BACHELOR

Design and Implementation of a Digital Twin of a Pan/Tilt/Roll/Zoom Camera in Gazebo Simulator with ROS Interface

van Hemmen, Nienke

Award date:
2022

Department of Mechanical Engineering
Dynamics and Control Group

Design and Implementation of a Digital Twin of a Pan/Tilt/Roll/Zoom Camera in Gazebo Simulator with ROS Interface

Bachelor End Project

N. van Hemmen
1459570

Supervisor:
Dr. Ö. Arslan

Report number: DC 2022.055

Eindhoven, Thursday 23rd June, 2022

Table of Contents

List of Symbols

Table 1: List of Symbols

Symbol	***Description***	***Unit***	***Symbol Unit***
a	Length of the short side of a link	Meter	m
b	Length of the long side of a link	Meter	m
I	Mass moment of interia	Kilogram meter squared	$kg \cdot m^2$
m	Mass	Kilogram	kg

1. Introduction

1.1 Project description

Smart camera networks have been implemented more and more into several areas over the past couple of years. They gain more applications in, for example, automated environmental, security, and traffic observation [1]. Another utilization of such a camera network can be found in the Robotics Lab at the Technical University of Eindhoven (TU/e). Here football robots are created that compete against other teams. It is therefore very important that these robots are able to operate optimally and can easily move across the field. For this several Pan, Tilt, Roll, and Zoom (PTRZ) cameras, which can be seen in Figure 1.1, are used to communicate the live location of the robots. For these cameras, a Digital Twin has been built, based on one of the PTRZ cameras. This has been implemented in the Robot Operating System (ROS) interface and visualized in Gazebo to use the same environment as the robots along with other benefits which will be covered in the background section of this chapter. In order to do this, insight is needed about the workings of the interfaces using tutorials and plugins. The cameras are tested in both an inside as well as an outside world in order to not only use the cameras in the Robotics Lab but also for the other applications.

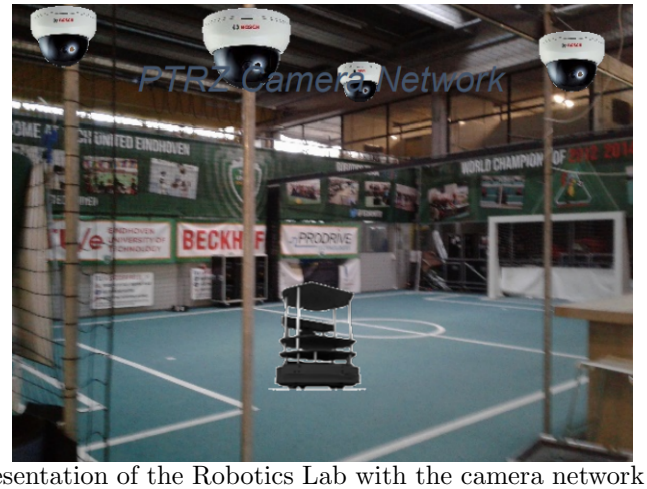

Figure 1.1: A representation of the Robotics Lab with the camera network and a football robot

In Chapter 1, the objective of this project is given together with a motivation from several sources. Additionally, some background information is given on the tools (ROS and Gazebo) that are used for the making of a Digital Twin. Chapter 2 will cover the design of the camera that was made and its functionality. Some coding is added to show how the camera is built and how it is made to move. Chapter 3 covers the control of the user of the camera. The two user interfaces, teleoperation, and scanning are elaborated upon. Chapter 4 discusses the limitations that were encountered and lists the assumptions that were made during the modeling of the PTRZ camera. Chapter 5 includes the conclusion which will contain a summary of the report and give further recommendations for future work on a PTRZ camera network simulator.

1.2 Objective

The goal of the project is to build several PTRZ cameras in a Gazebo environment. These cameras then have to be able to be controlled by a ROS interface. Additionally, the cameras should be tested in both an indoor as well as an outdoor simulation. This should all be present in a well-documented and validated ROS package in Python/C++.

1.3 Motivation

Designing, developing, and controlling the camera configurations can be very time-consuming and expensive. The existing camera networks, however, still do not cover as much area as is desired [2]. Consequently, a lot of adaptations still have to be made. Doing this on real-time cameras is, as mentioned, not profitable. Modeling the camera network in a computer simulation is therefore necessary.

To do this, a Digital Twin of a generic PTRZ camera can be created. This way new adaptations can first be tested on a simulator, before implementing it in the real-time cameras. Digital Twins are widely used to support design choices and to validate the properties used for the system. They merge the virtual and physical world in order to get a good overview of the entire system [3].

When a camera breaks in a simulator no harm is done and only some code needs to be changed. Besides that, tests in simulations can often be done much quicker by simply increasing the speed of the simulation. It is important that the code is first tested on the simulator rather than on the real cameras since the cameras can often contain expensive parts. Failures in these parts can be prevented by predictions made in the Digital Twin [4, 5]. So not only does the Digital Twin give a faster and safer output, but it also gives insight into the deficiencies that may occur in the components.
One such a simulator is the ROS interface. Together with the Gazebo simulator a 3D moving model can be made that represents the PTRZ camera.

1.4 Background

ROS is an open-source Robot Operating System and is widely used for many applications. Its main utilization is typically robotics. To attract a diverse set of users, the program is made to be multi-lingual. Programming languages such as C++, Python, and Octave can be used. In order to make the code easy to share with others, the program makes use of libraries in which the complex parts behind the coding are placed [6]. This also makes the user interface more comprehensible and easier.

Using ROS is beneficial since designs can be made and tested in an efficient way, minimizing costs, time, and energy [7]. Besides that, code can easily be reused and the user interface is made easier due to libraries [8].

Gazebo is tightly connected to ROS and visualizes the models created in 3D. The many physical properties present in Gazebo allow for an accurate representation of the real world [8]. Examples of physical properties are friction, gravity, and mass. To further increase the realistic environment, sensors can be added to, for example, create realistic feedback of the robot's movements. The robots are constructed of links and joints while the outside world consists of other created objects and landscapes. Gazebo is also an open-source platform and can therefore be widely used alongside ROS. ROS and Gazebo are connected through Gazebo plugins added in the ROS code [9].

For this project, it was chosen to work with these programs not only because of their many benefits but also due to the fact that the programs are already used by one of the main target groups. The Mechanical Engineering Robotics Lab has several TurtleBots, or football robots, driving around in their lab. These TurtleBots are modeled, simulated, and controlled using ROS and Gazebo. The camera network, also present in the Robotics Lab, is connected to these robots and helps them identify their position. It is therefore desirable to have both the moving robots as well as the cameras in the same program.

2. PTRZ Camera Simulation

2.1 Model Overview

In order to have a simulation model in Gazebo, a few steps need to be taken. First, the correct programs need to be installed. For this project, Ubuntu version 20.04 has been installed on the Oracle VM VirtualBox Manager. On Ubuntu, the latest version of ROS has been installed, which is ROS noetic, together with Gazebo11 [10, 11]. Then to get all of the systems working also some packages need to be installed [12, 13, 14]. Next, the structure of the folders with their allocated files can be built, a simplified version of this model can be seen in Figure 2.1.

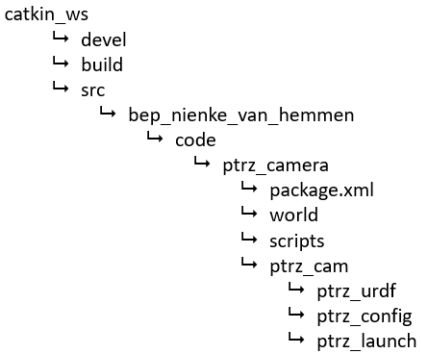

Figure 2.1: A short version of the order in which the files are placed in the repository

This always starts with a workspace, specifically the catkin workspace, which is a folder where you can modify, build, and install catkin packages. In this workspace a devel, build and src folder are present. Both the build as well as the devel folders are used to run or store the data present in the src folder [15]. The src, or source folder contains the code made for the packages and for this project also the GitLab repository. In the repository, the code folder is most important, here the package is located that contains all of the files regarding the PTRZ camera. A package is created to connect all of the different subfolders and keep them together. In the package.xml some additional lines need to be added in order to have certain systems working these can be found in section B.6. In order to use the visualization tool RVIZ, for example, which will be discussed later, the line $< run\_depend > rviz < /run\_depend >$ needs to be added. In the package three subfolders are present. One contains a world file that contains all of the information Gazebo needs to create a world for the camera to be placed into. The scripts folder contains two scripts with which the camera can be controlled. Lastly, the ptrz_cam folder contains the scripts that create the 3D model of the camera, the controller that ensures the camera moves, and a launch file. In the ptrz_urdf folder not only the two files that create the camera model are present but also the file which launches the RVIZ application. A full version of the structure of the folders, subfolders, and files can be seen in Appendix A.

