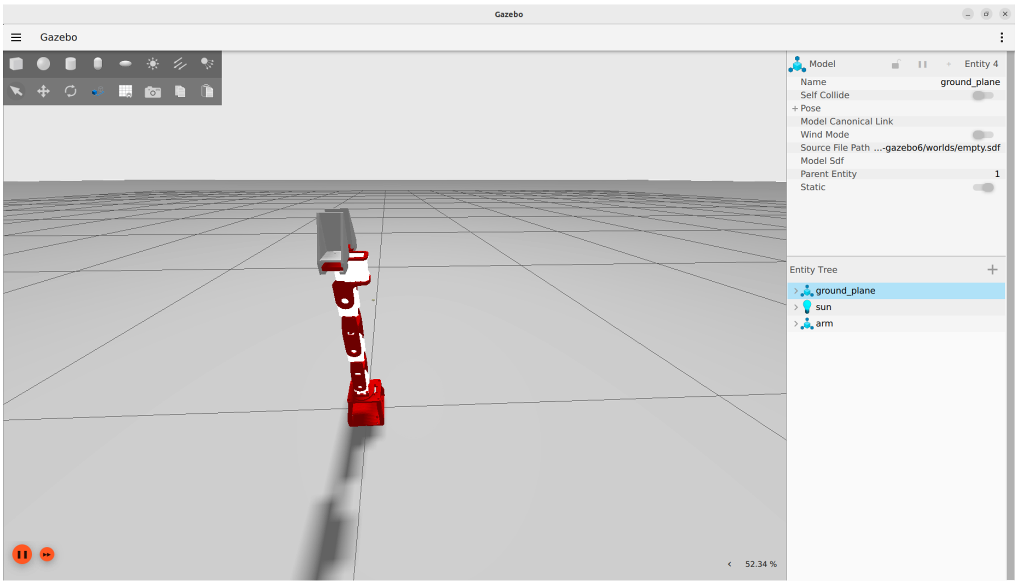

Figure 2.1: Manipulator with activated controllers

To manage the joint control for our robot, we created a new package named arm_control. This package contains all the configurations needed for controlling the robot's joints. Within the arm_control package, we established a launch file called arm_control.launch in the launch folder.

```
$ ros2 pkg create --build-type ament_cmake
  arm_gazebo
```

In the arm_control package, we created a YAML configuration file named arm_control.yaml within the config folder. This file defines a joint_state_broadcaster and a JointPositionController for each joint of the robot.

```
controller_manager:
  ros__parameters:
    update_rate: 225  # Hz

    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster

    position_controller:
      type: position_controllers/JointGroupPositionController

position_controller:
  ros__parameters:
    joints:
      - j0
      - j1
      - j2
      - j3
```

Then we created an arm_control.launch file to spawn the controllers:

```
    joint_state_broadcaster = Node(
        package="controller_manager",
        executable="spawner",
        arguments=["joint_state_broadcaster", "--controller-manager", "/controller_manager"],
    )

    position_controller = Node(
        package="controller_manager",
        executable="spawner",
        arguments=["position_controller", "--controller-manager", "/controller_manager"],
    )
```

Finally, we created an arm_gazebo.launch file in the launch folder of the arm_gazebo package. This launch file is responsible for coordinating the loading of the Gazebo world with the arm_world.launch file and for spawning the controllers using the arm_control.launch file. To make sure that the controller were spawned after the correct loading of Gazebo, we added a 5 seconds timer.

```
    arm_gazebo_launch = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([
            PathJoinSubstitution([
                FindPackageShare("arm_gazebo"),
```

```
                "launch",
                "arm_world.launch.py"
            ])
        ])
    )

    arm_control_launch = TimerAction(
        period=2.0,
        actions=[
            IncludeLaunchDescription(
                PythonLaunchDescriptionSource([
                    PathJoinSubstitution([
                        FindPackageShare("arm_control")
                            ,
                        "launch",
                        "arm_control.launch.py"
                    ])
                ])
            )
        ]
    )
```

Chapter 3

3.1 Add a camera sensor to your robot

In the arm.urdf.xacro file, we added a new camera_link and a fixed camera_joint connected to base_link as the parent. We made sure to size and position the camera link carefully, providing a clear view without obstructing any part of the robotic arm's movement.

```xml
<joint name="camera_joint" type="fixed">
    <parent link="base_link"/>
    <child link="camera_link"/>
    <origin xyz="0.0 0 0.00" rpy="0.0 0.0 1.57"/>
</joint>

<link name="camera_link">
    <visual>
        <geometry>
            <box size="0.0000010 0.00000003 0.00000003"
               />
        </geometry>
        <material name="red"/>
    </visual>
</link>
```

