Lecture 3 (14/9) - Learning ROS Transforms (TF), Robot Visualization (RVIZ) and Simulation (Gazebo)

Starts 14 September, 2020 1:00 PM

Learning ROS Transforms (TF), Robot Visualization (RVIZ) and Simulation (Gazebo)

We meet in Zoom at 13:00. Link: https://dtudk.zoom.us/j/63569471741?pwd=emNGMEQxNitXdGFhc3hSZXhqdVFNQT09

Reading Material

1. You can learn more about the tf protocol in this: paper.
2. For tf we will more or less follow these: tutorials
3. **Mastering ROS for Robotics Programming (link)**
Joseph, Lentin
Chapters 2,3, First half Chapter 3
4. **Programming Robots with ROS: A Practical Introduction to the Robot Operating System (link)**
Quigley, Morgan · Gerkey, Brian · Smart, William D.
Chapter 17

Time Plan

- 13:00 Introduction to Transformations for Robot Operation, to ROS robot modeling and to ROS simulation
- 15:00 exercises with support on Discord (use link: https://discord.gg/AEsFTp9)

Description

In this lecture we will learn about:

- The axis transforms in ROS using the tf protocol,
- The robot visualization tool of ROS (RVIZ)

- The Universal Robot Description Format (URDF)
- The simulator associated with ROS (Gazebo)

0 % 0 of 1 topics complete

31391-SFfAS-3-Learning_ROS_Transforms-seperate
PDF document

Exercise 3.1 - Broadcast a TF of our turtle

Exercise 3.1 - Broadcast a transformation (tf) of our turtle

NOTE:

We define notes with a red vertical line,
terminal commands with a black one,
and code with a blue one

We would like to broadcast a tf of our robot. To do so create the following file turtle_tf_broadcaster.py and save it to **~/catkin_ws/src/hello_ros/scripts/turtle_tf_broadcaster.py:**

```
#!/usr/bin/env python
import rospy

import tf
import turtlesim.msg

def handle_turtle_pose (msg, turtlename):
    br = tf.TransformBroadcaster()
    # send out the pose of the turtle in the form of a transform
    br.sendTransform((msg.x, msg.y, 0),
                     tf.transformations.quaternion_from_euler(0 , 0, msg.theta),
                     rospy.Time.now(),
                     turtlename,
                     "world")
```

```
if __name__ == '__main__':
    rospy.init_node( 'turtle_tf_broadcaster' )
    turtlename = rospy.get_param('~turtle', default='turtle1')
    # subscribe to the pose of the turtle
    rospy.Subscriber( '/%s/pose' %  turtlename,
                      turtlesim.msg.Pose,
                      handle_turtle_pose,
                      turtlename)
    rospy.spin()
```

Note: Remember to make the python file executable like in previous exercises.

The exercise here is to run the newly created turtle_tf_broadcaster.py together with the turtle simulator (from previous in the course) and watch the printout transformation change as you move the turtle with the arrow keys.

Exercise 3.2 - Add fixed or moving frames

Exercise 3.2 - Add fixed or moving frames

Fixed frame

For this exercise we would like to create a new fixed frame with respect to some other frame (e.g. the turtle pose). To do so create the following file fixed_tf_broadcaster.py and save it to **~/catkin_ws/src/hello_ros/scripts/fixed_tf_broadcaster.py:**

```
#!/usr/bin/env python
import rospy
import tf
```

```
if __name__ == '__main__':
    rospy.init_node('fixed_tf_broadcaster')
    br = tf.TransformBroadcaster()
    rate = rospy.Rate(10.0)
    while not rospy.is_shutdown():
        br.sendTransform((2.0, 0.0, 0.0),
                         (0.0, 0.0, 0.0, 1.0),
                         rospy.Time.now(),
                         "carrot1",
                         "turtle1")
        rate.sleep()
```

The new frame is called carrot1 and is fixed with reference to the turtle1 frame. It is placed such that it is always 2 meters from the turtle1 frame.

Note: Remember to make the python file executable like in previous exercises.