2.2 Model Visualization

In the ptrz_cam file, the subfolder ptrz_urdf is present. This folder contains the most important files regarding the visualization of the PTRZ camera. To create the camera a general design first had to be made.

Figure 2.2: A general PTRZ camera has a casing with half a sphere connected to it, with the camera mechanism in the half sphere

The general PTRZ camera consists of half a sphere connected to a casing, as can be seen in Figure 2.2. This way the camera is free to move in its desired directions. The design of the PTRZ camera was therefore initially meant to look like this sphere connected to a box on top of its design. This would then create the same functions as the used cameras. The sphere would then be connected to the box as a joint and make all of the movements, except for the Zoom function, which would be assigned to the camera attached to it. A small camera box would be attached to the sphere in order to visualize the camera and its orientation. There were, however, some issues with this concept. Adding almost all of the functions to one joint made it hard to separate all of the movements. Both for visualization as well as for operation this meant that all of the movements needed to be separate.

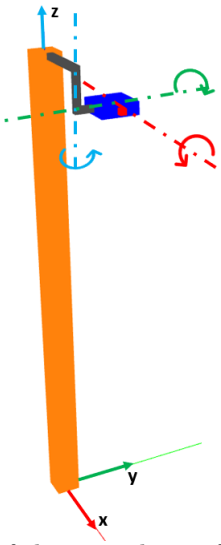

Figure 2.3: A schematic overview of the centerlines of the joints and the joint movement

The final camera design is illustrated in Figure 2.3. In the design, a big pole has been added to the construction in order to have the camera up in the air. Due to the fact that Gazebo makes use of gravity, the camera could not simply move in the air. Although gravity is a variable that can be turned off, it was decided not to do this. In Gazebo the gravity can be turned on and off for each object that is placed within the world. So the camera could be placed hanging in the air, or attached to a drone that enables it to move through the space. This would lead to additional movements that need to be controlled since the drone now also needs to be controlled. The cameras in the Robotics Lab are, however, not flying in the air, but are also attached to a construction that is connected with the ground. So to keep close to reality and to prevent even

further control, the gravity of the camera is left on. For a future design, the pole could be made thinner so that it becomes less noticeable. This way the pole blocks the view of the camera less.

Attached to the pole are some additional rigid links, which are colored gray in the figure. Not only do these rigid links connect the box with the camera to the pole, but they are also used to make some of the movements of the PTRZ camera. These movements will be further discussed in the following section. As mentioned, these rigid links connect the pole and a box. This box is used for the Tilt function and to carry the camera, which is visualized by a small red box.

The visualization of the camera was made using a URDF file [16]. This file contains the building blocks of the camera. The building blocks are called links. And all of the different links form the construction of the camera design. The individual links all have visual elements that define their shape together with collision elements. The collision elements can be seen as a protective shield around a visual element. Without the collision field, a block can move straight through the element, for example. The collision fields all use the same dimensions as their visual parts. In order to properly launch the links into Gazebo two things need to be done. The first is very important as it not only applies to the links but all of the elements present in the URDF file. In the URDF file, the Gazebo interface needs to be connected to the ROS code. This is done with the following piece of code:

```
<gazebo>
        <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
        <robotNamespace>/ptrz</robotNamespace>
        </plugin>
    </gazebo>
```

The definition of the namespace is also important here. The namespace links all of the subjects that are using this same namespace. Other files that use this namespace are the launch file and the controller, these will be discussed later in this section. The other important part that should be done in order to properly launch the links in Gazebo is that they need to have certain physical properties. Due to this the mass moment of inertia together with some minor friction constants have been added.

$$I = \frac{1}{12} m(a^2 + b^2) \tag{2.1}$$

Equation 2.1 shows the formula that was used to calculate the moment of inertia of all of the links [17]. With mass m, a the shortest side and b the longest. The pole was given a relatively high mass at first and the other components a low mass. This was done to prevent the pole from falling over. Fixing the pole to the world as a rigid joint, however, gave the same results, so this was later changed. The mass of the camera also needed to be drastically lower than the mass of the box because the box was otherwise unable to tilt upwards. To finalize the visualization of the links, colors were added. This way the links were easier to keep apart and it made it easier to visualize their movements.

2.3 Model Functionality

ROS coding makes use of different files that can ultimately all be combined. The URDF file, which has already been covered in the previous section, is one of them. In this section, the movements resulting from that same URDF file will be discussed.

The movements of the components are facilitated by joint functionalities.

```
<joint name="pan" type="continuous">
    <axis xyz="0 0 1"/>
    <origin xyz="${rl1len/2 - width/2} 0 -${rl2len/2 + width/2}" rpy="0 0 0"/>
    <parent link="rigid_link1"/>
    <child link="rigid_link2"/>
  </joint>
```

The snippet of code above shows how two of the rigid links are connected with a joint. It already shows a bit of the structure of the xacro file which will be discussed later in this chapter. The connection between the joints is done by addressing a parent link and a child link. In this case, the child link is the vertical rigid link used for the Pan movement. It moves relative to its parent, rigid link 1, which is connected to the top of the pole. A joint can either be stationary, using a fixed joint connection, or it can move. All of the moving joints used in this design are hinges, for this a revolute or a continuous type can be used. A continuous type was chosen since this does not need an additional upper and lower limit that is otherwise necessary to add with a revolute joint. This gave much smoother movements and it enabled the links to use the shortest route. With an upper and lower limit, the joint will limit its movements to these positions, not going beyond them. With no limitations, the joint can easily go back and forth without taking these numbers into account. The camera can therefore take the shortest route, which means that when the camera is at a lower position such as 1 and has to move to position 6, it will not move according to 1-2-3-4-5-6, but along the line 1-0-6, for example.

In order to properly connect the joints to Gazebo they need to be transmitted.

```
<transmission name="pan_trans">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="pan">
      <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface
    >
    </joint>
    <actuator name="pan_motor">
      <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface
    >
      <mechanicalReduction>1</mechanicalReduction>
    </actuator>
  </transmission>
```

The piece of code above shows how the joint for the Pan function is transmitted. The joints are all effort joints, and the value of 1 for the mechanical reduction is a generally used number. More information on these effort joints will be discussed later in this section.