Listing 3.1: URDF per camera joint e camera link

We then created a file called arm_camera.xacro inside the arm_gazebo/urdf folder. This file includes the Gazebo sensor reference tags and the gz-sim-sensors-system plugin.

```xml
<?xml version="1.0" encoding="utf-8"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

  <xacro:macro name="camera_ros2_control">

    <gazebo>
      <plugin filename="gz-sim-sensors-system"
```

```xml
              name="gz::sim::systems::Sensors">
        <render_engine>ogre2</render_engine>
      </plugin>
    </gazebo>

    <!-- Configurazione della camera nel link specificato -->
    <gazebo reference="camera_link">
      <sensor name="camera" type="camera">
        <camera>
          <horizontal_fov>1.047</horizontal_fov>
          <image>
            <width>320</width>
            <height>240</height>
          </image>
          <clip>
            <near>0.1</near>
            <far>100</far>
          </clip>
        </camera>
        <always_on>1</always_on>
        <update_rate>30</update_rate>
        <visualize>true</visualize>
        <topic>camera</topic>
      </sensor>
    </gazebo>

  </xacro:macro>

</robot>
```

Listing 3.2: XML configuration for the camera in ROS2

Also inside the arm.urdf.xacro we added the include:

```xml
<xacro:include filename="$(find arm_gazebo)/urdf/arm_camera.xacro"/>
<xacro:camera_ros2_control/>
```

We launched the Gazebo simulation with arm_gazebo.launch and used rqt_image_view to check if the camera was publishing the image topic correctly. The command used is:

```
$ ros2 run rqt_image_view rqt_image_view
```

The topic published is showed in Fig.3.1.

With the addition of ros_ign_bridge, that allowed us to exchange topics and messages between the two environments so that everything ran smoothly.

```
bridge_camera = Node(
    package='ros_ign_bridge',
    executable='parameter_bridge',
    arguments=[
        '/camera@sensor_msgs/msg/Image@gz.msgs.Image',
        '/camera_info@sensor_msgs/msg/CameraInfo@gz.msgs.CameraInfo',
        '--ros-args',
        '-r', '/camera:=/videocamera',
    ],
    output='screen'
)
```

Listing 3.3: Bridge configuration in ROS2

Then we were able to view the camera's image output in real time as showed in Fig.3.2 and Fig.3.3. To show that everything works correctly we added a cube in the gazebo environment.