Now we can also run the thing:

```
rosrun hello_ros fixed_tf_broadcaster.py
```

Notice that the program runs as before with no visual changes. In order to test if the fixed frame is broadcasted we can write the following in a new terminal window:

```
rosrun tf tf_echo /world /carrot1
```

This will print the TF from the world to the carrot frame. Notice it changes when you move the turtle around. We can also print the TF from the turtle1 frame to the carrot frame:

```
rosrun tf tf_echo /turtle1 /carrot1
```

Notice that the TF from turtle1 to carrot doesn't change as the carrot frame is fixed to the turtle1 frame.

Visualization of Tfs (and much more)

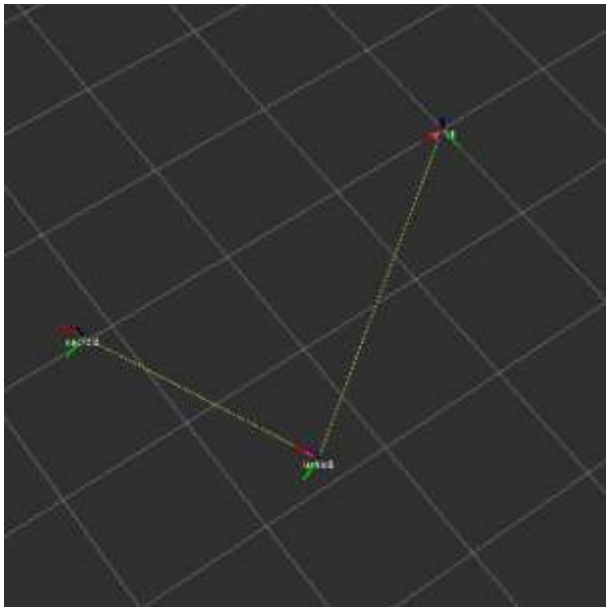
RViz is a tool to visualize many of the messages that get send in ROS. One of the things it can visualize are TFs.

```
rosrun rviz rviz
```

Topics are added in the left panel by pressing the *Add* button. Scroll down and find TF. As any in simulator you need to have a fixed position in which your entire simulation is related to. This is called *Fixed Frame* and can be found under the Global options in the left panel. **Hint:** In Exercise 3.1 we made an

assumption about what this frame is called when making the transformation to the turtle.

If you have everything setup correctly, you should have three frames in your view, one of which moves when you move the turtle and another always at a fixed distance from your turtle.



Moving frame

ROS makes it easy for us to get the current time with *rospy.Time.now().to_sec().*

Create a new file called moving_tf_broadcaster.py and use *rospy.Time.now().to_sec()* and change the above to make the carrot go in circles around the turtle.

Exercise 3.3 - Listen to TF of our turtle

Exercise 3.3 - Listen to TF of our turtle

In a previous exercise we made a script that broadcasts the TF of the turtle. For this exercise we would like to make a script that listens to the broadcasted TF such that we can make second turtle follow our original turtle.

To do so create the following file turtle_tf_listener.py and save it to **~/catkin_ws/src/hello_ros/scripts/turtle_tf_listener.py:**

```
#!/usr/bin/env python
import rospy
import math
import tf
import geometry_msgs.msg
import turtlesim.srv

if __name__ == '__main__':
    rospy.init_node('turtle_tf_listener')

    listener = tf.TransformListener()

    rospy.wait_for_service('spawn')
    spawner = rospy.ServiceProxy('spawn', turtlesim.srv.Spawn)
    spawner(4, 2, 0, 'turtle2')


turtle_vel = rospy.Publisher('turtle2/cmd_vel', geometry_msgs.msg.Twist,que

    rate = rospy.Rate(10.0)
    while not rospy.is_shutdown():
        try:
            (trans,rot) = listener.lookupTransform('/turtle2', '/turtle1', rospy.Time(0
        except (tf.LookupException, tf.ConnectivityException, tf.ExtrapolationE
            continue

        angular = 4 * math.atan2(trans[1], trans[0])
        linear = 0.5 * math.sqrt(trans[0] ** 2 + trans[1] ** 2)
        cmd = geometry_msgs.msg.Twist()
        cmd.linear.x = linear
        cmd.angular.z = angular
        turtle_vel.publish(cmd)

        rate.sleep()
```

In this script we get the TF from turtle1 to turtle2. We then use this transformation to make turtle2 move towards turtle1.