As mentioned before, the movements of the different functionalities of the PTRZ camera are separated into different components in the design. The vertical rigid link, or rigid link 2, rotates around the Z-axis in the world and therefore makes the movements needed for the Pan function. It is connected to two other rigid links of which the bottom one has a joint connection with the box. This enables the box to make the nodding kind of movement that belongs to the Tilt function. Then in order to rotate the camera, the red box rotates along its X-axis. The Zoom function works different than the previously mentioned movements. Instead of moving continuously and as a joint, it uses the functionalities of the Gazebo camera plugin in ROS [18]. Some important parts of the plugin can be found in the code below:

```
<!-- camera -->
  <gazebo reference="camera">
    <sensor type="camera" name="camera1">
      <update_rate>70.0</update_rate>
      <camera name="head">
        <horizontal_fov>1.3962634</horizontal_fov>
        <image>
          <width>2000</width>
          <height>800</height>
          <format>R8G8B8</format>
        </image>
      </camera>
      <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
        <alwaysOn>true</alwaysOn>
        <updateRate>70.0</updateRate>
        <cameraName>ptrz/camera1</cameraName>
        <imageTopicName>image_raw</imageTopicName>
        <cameraInfoTopicName>camera_info</cameraInfoTopicName>
```

```
19          <frameName>camera</frameName>
20        </plugin>
21      </sensor>
22      </gazebo>
```

Where line 6 is the line that controls the field of view (fov). By making this value a variable, using the functionality of ROS xacro, the Zoom function can also be controlled. Enlarging this number gives that the camera will zoom out. Figure 2.4, Figure 2.5, and Figure 2.6 show how this change in value can be seen in the output of the camera. The camera plugin is one of the

Figure 2.4: Image created with half of the original fov

Figure 2.5: Image created with the original fov

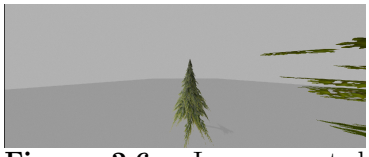

Figure 2.6: Image created with twice the original fov

sensor plugins that ROS and Gazebo use. Important is that the sensor refers to the link that it is connected to. In this case, this is the camera link that creates the small, red box in Gazebo. An increase in update rate can make the image created with the camera more smooth. The width of the image was increased to create a bigger and more rectangular image, which is generally the output of a camera. As mentioned earlier, the namespace is very important. In the section where the cameraName is defined, the correct namespace should be used. This ensures that the output of the camera is in the same directory as the controllers of the joints.

2.4 Model cleanup

When a lot of links and joints are used in an URDF file, it can become quite long. Often a lot of information is the same for multiple links. This is why a xacro file has been made. Xacro is a XML macro file that can be used not only to shorten XML files such as URDF files but also to define variables. The way in which the xacro file is structured is similar to the URDF file. The ROS-based xacro file is connected to Gazebo with the same piece of code that was used earlier, again making sure that the namespace is correctly defined. Then the different links and joints are defined, followed by potential sensor plugins and transmissions to transmit the joints to Gazebo. Two main things differ in the xacro file made from the original URDF code and a small change has been made to the physical properties. The first one is a list with all of the different variables. These are the widths and lengths of the several links and the value for the fov. The second one is that not all links are defined individually anymore. One piece of code represents the general setup of a link and in this code some variables are defined which can later be filled in. This piece of code is called a macro. A second macro is used for the transmission joints since these also have the same structure; only some small variables differ. The joints are still defined separately because they need very specific inputs on different spots. They do contain the variables defined at the beginning of the script as can be seen in Figure 2.3. A small change is also made to the values of the mass moment of inertia. Due to the low masses of some of the links, such as the camera, some inertia values came too close to zero. This can cause the camera model to collapse, which is not desirable. Therefore, the identity matrix of the mass moment of inertia is the best option to use for all of the links, since this matrix can always be used [19]. This also ensures that the physical properties can be included in the macro section of the links. A full version of the xacro file is in Appendix B.

2.5 Model Launch

The entire system is then put in Gazebo, this is done with a launch file in ROS. This launch file not only puts the camera in Gazebo, but it also starts the controllers, opens the visualization tool RVIZ, and places the camera into a Gazebo world. The launch file is a good place to put all of the made files together. To do this the files need to be loaded in the launch file using arguments and nodes. The arguments are used to locate the several files. For the xacro file, the correct directory in the URDF folder should be found. The node is then used to launch the file once it is found within the argument. The same should be done with the world file, RVIZ file, and the controller file. How to make these files will be discussed later. For the world file also an empty world can be launched. This creates a world in Gazebo with only a grid, a coordinate system, and a horizon. For the controller, it is important that the namespace is again referenced. The joint controllers should also be mentioned here. The URDF file does not need to be in the launch file since the xacro script is a replacement for this file. The code for the launch file can be found in Appendix B.

2.6 Model output

Besides an empty world in which only a horizon is present, a self-made world can also be launched. In order to visualize the movements of the camera better, a world has been created that places some items in the field of view of the camera. The world can be seen in Figure 2.7.

Figure 2.7: The outside world in Gazebo has a few elements placed in it in order to show the output of the camera

The items that are placed in the world are part of the Gazebo model collection. This is a repository which contains several items and worlds which can be modeled in Gazebo [20]. The world can then be saved as a .world file and launched by adding the world to the launch file using the following piece of code:

```
<include file="$(find gazebo_ros)/launch/empty_world.launch">
      <arg name="world_name" value="$(find ptrz_camera)/world/launch/camera.world"
    />
      <arg name="paused" value="false"/>
      <arg name="use_sim_time" value="true"/>
      <arg name="gui" value="true"/>
      <arg name="recording" value="false"/>
      <arg name="debug" value="false"/>
    </include>
```

The value should contain the correct world file that was made. It is best to save this in a separate world folder to keep everything organized.

In order to see how all of the different components mentioned are connected, a graph is made that connects all of the items of the camera. This graph can be generated by running the rqt_graph function in the terminal with *rosrun rqt_graph rqt_graph*.

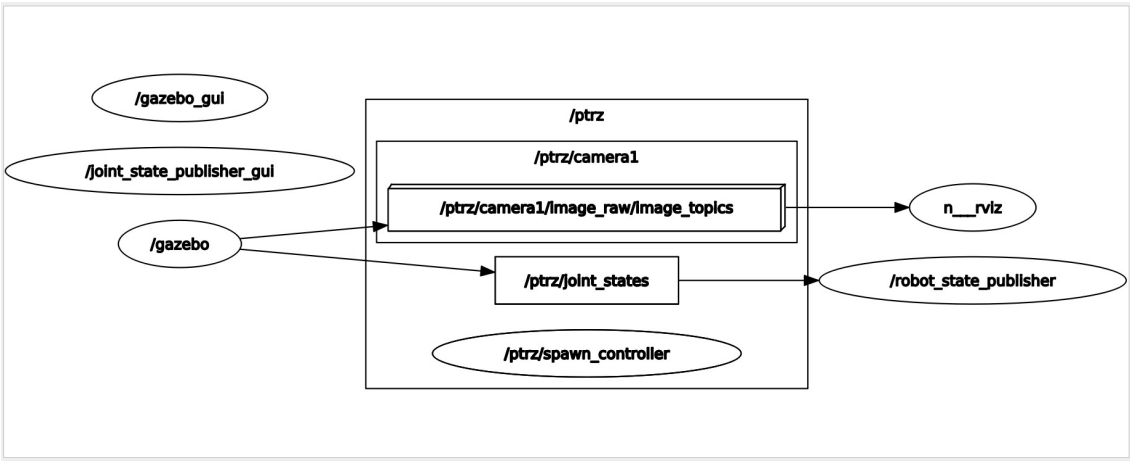

Figure 2.8: A simplification of the rqt_graph that shows the connection between the different programs, code, and links

Figure 2.8 shows a simplification of the graph, a more detailed graph can be seen in Figure C.1. Here it is visible that Gazebo is connected to the joint controllers and the camera. The camera is then connected to RVIZ in order to show an image. All of the components in the middle are connected because they are all under the same namespace - ptrz. Here it becomes visible why all of the different files needed the same namespace.

Finally, an image can be made. This image shows the output of the camera link. For this, the program RVIZ is used. It can easily be connected to the ROS files by adding a RVIZ file to the launch file. The RVIZ file can be made by opening RVIZ while running the Gazebo world. This can be done in a separate terminal using the command *rosrun rviz rviz*. The program will then open and several components can be altered. In order to get the output of the camera, its image should be added to the RVIZ desktop. This can be done by adding the camera1/image_raw topic that is visible in Figure 2.8. Now a new window is created that shows the live image generated by the camera. This should then be saved and put in a folder that is close to the URDF file, or it should be placed in the same folder as has been done for the RVIZ file for the PTRZ camera. Launching the launch file in the terminal will now automatically also open the desired RVIZ window so that the camera view can be seen immediately.