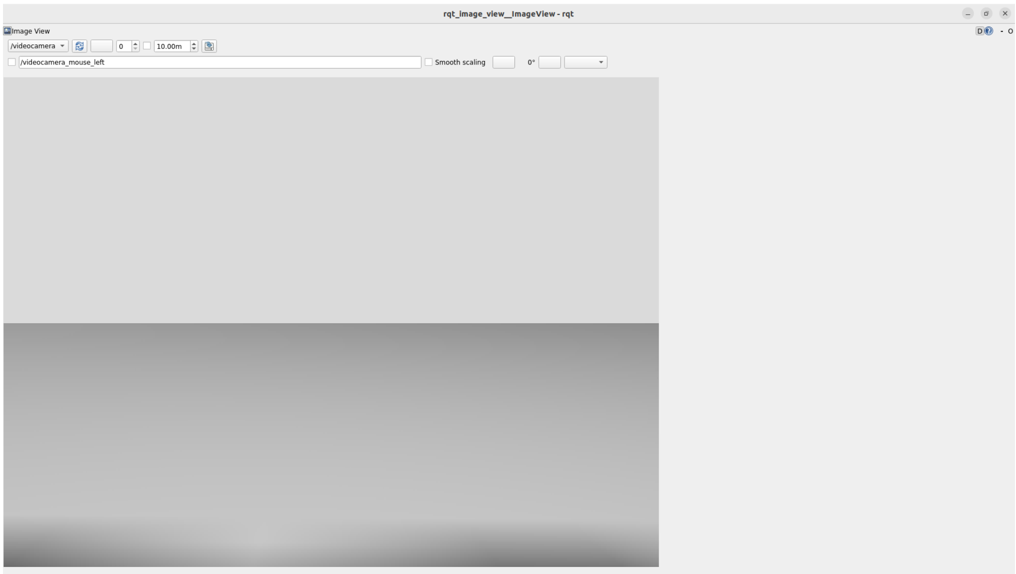

Figure 3.1: Rqt image view camera

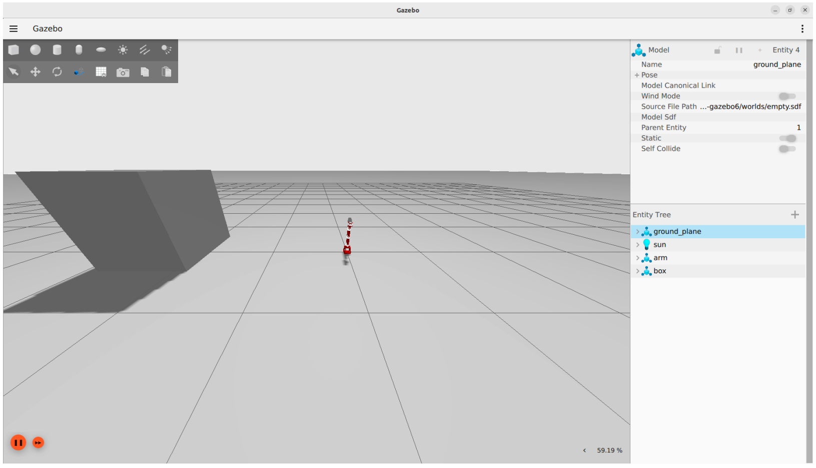

Figure 3.2: Manipulator and box in Gazebo

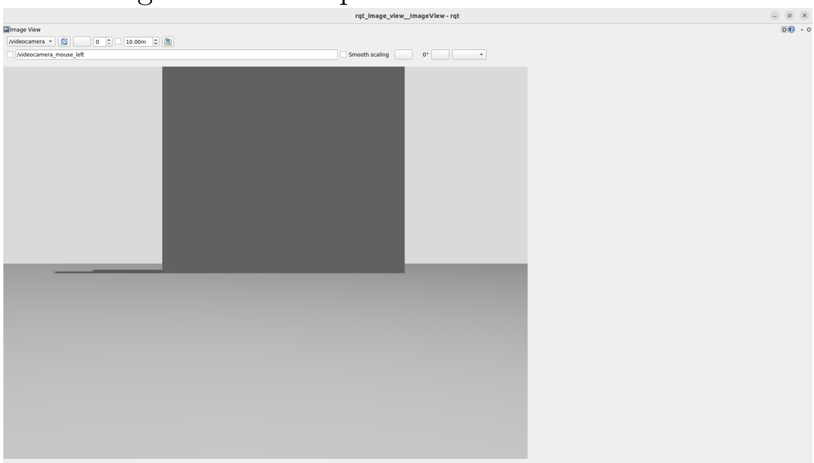

Figure 3.3: RQT visualization

Chapter 4

4.1 Create a ROS publisher node that reads the joint state and sends joint position commands to your robot

We started by creating a ROS C++ node called arm_controller_node within the arm_controller package.

```
$ ros2 pkg create --build-type ament_cmake
   arm_controller
```

The node's main job is to read joint states and send position commands to the robot. We added rclcpp, sensor_msgs, and std_msgs as dependencies in the package.xml, as they provide the necessary functions and message types for this task.

```
<depend>rclcpp</depend>
<depend>std_msgs</depend>
<depend>sensor_msgs</depend>
```

We also made sure to update the CMakeLists.txt file to compile the new node, using add_executable and ament_target_dependencies commands.

```
find_package(rclcpp REQUIRED)
find_package(sensor_msgs REQUIRED)
find_package(std_msgs REQUIRED)

# Aggiungi l eseguibile
add_executable(arm_controller_node src/
   arm_controller_node.cpp)

# Collega le dipendenze all eseguibile
ament_target_dependencies(arm_controller_node rclcpp
   sensor_msgs std_msgs)

```

```
install(TARGETS arm_controller_node DESTINATION lib/${
    PROJECT_NAME})
```

Next, we set up a subscriber for the joint_states topic, which contains sensor_msgs/JointState messages.

```
// Subscriber per il topic joint_states
joint_state_subscriber_ = this->create_subscription<
    sensor_msgs::msg::JointState>(
  "joint_states", 10, std::bind(&
      ArmControllerNode::jointStateCallback, this, _1));
```