Note: Remember to make the python file executable like in previous exercises.

The exercise here is to run the turtle_tf_broadcaster for turtle2 such that the position of turtle2 is a transformation.

Exercise 3.4 - Time traveling with TF

Exercise 3.4 - Time traveling with TF

We would like to implement some time delay to our system...

Lets create a file called turtle_tf_listener_time_travel.py and save it to **~/catkin_ws/src/hello_ros/scripts/turtle_tf_listener_time_travel.py:**

```
#!/usr/bin/env python
import rospy
import math
import tf
import geometry_msgs.msg
import turtlesim.srv

if __name__ == '__main__':
    rospy.init_node('turtle_tf_listener')

    listener = tf.TransformListener()

    rospy.wait_for_service('spawn')
    spawner = rospy.ServiceProxy('spawn', turtlesim.srv.Spawn)
    spawner(4, 2, 0, 'turtle2')

    turtle_vel = rospy.Publisher('turtle2/cmd_vel',
geometry_msgs.msg.Twist,queue_size=1)
```

```
rate = rospy.Rate(10.0)
while not rospy.is_shutdown():
    try:
        now = rospy.Time.now()
        past = now - rospy.Duration(5.0)
        listener.waitForTransformFull("/turtle2", now,
                                      "/turtle1", past,
                                      "/world", rospy.Duration(1.0))
        (trans, rot) = listener.lookupTransformFull("/turtle2", now,
                                      "/turtle1", past,
                                      "/world")
    except (tf.Exception, tf.LookupException, tf.ConnectivityException):
        continue

    angular = 4 * math.atan2(trans[1], trans[0])
    linear = 0.5 * math.sqrt(trans[0] ** 2 + trans[1] ** 2)
    cmd = geometry_msgs.msg.Twist()
    cmd.linear.x = linear
    cmd.angular.z = angular
    turtle_vel.publish(cmd)

    rate.sleep()
```

This time we first use the function waitfortransformfull() to wait for the given transformation to become available. Afterwards we use the function *lookuptransformfull* to actually get the transformation. Notice that we use to different times with five seconds between them.

Note: Remember to make the python file executable like in previous exercises.

Try to change the delay such that turtle2 is only 2 seconds behind turtle1.

Exercise 3.6 - URDF first steps

Exercise 3.6 - URDF first steps

We will create our robot from scratch..

Lets create a file named **hello_ros_robot_1**.urdf and save it to **~/catkin_ws/src/hello_ros/urdf/hello_ros_robot_1.urdf:**

```
<?xml version="1.0"?>
<robot name="hello_ros_robot">
  <link name="world"/>
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.05" radius="0.1"/>
      </geometry>
      <material name="silver">
        <color rgba="0.75 0.75 0.75 1"/>
      </material>
      <origin rpy="0 0 0" xyz="0 0 0.025"/>
    </visual>
  </link>
  <joint name="fixed" type="fixed">
    <parent link="world"/>
    <child link="base_link"/>
  </joint>
</robot>
```

Lets make sure that it has no errors:

```
roscd hello_ros/urdf/
check_urdf hello_ros_robot_1.urdf
```

In order to load the model of our robot we need to publish the robot description.

Let's create the ros launch file: **urdf_launch_1.launch** to run this. The file simply loads the parameters from the urdf file to the paramter server:

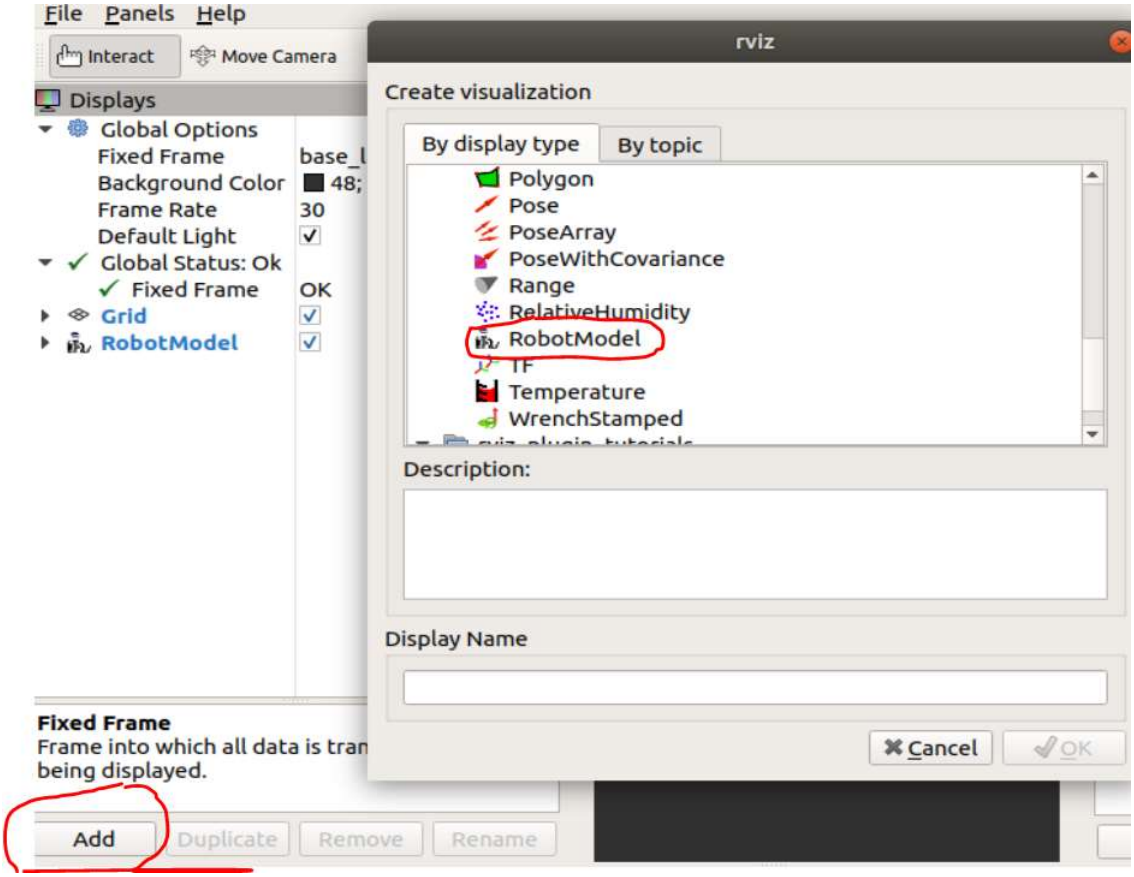
```
<launch>
  <!-- Load the hello_ros_robot URDF model into the parameter server -->
    <param name="robot_description" textfile="$(find
hello_ros)/urdf/hello_ros_robot_1.urdf" />

</launch>
```

Run the launch file as you have done in all the previous exercises.