2.7 Launching a Second Camera

The objective of this project states that not only one camera should be created but several. A start has been made on this. When making several cameras, it is possible to only have one general URDF and controller file. The launch file should duplicate these and launch the camera several times. First, a normal launch file should be made that launched one camera, just like the launch file that was mentioned earlier in this chapter. This should search for the correct URDF or xacro file. The name and stating position are left variable, as can be seen in the piece of code below.

```
<arg name="robot_name"/>
<arg name="init_pose"/>
<node name="spawn_one_camera" pkg="gazebo_ros" type="spawn_model"
     args="$(arg init_pose) -param /robot_description -urdf -model $(arg robot_name
    )"
     respawn="false" output="screen" />
```

A different launch file should be made that launches several of these individual launch files, but with different names and starting positions. Here it is also important to have the distinct cameras under different namespaces. If all of the files were under the same namespace, the program would give an error stating that the same files are used several times. This is of course the case, but when using the files under distinct namespaces, the program sees them as different files and will not return an error. A third launch file is then used to launch the cameras in a Gazebo world, which can either be the empty world or a world that has been created. The code has been added to the repository as can be seen in Figure A.1.

3. PTRZ Camera Control via ROS

3.1 PID Camera Control

At this point, the joints are still not able to move. For this, a controller is needed. First, the joints are transmitted to become effort joints and to convert the movements of the joint to Gazebo. An effort controller is a controller that sends the desired force/torque (effort) to the interface.
The controller is made in a different file, a *.yaml* file. Here the different controllers are placed together with some additional parameters. One of the parameters is the publish rate. This value influences the update rate of the movements of the camera. At the moment this value is set to be 50 Hz and it can be increased in order to create even smoother movements. At the beginning of the controller file, the namespace is again defined so that the controllers of the various functions have the same namespace as their allocated joints have in the xacro file.
Another important factor is the PID controller. This regulates the movement of the links. The controllers of the joints are Joint Position Controllers. This means that the input given into the system will be a position. The Pan, Tilt, or Roll function will then move to the desired location. Without the PID controller, the camera can have an overshoot. This means that the camera will move a bit further than the desired position. To compensate for this it will again move backwards to reach its destination, again with an overshoot. This can repeat itself many times, causing the link to oscillate. The PID controller has been designed in such a way, that the camera will move almost instantly to its correct location, without any major oscillations. The PID controller has been designed using the rqt_reconfigure application.

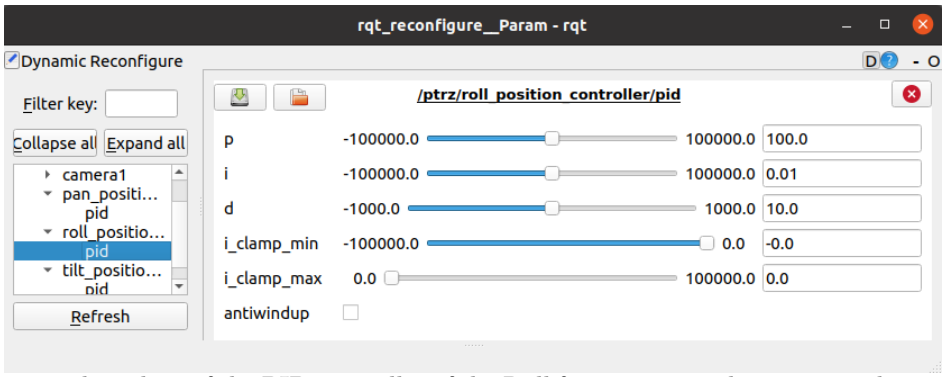

Figure 3.1: The values of the PID controller of the Roll function were chosen using the reconfigure application

Figure 3.1 shows the PID controller of the Roll function. The sliders can be used to alter the values of the PID controller while doing the simulation in Gazebo. Then the desired values can be put in the controller file and used to stabilize the camera's movements. The full controller file is put in the configuration folder of the camera and the full script can be found in Appendix B.

3.2 Teleoperation Control

The controller of the camera ensures that the camera can move to its desired position in a stable and purposeful way. The controller, however, needs an input for it to make the system move. Without any position input, the camera will remain stationary, according to its controller. Therefore some code has been made that allows the user to give the system input, also known as a publisher [21]. Through the keys of the keyboard, the user can make the camera move.

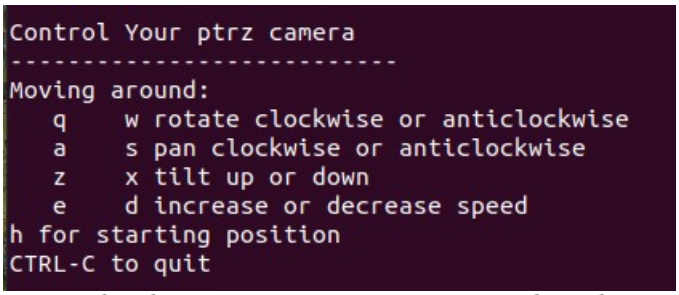

Figure 3.2: In the terminal a short instruction is given to introduce the use of the keyboard for teleoperation

Figure 3.2 shows the screen that the user gets when adding the following lines in a new terminal:

```
$ rosrun ptrz_camera keyboard_joint_control.py
```

The input is designed in such a way that when the keys q and w are used, the camera will Roll. q, for example, makes the camera rotate clockwise. a and s make the camera pan, where s makes it go in anticlockwise direction and a in the other direction. For the Tilt function, z and x are used. z makes the camera move up. Pressing h makes the camera move to its original beginning position or 'home' position. An update of where the camera is is published in the terminal. To prevent the terminal from filling up so that the piece of explanatory code, the one in Figure 3.2, is not visible anymore, the text is reprinted after 6 published positions.

The speed at which the camera is moving can also be controlled here. Pressing e causes the step size of the camera to increase, resulting in a faster-moving camera. d decreases this value so that the camera will slow down its movements and the view can be analyzed more in detail.

```
elif key== 'd' :
            DIFF = DIFF/STEP
            status = status + 1
            print(stripes)
            print(target_pos1, target_pos2, target_pos3)
            print(DIFF)
```

The piece of code above shows how the step size is decreased when pressing d. DIFF is the current step size and STEP is the amount with which the step size can be increased or decreased. The initial value for DIFF is 0.1 and for STEP 1.1. Dividing by the value of STEP lowers DIFF by about 0.01. For e the value of DIFF is multiplied with STEP, which in turn increases the value with 0.01. This small increase and decrease in step size enables the robot to keep its smooth movement while the values are changed. After each press of a key the position of the joints, the value of DIFF, and some stripes for readability are printed on the terminal. Target_pos1 is the position of the Roll joint, and pos2 and pos3 are the positions of respectively the Pan joint and the Tilt joint. The variable status keeps track of the number of keys that have been pressed. As has been mentioned earlier, when this value reaches 6, the text of the instruction, together with the text showing the position and step size will be reprinted.

With the use of the publisher and the controller, the movements of the camera can be illustrated in the Gazebo simulator. The only movement that is still missing is the Zoom function. For now, this value can only be changed in the xacro file. For future use, this can be changed to also be controlled live in the terminal.

The order in which the keys are pressed does not matter. In other words, when the order q-a-z is pressed, the camera will come to the same endpoint as when the order a-q-z is pressed. This is a very nice concept since the user does not need to think very carefully about their input sequence. The full script of the teleoperation control with the use of the keyboard can be seen in section B.4.

3.3 Scanning Control

It is not always desirable to manually change the position of the camera for the desired view. Often, the cameras make scanning motions. This is a motion where the camera moves back and forth covering certain areas. A different piece of code was made to activate this. The following input activates the scanning control script:

```
$ rosrun ptrz_camera scanner_joint_control.py
```

The code was first created for the Tilt function to see if it made the desired motion.