A callback function was implemented to read and print the current joint positions every time a message is received on this topic. This gave us live feedback on joint states as showed in Fig.4.1.

```
// Funzione callback per il subscriber del topic
    joint_states
void jointStateCallback(const
    sensor_msgs::msg::JointState::SharedPtr msg)
{
    RCLCPP_INFO(this->get_logger(), "Current joint
        positions:");
    for (size_t i = 0; i < msg->position.size(); ++i) {
      RCLCPP_INFO(this->get_logger(), "  Joint %zu
          position: %f", i, msg->position[i]);
    }
}
```

Finally, we created a publisher to send position commands to the /position_controller/commands topics. This publisher publish messages in the std_msgs/msg/Float64MultiArray format, allowing us to control each joint's position by publishing appropriate command arrays.

```
// Publisher per il topic /position_controller/commands
command_publisher_ = this->create_publisher<
    std_msgs::msg::Float64MultiArray>("/
    position_controller/commands", 10);
```

To test that the node is working correctly, we specified a position for the joint of [1 1 1 1] as one can see in Fig.4.2. To clearly observe the robot's movement and verify that the publisher was working correctly, we added a 5-seconds timer. This delay allowed us to view each movement more distinctly.

```
timer_ = this->create_wall_timer(
  5000ms, std::bind(&ArmControllerNode::publishCommand,
      this));
```

```
void publishCommand()
  {
    auto command_msg = std_msgs::msg::Float64MultiArray();
    command_msg.data = {1.0, 1.0, 1.0, 1.0}; // Imposta il comando a (1, 1, 1, 1)
    RCLCPP_INFO(this->get_logger(), "Publishing command to position (1, 1, 1, 1)");

    command_publisher_->publish(command_msg);

    // Ferma il timer dopo aver inviato il comando una volta
    timer_->cancel();
  }
```

To run the arm_controller_node we used:

```
$ ros2 run arm_controller arm_controller_node
```