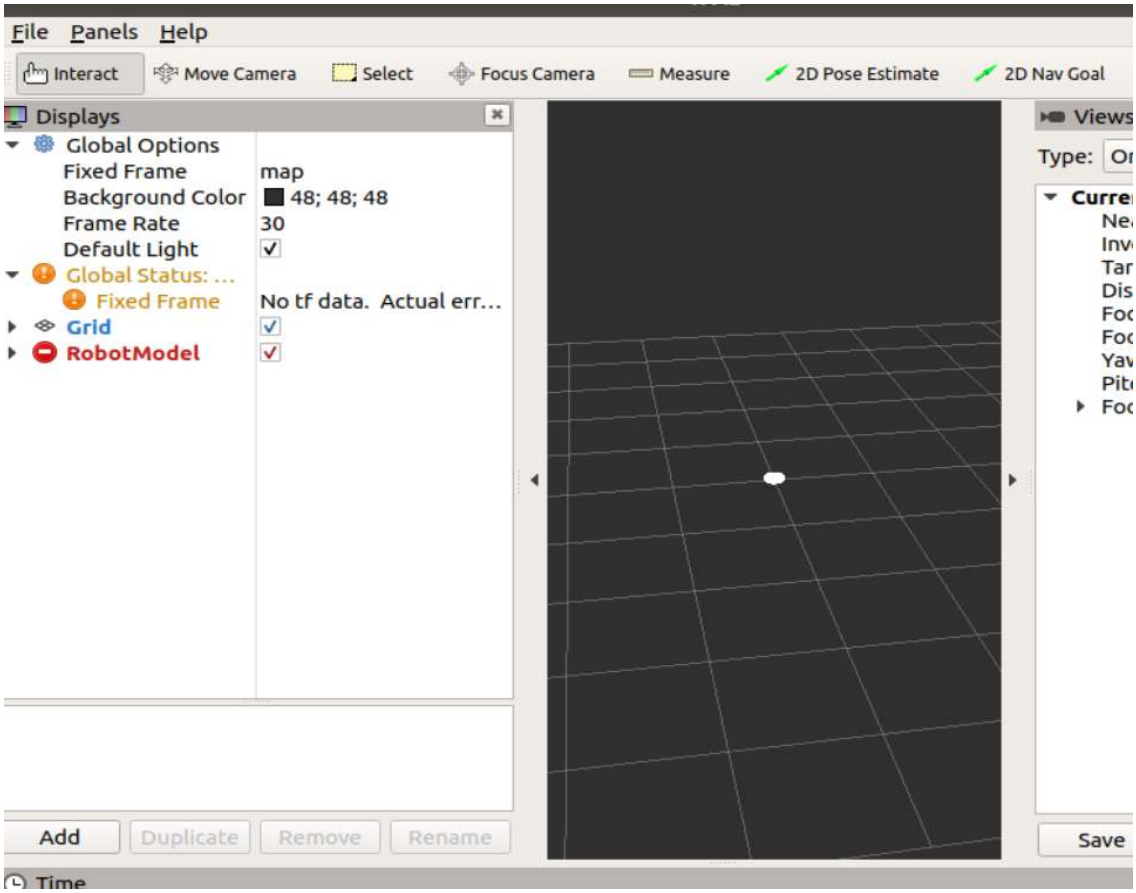
Lets run the program **RVIZ:**

```
rosrun rviz rviz
```

This opens a new window. In there, press the button "add" and pick "robotmodel":



Now our robot is shown:



It doesn't look like much yet as we only have a base link, but we'll add more to it in the following exercises.

As an exercise, have a look at the URDF file and see if you can identify which kind of geometric object we have added. See if you can change the size and color of it.

Exercise 3.7 - Now this starts being a robot

Exercise 3.7 - Now this starts being a robot

We are still creating our robot from scratch..

Let's add another part to our robot by creating a file named **hello_ros_robot_2**.urdf and save it to **~/catkin_ws/src/hello_ros/urdf/hello_ros_robot_2.urdf:**

```
<?xml version="1.0"?>
<robot name="hello_ros_robot">
  <link name="world"/>
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.05" radius="0.1"/>
      </geometry>
      <material name="silver">
        <color rgba="0.75 0.75 0.75 1"/>
      </material>
      <origin rpy="0 0 0" xyz="0 0 0.025"/>
    </visual>
  </link>
  <joint name="fixed" type="fixed">
    <parent link="world"/>
    <child link="base_link"/>
  </joint>
  <link name="torso">
    <visual>
      <geometry>
        <cylinder length="0.5" radius="0.05"/>
      </geometry>
      <material name="silver">
        <color rgba="0.75 0.75 0.75 1"/>
      </material>
      <origin rpy="0 0 0" xyz="0 0 0.25"/>
    </visual>
  </link>
  <joint name="hip" type="continuous">
    <axis xyz="0 0 1"/>
    <parent link="base_link"/>
    <child link="torso"/>
    <origin rpy="0 0 0" xyz="0.0 0.0 0.05"/>
  </joint>
</robot>
```

Let's make sure that it has no errors:

```
roscd hello_ros/urdf/
check_urdf hello_ros_robot_2.urdf
```