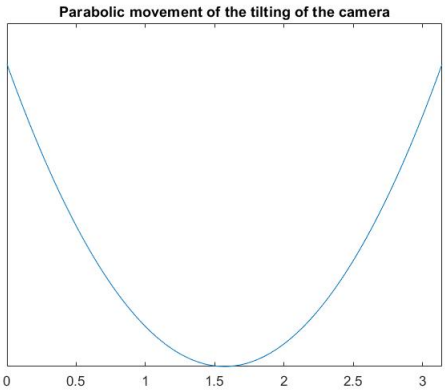

Figure 3.3: For the scanning motion the camera will go back and forth so its movement resembles a parabola

To create a scanning motion, the camera needs to go back and forth between two given values. For the created camera this was chosen to be between 0 and π as can be seen in Figure 3.3. This way the camera would not make an entire circle but rather focuses on the space on the ground. This is also the case for the cameras present in the Robotics Lab since the robots will be present on the ground plane and not somewhere on the ceiling. The starting position of the camera is at 0. It will then move to π, which is initially the value of the variable 'upper', with small steps to ensure that the output can be thoroughly analyzed. When the camera reaches the upper position π it will have a small overshoot, this is because the step size is not so small that it can land precisely on π. This small overshoot is essential because the upper value is then changed to 0. This will cause the camera to move back to its starting position again with the same step size. When 0 is reached the upper limit will change back to π again and the camera will head towards its new target.

```
    status = 0
    target_pos1 = 0
    target_pos2 = 0
    target_pos3 = 0
    upper = math.pi
    DIFF = 0.05
    print(target_pos1)
    print(target_pos2)
    print(target_pos3)
    while not rospy.is_shutdown():
      if target_pos1 < upper :
        target_pos1 += DIFF
        upper = math.pi
        status = status + 1
        time.sleep(0.05)
        print(target_pos1)
      elif target_pos1 >= upper :
        target_pos1 += -DIFF
```

```
19          upper = 0
20          status = status + 1
21          time.sleep(0.05)
22          print(target_pos1)
```

The code above shows the part of the code that controls the Pan function together with some initial values. The variable DIFF represents the step size and upper the constantly changing upper limit. A delay is added to each step of 0.05 seconds in order to slow the code down. Without this delay, all of the positions of the joints would be sent within a second. The values can be sent immediately because the condition of the while loop is always true as long as the terminal is not stopped. The positions of the different joints are again printed in the terminal just as with the teleoperation controller. This time no text needs to be reprinted after a couple of outputs, since there is no introduction text this time. The status of the loop is, nevertheless, scanned of so that errors can easily be tracked.

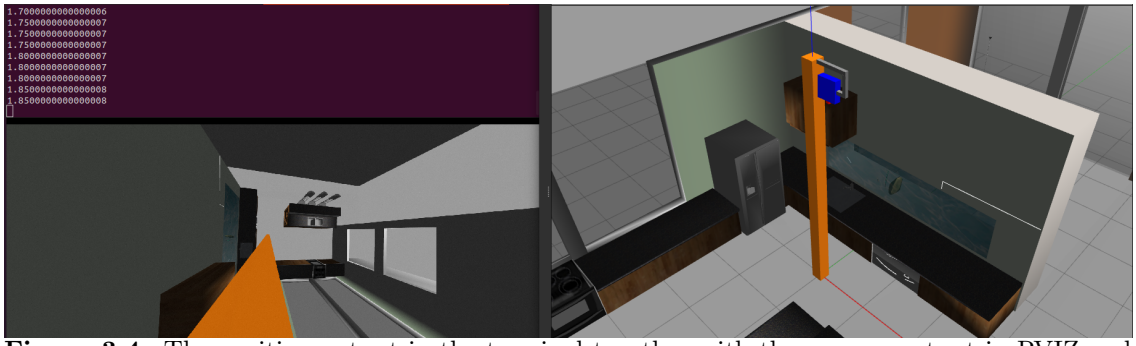

Figure 3.4: The position output in the terminal together with the camera output in RVIZ and the camera movement in the inside world in Gazebo next to each other

Figure 3.4 shows how the simulation of the scanner controller works. The joint positions of the joints are published in the terminal. The output of the camera can be seen in the image in RVIZ. And the moving 3D model is shown in the Gazebo model. The entire code can be found in Appendix B.

4. Limitations and Assumptions

4.1 Limitations

During the making of the camera in ROS and Gazebo, some limitations got in the way. One of them was the feedback system of the code. While ROS is an open-source platform that uses multiple languages, it still uses very specific ways of coding. A lot of tutorials and question forums help to learn a lot about the code and its language but this does not always cover the many different errors that can occur. When an error does occur, this is only visible once the code is launched. Prior to this, no errors can be tracked, this is different in other platforms such as MATLAB in which suggestions are already done while writing the code. When the error appears in the terminal, it is not always traceable where the problem comes from. Other times the exact file and line numbers are given together with the cause of the error. But overall identifying and solving errors was one of the main limitations.

The platform on which Gazebo and ROS were run was another limitation. The interfaces were run on Ubuntu on a Virtual Machine. These programs could often cause errors of their own or take a long time to run processes. This meant that changing and trying out even small alterations could take up several hours.

4.2 Assumptions

During the design of the PTRZ camera, certain assumptions had to be made. One of the bigger assumptions would be the dimensions of the camera. The camera that was created using a ROS URDF file was mostly made with the assumption that it had to capture the various movements of the camera. Less focus was therefore put on making the camera look like the cameras that are present at the Robotics Lab. This also means that all of the dimensions were chosen, not to resemble the used cameras, but to visualize the movements of the different joints. An example would be that the pole, used to keep the camera up, was chosen to be 4 meters high. This was done so that the camera would be high up in the air and would be able to capture a lot of items that could be placed in the Gazebo world.

Another assumption that was made, was closely connected to the dimensions of the part. First, all of the parts had the same mass. This was later changed because some of the parts would either fall over or would not be able to move because the links attached to it were heavier than they were. This led to a mass of $1 \cdot 10^{-}5kg$ for the small, red camera box. This negligible mass for the camera was needed since the blue box would otherwise not be able to tilt upwards.

For the user interfaces also some assumptions had to be made. The starting position, for example, was chosen to be the same starting position as the created camera in the URDF file. This was done to keep everything consistent. The starting position of a camera can, however, be different. Another assumption that was made was the angle for the scanning motion. It was assumed that the cameras at the Robotics Lab only need to scan the ground plane since this is where the football robots will be.

5. Conclusion

5.1 Summary

To conclude, due to the importance of PTRZ cameras and their many applications it is important that the cameras can be designed, developed, and controlled in an efficient and profitable way. For this, a Digital Twin was created that embodies the movements of the camera. This way the virtual and physical world are merged. The open-source interfaces of ROS and Gazebo were used as the platforms for this Digital Twin. The reason for this was because the target user group of the Robotics Lab at the TU/e was already using these programs. Besides that, ROS code is interchangeable, reusable, and widely used due to its many libraries and because it is multi-lingual.

For the camera design, the various functions were decoupled and each was placed on their own joint. This way the Pan, Tilt, and Roll movements were created in the model. The addition of a controller also made them move accordingly. The Zoom function was present in the value of the fov, which was made variable using xacro variables.

Then two user interfaces were created. One operates in a very interactive way with the user. The user can use certain keys on the keyboard to make the desired movements at the desired speed. The second one only needs to be launched and performs a scanning motion, going back and forth, in order to cover the ground area.

A Gazebo environment was created to visualize all of the movements. And RVIZ was used to show the output image of the camera.

5.2 Discussion

The goal of the project was to make and control several PTRZ cameras in a Gazebo environment, using the ROS interface. The cameras should then be tested in both an inside as well as an outside environment. Then a well-documented report should be given regarding the package made. For now, one PTRZ camera can be simulated and controlled in an inside environment and an outside environment with several items placed in strategic places so that the angles of the camera can be tested. The control of the camera can be done in two ways. One gives a lot of control to the user, giving space to test out the several joints and their orientation. The other makes the camera move more autonomously, giving the user room to analyze the output of the camera image. A start has been made on the addition of a second camera. For now, these cameras can only be placed in Gazebo but no control has been added yet.

The report is structured in such a way that the reader can make their own PRTZ camera and understands the different steps and choices that have been made. This leaves room for other design and control choices.

