Università degli Studi di Napoli

Federico II

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea in Ingegneria dell'Automazione e Robotica

Robotics Lab

Homework 1

Matteo De Simone P38000232, Giacomo Caiazzo P38000236, Marco Bartone P38000237, Nicola Monetti P38000238

Introduction

The following report documents the work conducted for Homework 1 of the Robotics Lab course. The objective of this assignment is to simulate the behavior of a 4-degree-of-freedom robotic manipulator in the Gazebo simulation environment, using ROS2. The document will be structured in sections, each presenting the steps followed to develop the different aspects of the homework.

1 Manipulator description and visualization in Rviz2

Using the terminal command "git clone," we downloaded the arm_description package, which contains the URDF and Stl files describing the manipulator. Once downloaded, we created a folder named "launch", which contains the launch file, named "display.launch.py", which loads the URDF and starts the robot_state_publisher node, the joint_state_publisher node, and the rviz2 node:

```python

arm_description_path = get_package_share_directory('arm_description')

urdf_arm_description = os.path.join(arm_description_path, "urdf", "arm.urdf")



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

robot_description_links = {"robot_description": link_desc}



joint_state_publisher_node = Node(
    package="joint_state_publisher_gui",
    executable="joint_state_publisher_gui",
)

robot_state_publisher_node_links = Node(
    package="robot_state_publisher", #ros2 run robot_state_publisher robot_state_publisher
    executable="robot_state_publisher",
    output="both",
    parameters=[ robot_description_links,
                {"use_sim_time": True},
        ],
    remappings=[('/robot_description', '/robot_description')]
)


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

nodes_to_start = [
    joint_state_publisher_node,
    robot_state_publisher_node_links,
    rviz_node
]

return LaunchDescription(declared_arguments + nodes_to_start)
```

Figure 1: arm.urdf

Using the command ”ros2 launch arm_description display.launch.py” we executed the launch file, which opened the Rviz2 screen containing the simulated model of the manipulator:

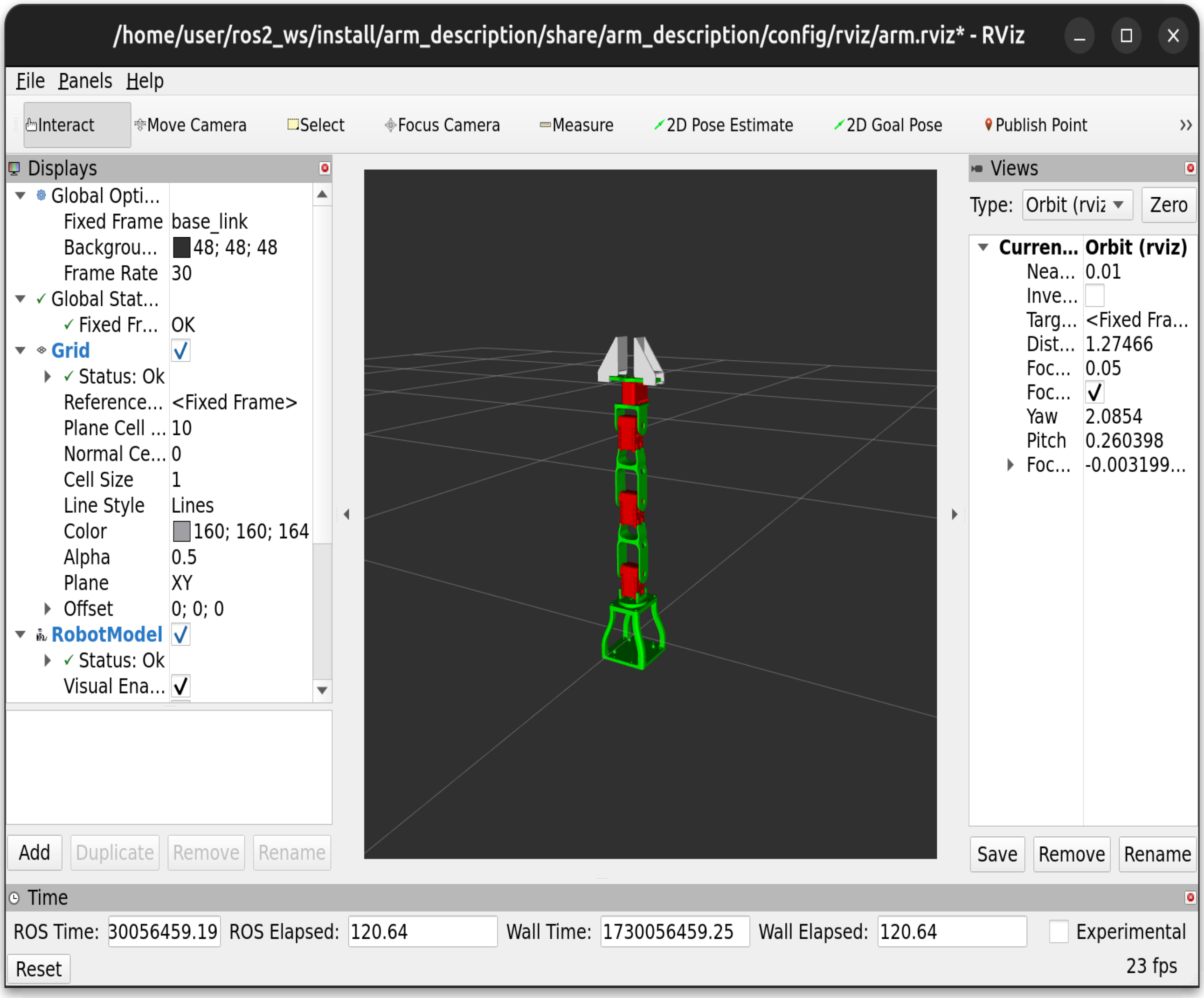

Figure 2: Rviz2 screen

The current rviz configuration has been saved in the /config/rviz folder as "arm.rviz", and can be loaded when rviz is launched with the following command:

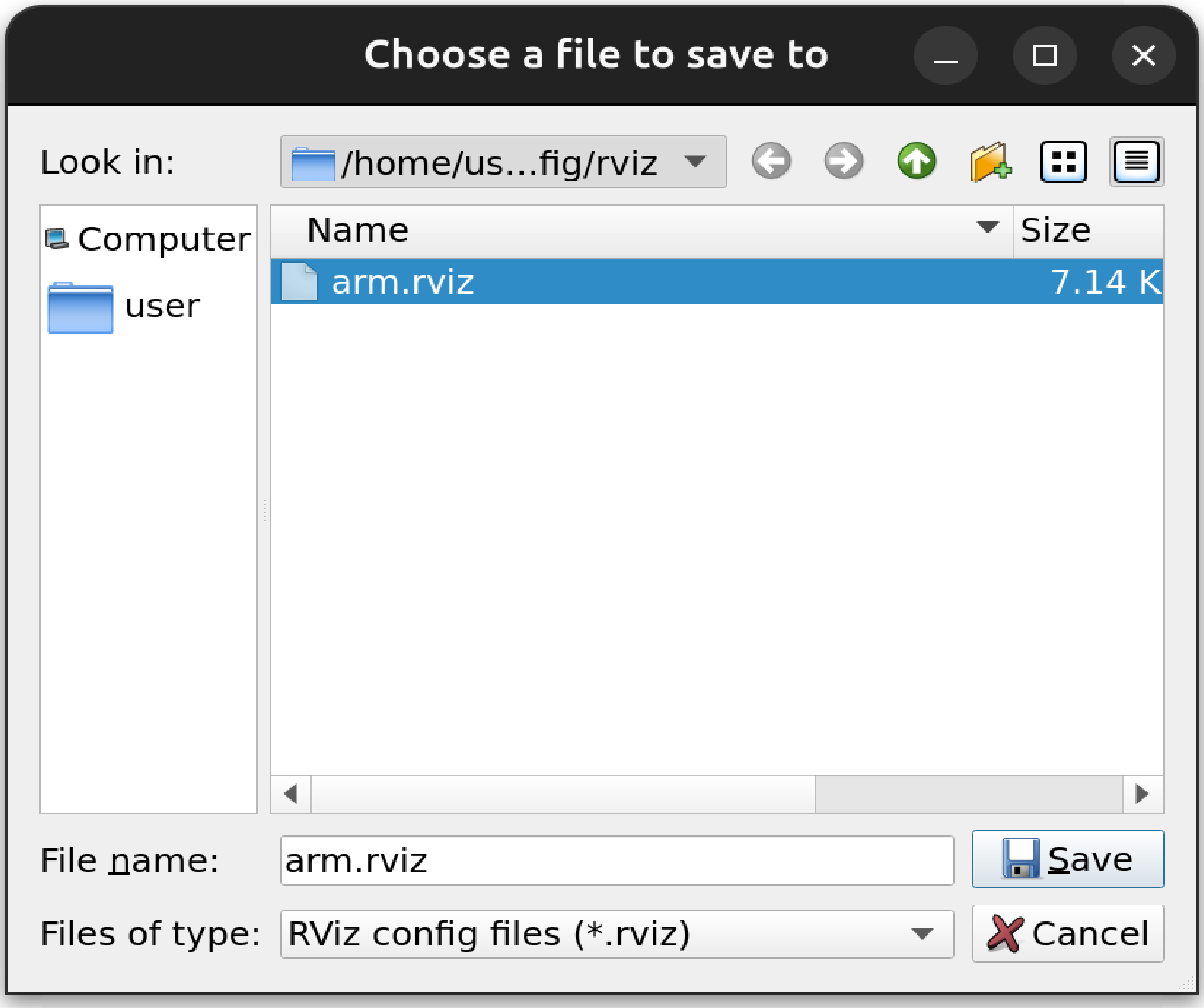

Figure 3: saving configuration

Below, we replaced the collision meshes of the manipulator in the URDF file using primitive shapes, specifically "box", modifying their dimensions to fit the elements of the manipulator:

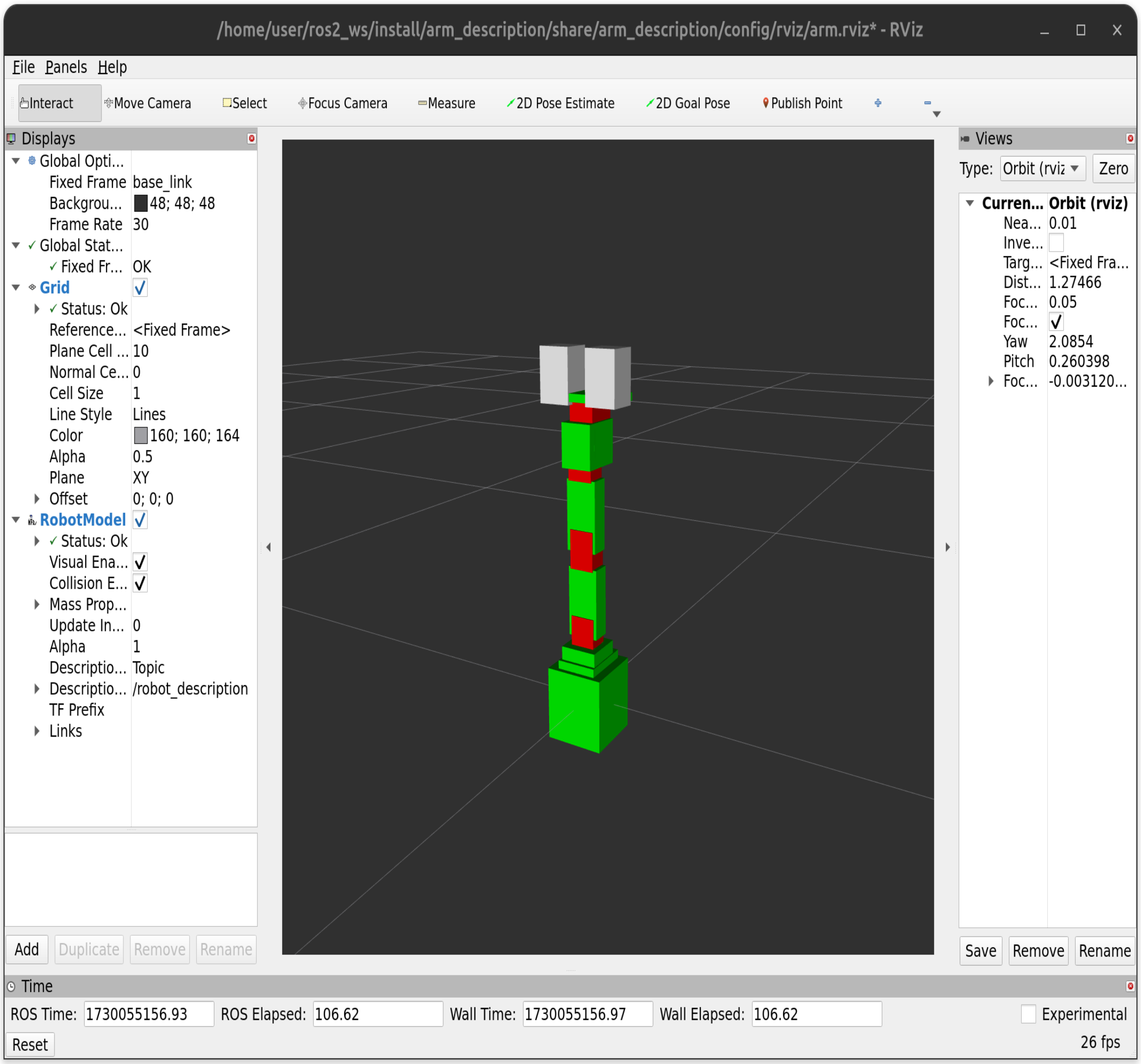

Figure 4: Rviz2 collision enabled

Some of these collision shapes were not centered with respect to the assigned elements, so the parameters related to the origin of these shapes, xyz, were modified:

```
<link name="base_turn">
   <visual>
     <geometry>
       <mesh filename="package://arm_description/meshes/base_turn.stl" scale="0.001 0.001 0.001"/>
     </geometry>
     <origin rpy="0 0 0" xyz="0 0 0"/>
     <material name="red"/>
   </visual>
   <collision>
     <geometry>
         <box size="0.065 0.08 0.012"/>
       </geometry>
     <origin rpy="0 0 0" xyz="0.0009 0.011 0.0025"/>
   </collision>
   <inertial>
     <mass value="0.1"/>
     <inertia ixx="1.35223008e+07" ixy="0.0" ixz="0.0" iyy="8.13814925e+06" iyz="0.0" izz="2.14903814e+07"/>
   </inertial>
 </link>
```

Figure 5: Collision shifted

2 Adding sensors and controllers and Gazebo visualization

An additional package, named arm_gazebo, has been created, which includes a launch file, arm_world.launch.py. This launch file is designed to load the URDF file and spawn the manipulator within the Gazebo simulation environment using the create node from the ros_gz_sim package:

```python
arm_description_path = get_package_share_directory('arm_description')

  arm_gazebo_path = get_package_share_directory('arm_gazebo')

  urdf_arm_gazebo = os.path.join(arm_description_path, "urdf", "arm.urdf")


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

  robot_description_links = {"robot_description": link_desc}



  joint_state_publisher_node = Node(
      package="joint_state_publisher_gui",
      executable="joint_state_publisher_gui",
  )

  robot_state_publisher_node_links = Node(
      package="robot_state_publisher", #ros2 run robot_state_publisher robot_state_publisher
      executable="robot_state_publisher",
      output="both",
      parameters=[robot_description_links,
                  {"use_sim_time": True},
          ],
      remappings=[('/robot_description', '/robot_description')]
  )

  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, 0.033]

  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]

  nodes_to_start = [
      joint_state_publisher_node,
      robot_state_publisher_node_links,
      *ign
  ]
```

Figure 6: arm_world.launch.py

In order to equip the manipulator with a hardware interface named PositionJointInterface, a xacro macro has been developed:

```xml
<xacro:macro name="define_arm_hardware_interface" 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>
```

Figure 7: arm_hardware.xacro

The xacro requires each joint, identified by the parameter name, to have an initial position and an initial velocity. In the case being examined, only the initial position is specified, therefore, it is treated as a parameter provided by the calling file, while the initial velocity is set to zero. This is made possible by the command_interface, which allows for the specification of position and velocity commands. Additionally, the xacro includes a state_interface, which enables the detection of the joint's state. The xacro thus constructed can be included in the URDF file as shown:

```
<xacro:include filename="$(find arm_description)/urdf/arm_hardware_interface.xacro"/>



  <ros2_control name="PositionJointInterface" type="system">

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

    <xacro:define_arm_hardware_interface name="j0" initial_pos="0.0"/>
    <xacro:define_arm_hardware_interface name="j1" initial_pos="0.0"/>
    <xacro:define_arm_hardware_interface name="j2" initial_pos="0.0"/>
    <xacro:define_arm_hardware_interface name="j3" initial_pos="0.0"/>

  </ros2_control>
```

Figure 8: ros2_control

The ros2_control framework provides a range of hardware interface types that can be used to create hardware components for specific robots or devices. Lines 444-447 reference the xacro for each of the four joints, passing the name and position as parameters.