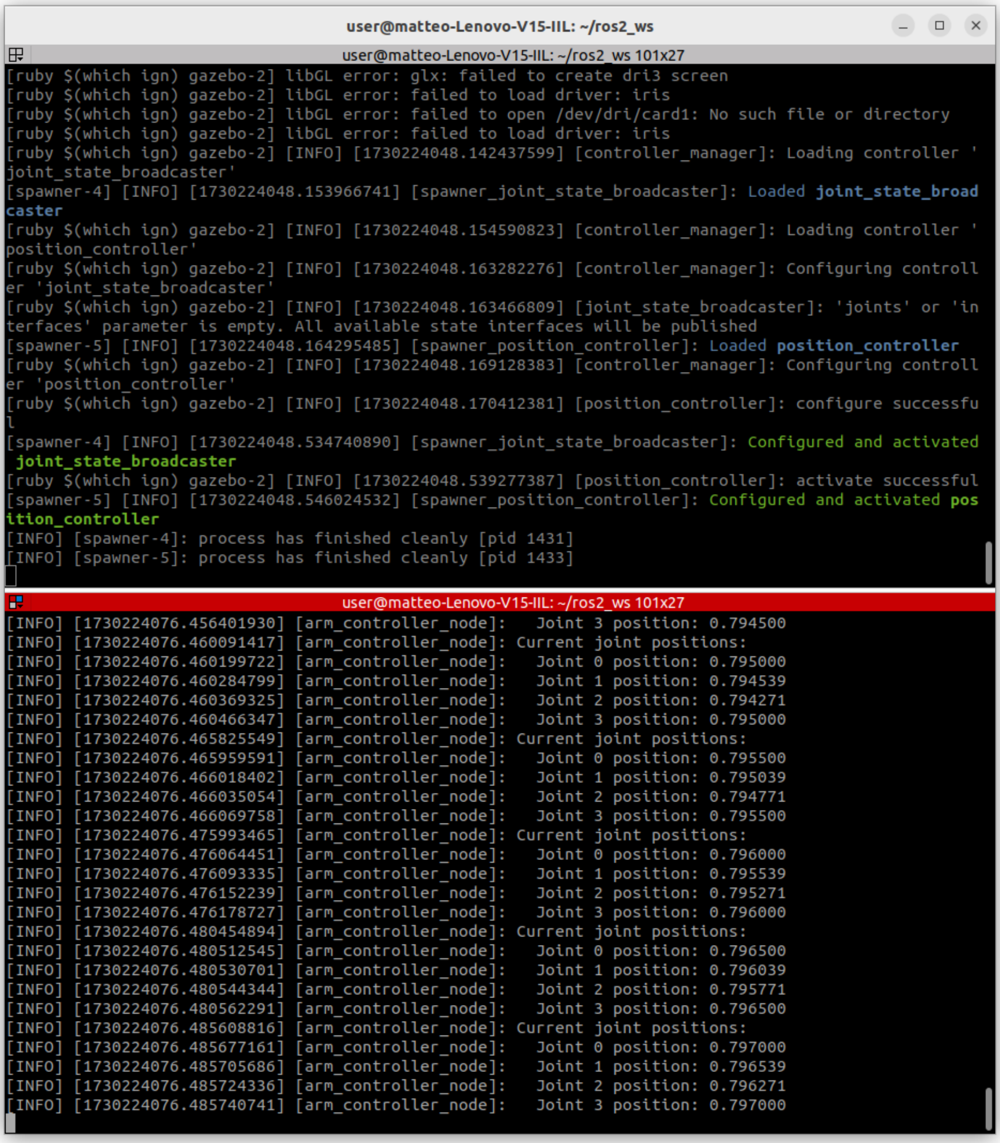

Figure 4.1: Terminal publisher subscriber

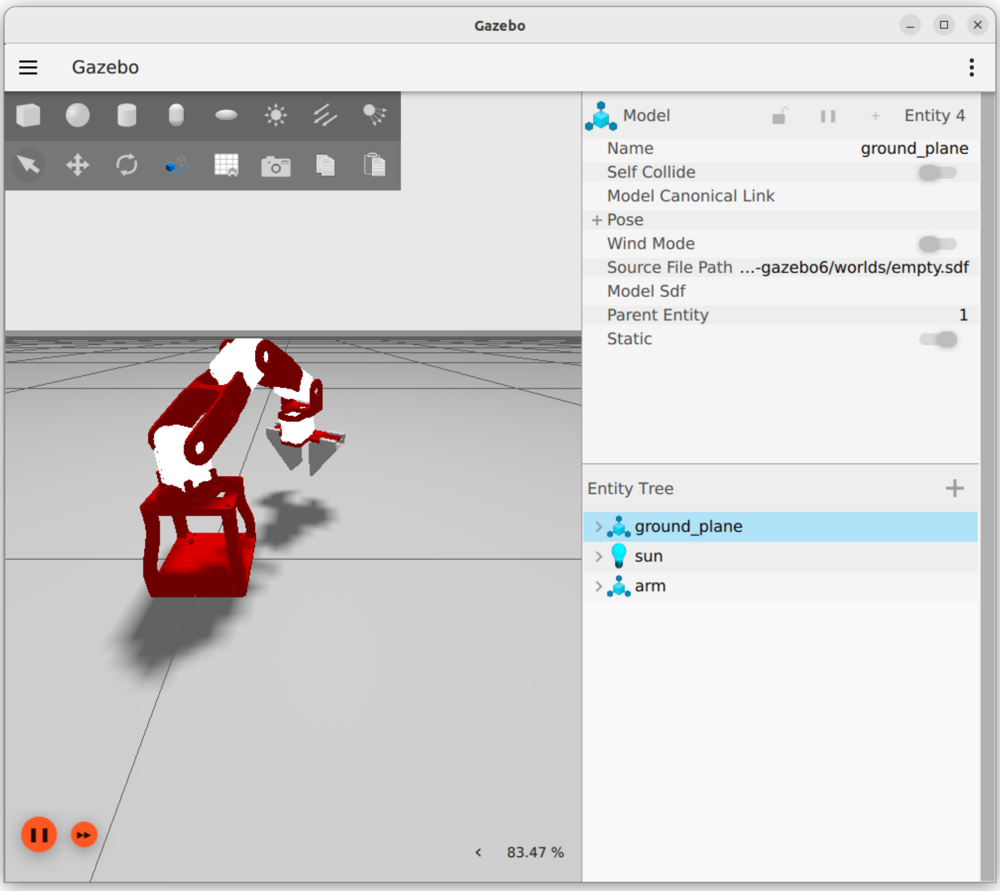

Figure 4.2: Robot after the publish command