5.3 Future work

Future work can be done to further optimize the development of camera networks in inside and outside environments. The first step would be to have more cameras present in the simulation so that later these can all work together and to control all of these individual cameras. Another item would be to also add the Zoom function to the user interface. Now the value of the amount that the camera zooms in can only be changed in the code. This means that for a different setting of the Zoom function the value needs to be changed and after that, the scripts can be launched. If the value again needs to be changed the simulation needs to be stopped, the value should be changed, and the program should be launched again. Adding this to the user interface would be much more beneficial because the value can be changed during the simulation and the output can immediately be seen live. Then a code needs to be written that connects all of the cameras. This way the cameras can communicate with each other just like the camera networks that are already

widely used. The design of the camera can also be improved. The 3D model can look more like the real life camera models present in the Robotics Lab at the TU/e. Additionally, the pole can be made thinner to make it block the image less.

Bibliography

[1] Foresti GL Cavallaro SanMiguel JC Micheloni C, Shoop K. Self-reconfigurable smart camera networks. *Computer*, 47, 5 2014. 1

[2] A A Morye, C Ding, B Song, A Roy-Chowdhury, and J A Farrell. *Optimized Imaging and Target Tracking within a Distributed Camera Network*.* 2011. 2

[3] Stefan Boschert and Roland Rosen. Digital twin-the simulation aspect. *Mechatronic Futures: Challenges and Solutions for Mechatronic Systems and Their Designers*, pages 59–74, 1 2016. 2

[4] Digital twins for predictive maintenance. 2

[5] P. Aivaliotis, K. Georgoulias, and G. Chryssolouris. The use of digital twin for predictive maintenance in manufacturing. *https://doi.org/10.1080/0951192X.2019.1686173*, 32:1067–1080, 11 2019. 2

[6] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. 2

[7] View of rosplan: Planning in the robot operating system. 2

[8] Wei Qian, Zeyang Xia, Jing Xiong, Yangzhou Gan, Yangchao Guo, Shaokui Weng, Hao Deng, Ying Hu, and Jianwei Zhang. Manipulation task simulation using ros and gazebo. *2014 IEEE International Conference on Robotics and Biomimetics, IEEE ROBIO 2014*, pages 2594–2598, 4 2014. 2

[9] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 3:2149–2154, 2004. 2

[10] Ros/tutorials - ros wiki. 3

[11] Gazebo : Tutorial : Installing gazebo_ros_pkgs (ros 1). 3

[12] Ros/tutorials/understandingtopics - ros wiki. 3

[13] Ros/tutorials/navigatingthefilesystem - ros wiki. 3

[14] Robotic simulation scenarios with gazebo and ros. 3

[15] catkin/workspaces - ros wiki. 3

[16] urdf/tutorials - ros wiki. 5

[17] Rectangular plane mass moment of inertia calculator. 5

[18] Gazebo : Tutorial : Gazebo plugins in ros. 6

[19] urdf/tutorials/adding physical and collision properties to a urdf model. 7

[20] Github - osrf/gazebo_models: Gazebo database of sdf models. this is a predecessor to https://app.gazebosim.org. 8

[21] Ros/tutorials/writingpublishersubscriber(c++) - ros wiki. 11

A. Appendix A

The code has been structured over several documents. Its main folder is the catkin workspace in which several packages can run. The package made for this project has been added in a repository on GitLab under the folder bep_nienke_van_hemmen. Several files have been made, each in their own subfolder to keep the code organized and structured.

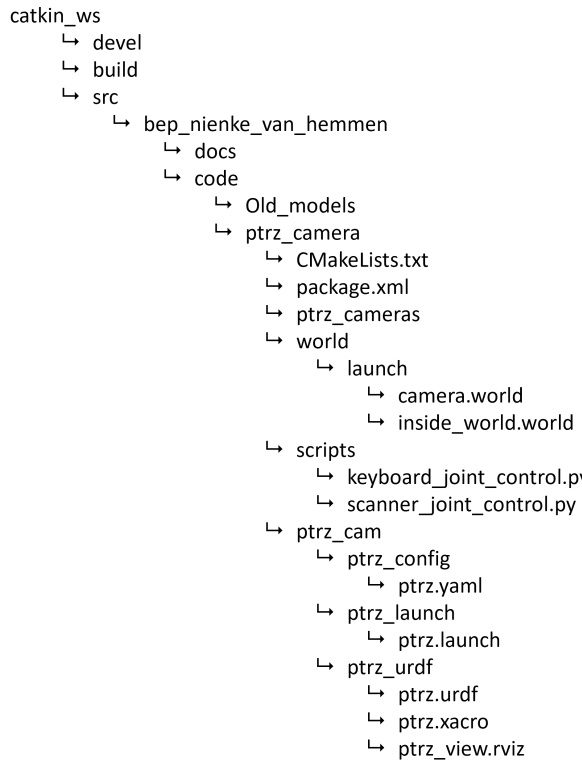

Figure A.1: A detailed overview of all of the files present in the GitLab repository

The final design of the PTRZ camera can be found in the ptrz.xacro file and can be launched by running the following line:

```
$ roslaunch ptrz_camera ptrz.launch
```

B. Appendix B

B.1 Xacro File

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

  <!--A Gazebo reference needs to be made so that the model can be launched into
    Gazebo-->
  <gazebo>
    <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
      <robotNamespace>/ptrz</robotNamespace>
    </plugin>
  </gazebo>


  <!--Defenition of the variables-->
  <xacro:property name="width"      value="0.05"/>
  <xacro:property name="width_pole" value="0.2"/>
  <xacro:property name="width_box"  value="0.1"/>
  <xacro:property name="polelen"    value="4"/>
  <xacro:property name="rl1len"     value="0.5"/>
  <xacro:property name="rl2len"     value="0.25"/>
  <xacro:property name="rl3len"     value="0.15"/>
  <xacro:property name="boxlen"     value="0.3"/>
  <xacro:property name="fov"        value="2.7925268"/>

  <!--Defenition of the links-->
  <xacro:macro name="link" params="name w l h mass color">
    <link name="${name}">
      <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
          <box size="${w} ${l} ${h}"/>
        </geometry>
      </visual>
      <collision>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
          <box size="${w} ${l} ${h}"/>
        </geometry>
      </collision>
      <inertial>
        <mass value="${mass}" />
        <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0" />
      </inertial>
    </link>
    <gazebo reference="${name}">
        <mu1>1.0</mu1>
        <mu2>1.0</mu2>
        <material>Gazebo/${color}</material>
    </gazebo>
  </xacro:macro>

  <!--Ground plane of the Gazebo world-->
  <link name="world"/>

  <!--Fixing the pole to the gorund so it remains stationary-->
  <joint name="base_joint" type="fixed">
    <origin xyz="0 0 ${polelen/2}" rpy="0.0 0.0 0.0"/>
    <parent link="world"/>
    <child link="pole"/>
  </joint>

  <!--Specifying the parameters of the pole link-->
  <xacro:link name="pole" w="${width_pole}" l="${width_pole}" h="${polelen}" mass="
    1" color="Orange"/>

  <joint name="rl1_joint" type="fixed">
```

```
    <origin xyz="${rl1len/2 + width_pole/2} 0 ${polelen/2 - width/2}" rpy="0 0 0"/>
    <parent link="pole"/>
    <child link="rigid_link1"/>
  </joint>

  <xacro:link name="rigid_link1" w="${rl1len}" l="${width}" h="${width}" mass="100"
     color="Grey"/>

  <!--Adding a continouos joint for the Pan function-->
  <joint name="pan" type="continuous">
    <axis xyz="0 0 1"/>
    <origin xyz="${rl1len/2 - width/2} 0 -${rl2len/2 + width/2}" rpy="0 0 0"/>
    <parent link="rigid_link1"/>
    <child link="rigid_link2"/>
  </joint>

  <xacro:link name="rigid_link2" w="${width}" l="${width}" h="${rl2len}" mass="100"
     color="Grey"/>

  <joint name="rl3_joint" type="fixed">
    <origin xyz="0 ${rl3len/2 - width/2} -${rl2len/2 + width/2}" rpy="0 0 0"/>
    <parent link="rigid_link2"/>
    <child link="rigid_link3"/>
  </joint>

  <xacro:link name="rigid_link3" w="${width}" l="${rl3len}" h="${width}" mass="100"
     color="Grey"/>