It is important to note that a xacro macro cannot be called from a URDF file. To overcome this limitation, the physical and geometric description of the manipulator has been transferred to a urdf.xacro file. Subsequently, to load the robot in Gazebo, a conversion from urdf.xacro to .urdf has been implemented within arm_world.launch.py:

```python
urdf_arm_gazebo_xacro = os.path.join(arm_description_path, "urdf", "arm.urdf.xacro")

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

robot_description_links = {"robot_description": link_desc}

r_d_x = {"robot_description":Command(['xacro ',  urdf_arm_gazebo_xacro])}

joint_state_publisher_node = Node(
    package="joint_state_publisher_gui",
    executable="joint_state_publisher_gui",
)

robot_state_publisher_node_links = Node(
    package="robot_state_publisher", #ros2 run robot_state_publisher robot_state_publisher
    executable="robot_state_publisher",
    output="both",
    parameters=[r_d_x,
                {"use_sim_time": True},
        ],
    remappings=[('/robot_description', '/robot_description')]
)
```

Figure 9: input via xacro

Once the hardware interface has been correctly set up, it's necessary to launch the controller_manager node using the following:

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

Figure 10: control manager

To correctly set up the controller_manager a yaml configuration file is needed. At this point, through the Gazebo simulation, it's possible to observe the difference between the actuated and non actuated robot:

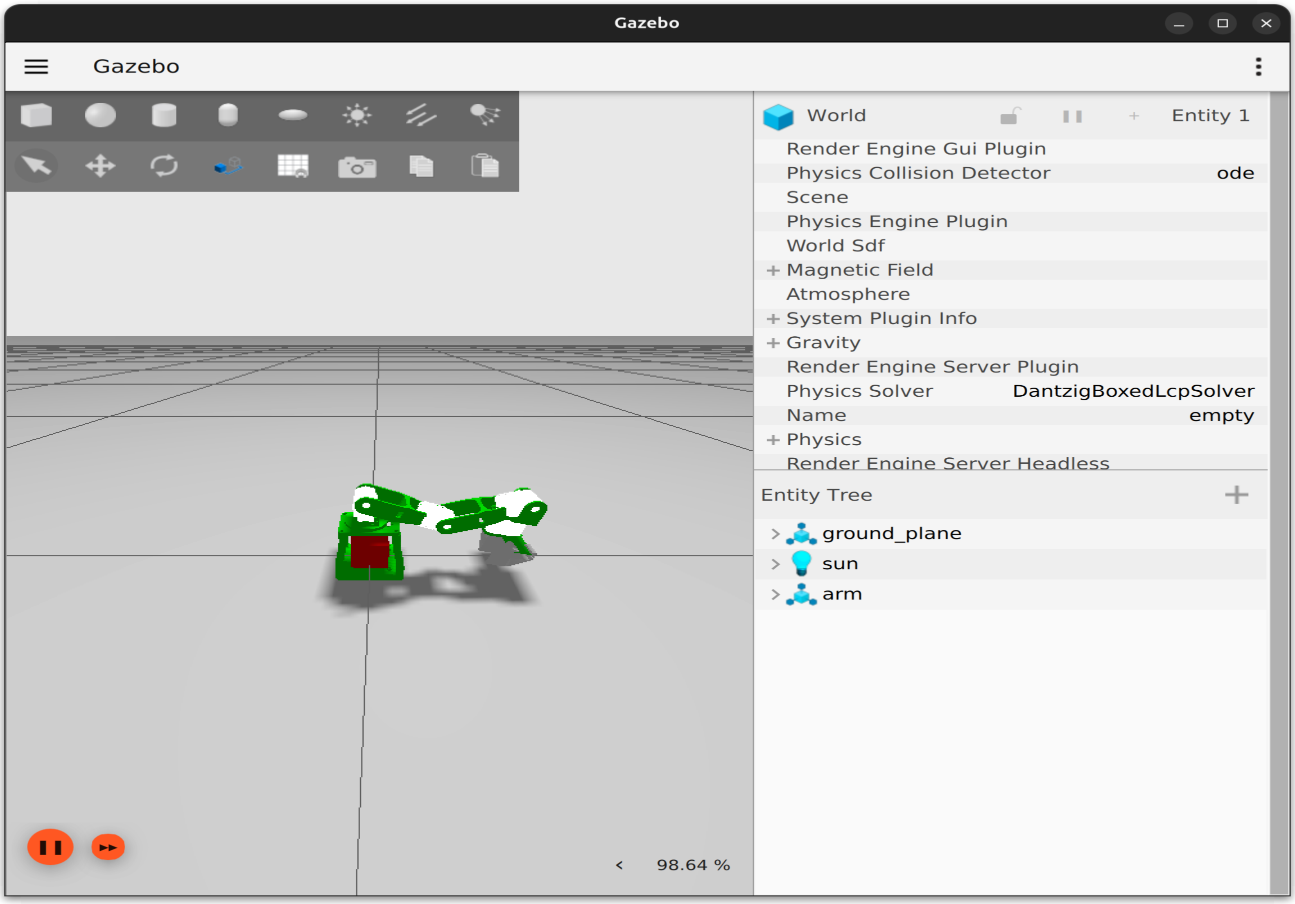

Figure 11: Manipulator without control manager active

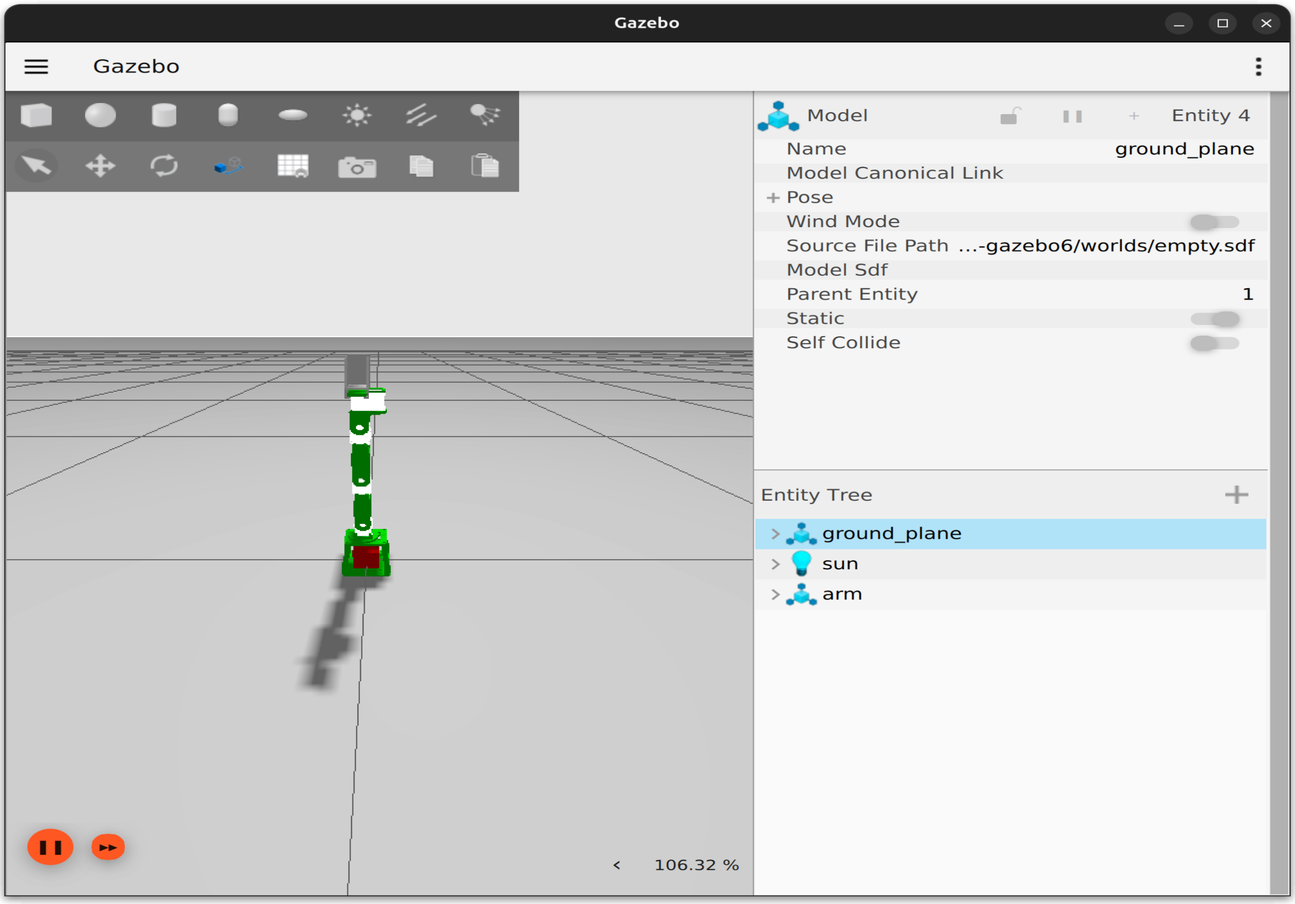

Figure 12: Manipulator with control manager active

In order to actively control robot's postion, the joint_position_controller has been defined, specifying in the yaml file as ros_parameters the four revolut joints. The same procedure needs to be carried out in order to add the joint_state_broadcaster:

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

    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster

    position_controller:
      type: position_controllers/JointGroupPositionController


    state_publish_rate: 200.0
    action_monitor_rate: 20.0

    allow_partial_joints_goal: true
    open_loop_control: true
    allow_integration_in_goal_trajectories: true


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

```

Figure 13: file controller.yaml

The final step is to create a arm_control.launch.py file which contains the instructions to launch the joint_state_broad and joint_position_controller nodes:

```python
def generate_launch_description():

    declared_arguments = []


    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"],
    )



    nodes_to_start = [
        position_controller,
        joint_state_broadcaster
    ]
```

Figure 14: arm_control.launch.py

In order to ease the execution of the whole project a arm_gazebo.launch.py file has been written including both the arm_world.launch.py and arm_control.launch.py files:

```
def generate_launch_description():

   arm_world = IncludeLaunchDescription(
      PythonLaunchDescriptionSource([os.path.join(
         get_package_share_directory('arm_gazebo'), 'launch'),
         '/arm_world.launch.py'])
      )

   arm_control = IncludeLaunchDescription(
      PythonLaunchDescriptionSource([os.path.join(
         get_package_share_directory('arm_control'), 'launch'),
         '/arm_control.launch.py'])
      )

   return LaunchDescription([
      arm_world,
      arm_control
   ])
```

Figure 15: arm_gazebo.launch.py

3 Adding a camera sensor to the robot

The first step to add a camera sensor to the robot manipulator was creating a camera link connected to the base link, through to a fixed camera joint:

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

  <link name="camera_link">
    <visual>
      <geometry>
       <box size="0.05 0.05 0.05"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <material name="red">
       <color rgba="1 0 0 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
       <box size="0.05 0.05 0.05"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
    </collision>
  </link>
```

Figure 16: camera_link in urdf.xacro

In order to effectively set up the sensor it's necessary write a arm_camera.xacro which attaches the camera sensor to the camera link using the gz-sim-sensors-system plugin:

```
<xacro:macro name="arm_camera" params="reference">

   <gazebo>
    <plugin filename="gz-sim-sensors-system" name="gz::sim::systems::Sensors">
      <render_engine>ogre2</render_engine>
    </plugin>
   </gazebo>

  <gazebo reference="${reference}">
    <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>
```