  <!--Adding a continouos joint for the Tilt function-->
  <joint name="tilt" type="continuous">
    <axis xyz="0 1 0"/>
    <origin xyz="0 ${rl3len/2 + boxlen/2} 0" rpy="0 0 0"/>
    <parent link="rigid_link3"/>
    <child link="box"/>
  </joint>

  <xacro:link name="box" w="${boxlen}" l="${boxlen}" h="${width_box}" mass="10"
    color="Blue"/>

  <!--Adding a continouos joint for the Roll function-->
  <joint name="roll" type="continuous">
    <axis xyz="1 0 0"/>
    <origin xyz="${width/2 + boxlen/2} 0 0" rpy="0 0 0"/>
    <parent link="box"/>
    <child link="camera"/>
  </joint>

  <xacro:link name="camera" w="${width}" l="${width}" h="${width}" mass="1e-5"
    color="Red"/>

  <!--Specifiyng the parameters for the camera sensor, with a variable for the fov
    for the Zoom function-->
  <gazebo reference="camera">
    <sensor type="camera" name="camera1">
      <update_rate>70.0</update_rate>
      <camera name="head">
        <horizontal_fov>${fov}</horizontal_fov>
        <image>
          <width>2000</width>
          <height>800</height>
          <format>R8G8B8</format>
        </image>
        <clip>
          <near>0.02</near>
          <far>300</far>
        </clip>
        <noise>
```

```
          <type>gaussian</type>
          <!-- Noise is sampled independently per pixel on each frame.
               That pixel's noise value is added to each of its color
               channels, which at that point lie in the range [0,1]. -->
          <mean>0.0</mean>
          <stddev>0.007</stddev>
        </noise>
      </camera>
      <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
        <alwaysOn>true</alwaysOn>
        <updateRate>70.0</updateRate>
        <cameraName>ptrz/camera1</cameraName>
        <imageTopicName>image_raw</imageTopicName>
        <cameraInfoTopicName>camera_info</cameraInfoTopicName>
        <frameName>camera</frameName>
        <hackBaseline>0.07</hackBaseline>
        <distortionK1>0.0</distortionK1>
        <distortionK2>0.0</distortionK2>
        <distortionK3>0.0</distortionK3>
        <distortionT1>0.0</distortionT1>
        <distortionT2>0.0</distortionT2>
      </plugin>
    </sensor>
    </gazebo>

  <!--Transmitting the joints in order to use them in Gazebo-->
  <xacro:macro name="transmission" params="prefix">
    <transmission name="${prefix}_trans">
      <type>transmission_interface/SimpleTransmission</type>
      <joint name="${prefix}">
        <hardwareInterface>hardware_interface/EffortJointInterface</
    hardwareInterface>
      </joint>
      <actuator name="${prefix}_motor">
        <hardwareInterface>hardware_interface/EffortJointInterface</
    hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
      </actuator>
    </transmission>
  </xacro:macro>

  <!--Specifying the distinct joints-->
  <xacro:transmission prefix="pan"/>
  <xacro:transmission prefix="tilt"/>
  <xacro:transmission prefix="roll"/>


</robot>
```

B.2 Launch File

```
<?xml version="1.0"?>
<launch>
    <!--The xacro file is loaded from the file it is located in-->
    <arg name="urdf_file" default="$(find xacro)/xacro --inorder '$(find
    ptrz_camera)/ptrz_cam/ptrz_urdf/ptrz.xacro'" />
    <param name="robot_description" command="$(arg urdf_file)" />
    <!--The rviz file is loaded-->
    <arg name="rvizconfig" default="$(find ptrz_camera)/ptrz_cam/ptrz_urdf/
    ptrz_view.rviz" />

    <!--Loading the world-->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
      <arg name="world_name" value="$(find ptrz_camera)/world/launch/camera.world"
    />
      <arg name="paused" value="false"/>
      <arg name="use_sim_time" value="true"/>
```

```
      <arg name="gui" value="true"/>
      <arg name="recording" value="false"/>
      <arg name="debug" value="false"/>
    </include>

    <!--The xacro file is spawned into Gazebo-->
    <node name="spawn_box" pkg="gazebo_ros" type="spawn_model" output="screen"
    args="-param /robot_description -urdf -model pan"/>

    <!--Loading the controller-->
    <rosparam file="$(find ptrz_camera)/ptrz_cam/ptrz_config/ptrz.yaml" command="
    load"/>

    <!--Defining the controllers that need to be spawned from the controller file
    -->
    <node name="spawn_controller" pkg="controller_manager" type="spawner" respawn="
    false" output="screen" ns="/ptrz" args="joint_state_controller
    roll_position_controller tilt_position_controller pan_position_controller"/>
    <node name="robot_state_publisher" pkg ="robot_state_publisher" type="
    robot_state_publisher" respawn="false" output="screen">
      <remap from="/joint_states" to="/ptrz/joint_states"/>
    </node>


    <node name="joint_state_publisher_gui" pkg="joint_state_publisher" type="
    joint_state_publisher"/>
    <node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)" />

</launch>
```

B.3 Controller

```
ptrz:
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50

  roll_position_controller:
    type: effort_controllers/JointPositionController
    joint: roll
    pid: {p: 100, i: 0.01, d: 10}

  tilt_position_controller:
    type: effort_controllers/JointPositionController
    joint: tilt
    pid: {p: 100.0, i: 0.01, d: 30}

  pan_position_controller:
    type: effort_controllers/JointPositionController
    joint: pan
    pid: {p: 100.0, i: 0.01, d: 30}
```

B.4 Key Teleoperation Control

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

import rospy
from std_msgs.msg import Float64
import sys, select, os
if os.name == 'nt':
  import msvcrt
else:
  import tty, termios


```

```
DIFF = 0.1
STEP = 1.1

msg = """
Control Your ptrz camera
---------------------------
Moving around:
   q    w roll clockwise or anticlockwise
   a    s pan clockwise or anticlockwise
   z    x tilt up or down
   e  d increase or decrease speed
h for starting position
CTRL-C to quit
"""

e = """
Communications Failed
"""
stripes = """
-----------
"""


def getKey():
    if os.name == 'nt':
      if sys.version_info[0] >= 3:
        return msvcrt.getch().decode()
      else:
        return msvcrt.getch()

    tty.setraw(sys.stdin.fileno())
    rlist, _, _ = select.select([sys.stdin], [], [], 0.1)
    if rlist:
        key = sys.stdin.read(1)
    else:
        key = ''

    termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)
    return key

if __name__=="__main__":
    if os.name != 'nt':
        settings = termios.tcgetattr(sys.stdin)

    rospy.init_node('ptrz_key_teleop')
    pos1pub = rospy.Publisher('ptrz/roll_position_controller/command', Float64,
    queue_size=10)
    pos2pub = rospy.Publisher('ptrz/pan_position_controller/command', Float64,
    queue_size=10)
    pos3pub = rospy.Publisher('ptrz/tilt_position_controller/command', Float64,
    queue_size=10)

    status = 0
    target_pos1 = 0
    target_pos2 = 0
    target_pos3 = 0


    print(msg)
    while(1):
        key = getKey()
        if key == 'q' :
            target_pos1 += -DIFF
            status = status + 1
            print(stripes)
            print(target_pos1, target_pos2, target_pos3)
            print(DIFF)
```

```
        elif key == 'w' :
            target_pos1 += DIFF
            status = status + 1
            print(stripes)
            print(target_pos1, target_pos2, target_pos3)
            print(DIFF)
        elif key == 'a' :
            target_pos2 += -DIFF
            status = status + 1
            print(stripes)
            print(target_pos1, target_pos2, target_pos3)
            print(DIFF)
        elif key == 's' :
            target_pos2 += DIFF
            status = status + 1
            print(stripes)
            print(target_pos1, target_pos2, target_pos3)
            print(DIFF)
        elif key == 'z':
            target_pos3 += -DIFF
            status = status + 1
            print(stripes)
            print(target_pos1, target_pos2, target_pos3)
            print(DIFF)
        elif key == 'x':
            target_pos3 += DIFF
            status = status + 1
            print(stripes)
            print(target_pos1, target_pos2, target_pos3)
            print(DIFF)
        elif key== 'h' :
            target_pos1 = 0
            target_pos2 = 0
            target_pos3 = 0
            status = 0
            DIFF = 0.1
            print(stripes)
            print(target_pos1, target_pos2, target_pos3)
            print(DIFF)
        elif key== 'e' :
            DIFF = DIFF*STEP
            status = status + 1
            print(stripes)
            print(target_pos1, target_pos2, target_pos3)
            print(DIFF)
        elif key== 'd' :
            DIFF = DIFF/STEP
            status = status + 1
            print(stripes)
            print(target_pos1, target_pos2, target_pos3)
            print(DIFF)
        else:
            if (key == '\x03'):
                break

        if status == 6 :
            print(msg)
            status = 0

        pos1pub.publish(target_pos1)
        pos2pub.publish(target_pos2)
        pos3pub.publish(target_pos3)

    if os.name != 'nt':
        termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)
```

B.5 Scanner Control

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

import rospy
from std_msgs.msg import Float64
import sys, select, os, math, time
PI = 3.1415926535897
if os.name == 'nt':
  import msvcrt
else:
  import tty, termios

stripes = """
-----------
"""


if __name__=="__main__":
    if os.name != 'nt':
        settings = termios.tcgetattr(sys.stdin)

    rospy.init_node('ptrz_scanner')
    pos1pub = rospy.Publisher('ptrz/roll_position_controller/command', Float64,
    queue_size=10)
    pos2pub = rospy.Publisher('ptrz/pan_position_controller/command', Float64,
    queue_size=10)
    pos3pub = rospy.Publisher('ptrz/tilt_position_controller/command', Float64,
    queue_size=10)

    status = 0
    target_pos1 = 0
    target_pos2 = 0
    target_pos3 = 0
    upper = math.pi
    DIFF = 0.05
    print(target_pos1)
    print(target_pos2)
    print(target_pos3)
    while not rospy.is_shutdown():
      if target_pos1 < upper :
        target_pos1 += DIFF
        upper = math.pi
        status = status + 1
        time.sleep(0.05)
        print(target_pos1)
      elif target_pos1 >= upper :
        target_pos1 += -DIFF
        upper = 0
        status = status + 1
        time.sleep(0.05)
        print(target_pos1)
      if target_pos2 < upper :
        target_pos2 += DIFF
        upper = math.pi
        status = status + 1
        time.sleep(0.05)
        print(target_pos2)
      elif target_pos2 >= upper :
        target_pos2 += -DIFF
        upper = 0
        status = status + 1
        time.sleep(0.05)
        print(target_pos2)
      if target_pos3 < upper :
        target_pos3 += DIFF
        upper = math.pi
        status = status + 1
```

```
        time.sleep(0.05)
        print(target_pos3)
      elif target_pos3 >= upper :
        target_pos3 += -DIFF
        upper = 0
        status = status + 1
        time.sleep(0.05)
        print(target_pos3)


      pos1pub.publish(target_pos1)
      pos2pub.publish(target_pos2)
      pos3pub.publish(target_pos3)




if os.name != 'nt':
        termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)
```

B.6 Package XML

```
<?xml version="1.0"?>
<package>
  <name>ptrz_camera</name>
  <version>0.0.0</version>
  <description>The ptrz_camera package</description>

  <maintainer email="n.v.hemmen@student.tue.nl">nienke</maintainer>

  <license>BSD</license>

  <!-- The *depend tags are used to specify dependencies -->
  <buildtool_depend>catkin</buildtool_depend>
  <run_depend>controller_manager</run_depend>
  <run_depend>diff_drive_controller</run_depend>
  <run_depend>gazebo_ros</run_depend>
  <run_depend>gazebo_ros_control</run_depend>
  <run_depend>joint_state_controller</run_depend>
  <run_depend>joint_state_publisher_gui</run_depend>
  <run_depend>position_controllers</run_depend>
  <run_depend>robot_state_publisher</run_depend>
  <run_depend>rqt_robot_steering</run_depend>
  <run_depend>rviz</run_depend>
  <run_depend>urdf_tutorial</run_depend>
  <run_depend>xacro</run_depend>


  <export>
  </export>
</package>
```

C. Appendix C

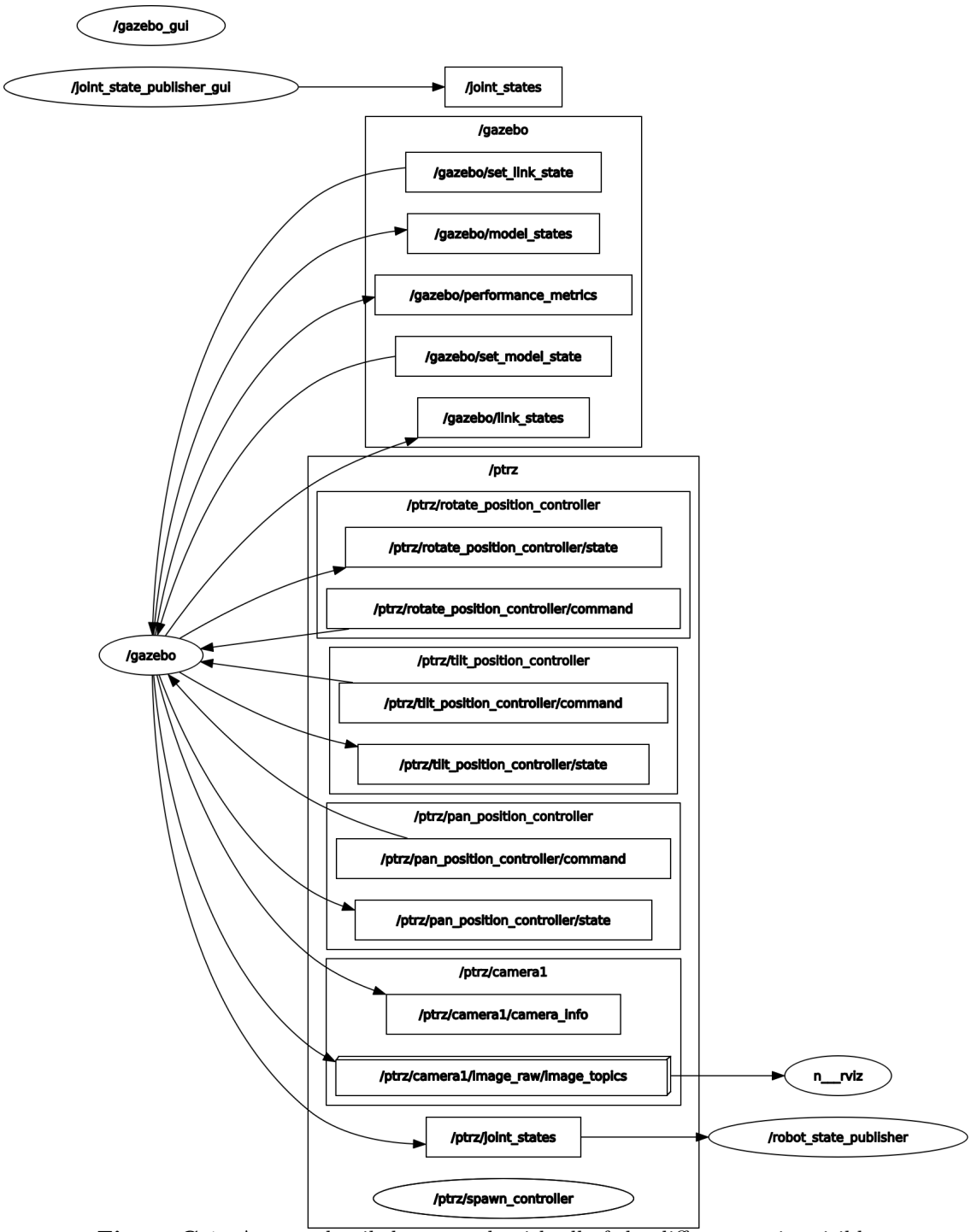

Figure C.1: A more detailed rqt_graph with all of the different topics visible