Figure 17: arm_camera.xacro

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

  <xacro:arm_camera reference="camera_link"/>

```

Figure 18: camera in urdf.xacro

It's now possible to appreciate the camera point of view using the Gazebo simulation and rqt_image_view:

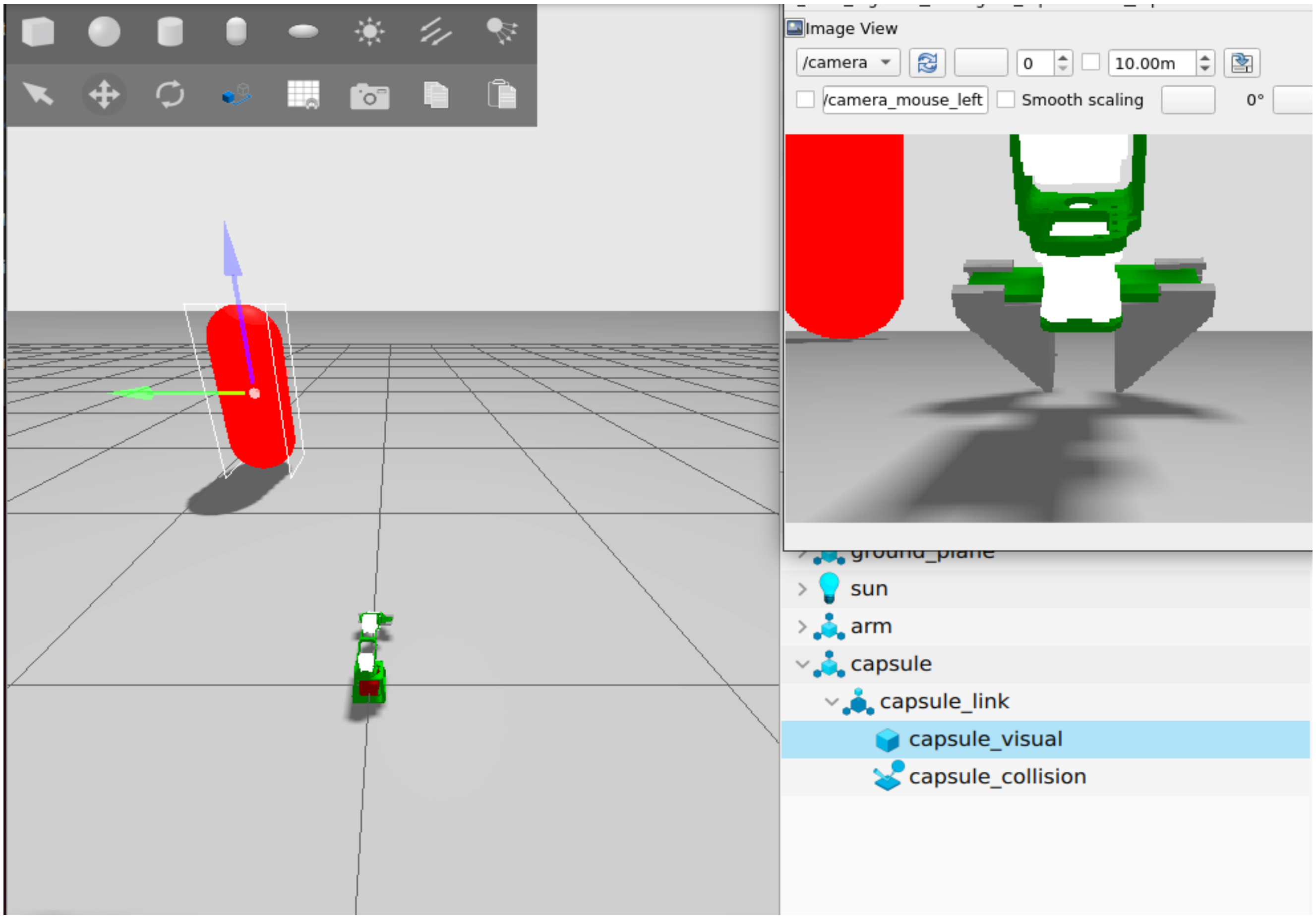

Figure 19: camera point of view

4 Adding a ROS publisher node

Inside the arm_control package, we created a ROS C++ node called arm_controller_node.cpp, which includes a publisher that writes commands to the /position_controller/commands topic via keyboard input, and a subscriber to the /joint_states topic that prints the current position of the joints:

```cpp
class Controller : public rclcpp::Node
{
  public:
    Controller()
    : Node("controller")
    {

      values_.resize(4);

      std::cout << "Enter " << "4" << " elements:\n";
        std::cin >> values_[0] >> values_[1] >> values_[2] >> values_[3];

      // Cambia il tipo del publisher
      publisher_ = this->create_publisher<std_msgs::msg::Float64MultiArray>("/position_controller/commands", 10);

      // Timer che chiama timer_callback ogni 500 ms
      timer_ = this->create_wall_timer(
      500ms, std::bind(&Controller::timer_callback, this));

      // Cambia il tipo della subscription a Float64MultiArray
      subscription_ = this->create_subscription<sensor_msgs::msg::JointState>(
      "/joint_states", 10, std::bind(&Controller::topic_callback, this, _1));

    }

  private:
    void timer_callback()
    {
      // Crea un oggetto Float64MultiArray
      auto message = std_msgs::msg::Float64MultiArray();
      // Popola l'array con dei valori (ad esempio, valori incrementali)
      for (size_t i = 0; i < 4; ++i) {
        message.data.push_back(static_cast<double>(values_[i])); // Aggiungi valori all'array
      }

      // Pubblica il messaggio
      publisher_->publish(message);
    }

    // Modifica del tipo di parametro nel callback
    void topic_callback(const sensor_msgs::msg::JointState & message) const
    {
      // Logga i valori ricevuti
      RCLCPP_INFO(this->get_logger(), "position: [%.2f, %.2f, %.2f, %.2f]", message.position[0], message.position[1], message.position[2], message.position[3]);

    }

   // Cambia il tipo della subscription
    rclcpp::Subscription<sensor_msgs::msg::JointState>::SharedPtr subscription_;
    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Publisher<std_msgs::msg::Float64MultiArray>::SharedPtr publisher_;
    std::vector<float> values_;
};
```

Figure 20: Controller node class

To ensure it was recognized, dependencies were added to the CMakeLists.txt file, specifically rlcpp, sensor_msgs, and std_msgs:

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

add_executable(controller src/arm_controller_node.cpp)
ament_target_dependencies(controller rclcpp std_msgs sensor_msgs)



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

Figure 21: dependencies in CMakeLists.txt