In order to load the model of our robot we need to use the:

joint_state_publisher and robot_state_publisher

Let's create the ros launch file: **urdf_launch_2.launch** to run this:

```
<launch>
  <!-- Load the hello_ros_robot URDF model into the parameter server -->
    <param name="robot_description" textfile="$(find
hello_ros)/urdf/hello_ros_robot_2.urdf" />
  <param name="use_gui" value="true" />
  <node name="joint_state_publisher" pkg="joint_state_publisher"
        type="joint_state_publisher"/>
  <node name="robot_state_publisher" pkg="robot_state_publisher"
        type="robot_state_publisher"/>
</launch>
```

When you run the launch file a new window with a slide will open. We'll use this to control our robot. In case you get an error about joint_state_publisher_gui not being installed, you can install it using:

```
sudo apt install ros-melodic-joint-state-publisher-gui
```

Lets see what tf saw. The following command creates a PDF graph of our current transform tree:

```
rosrun tf view_frames
```

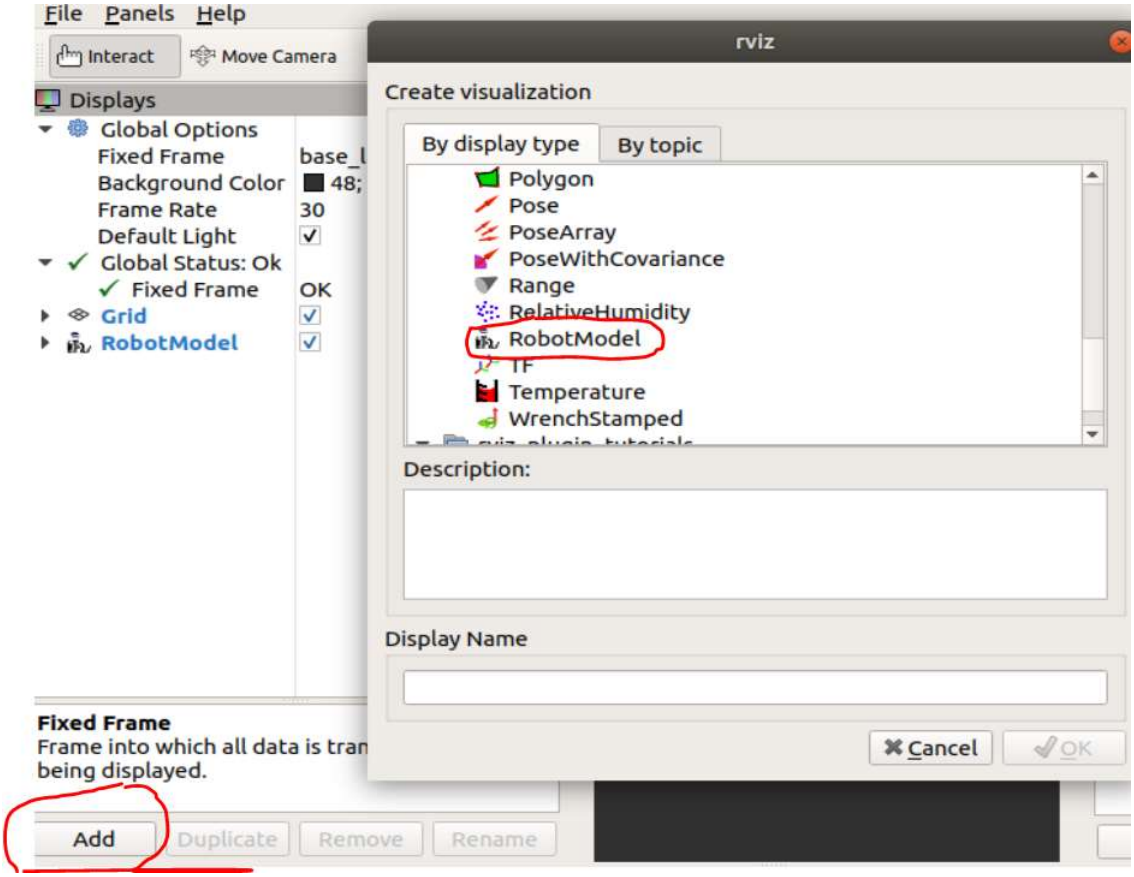
To view the pdf, type:

```
evince frames.pdf
```

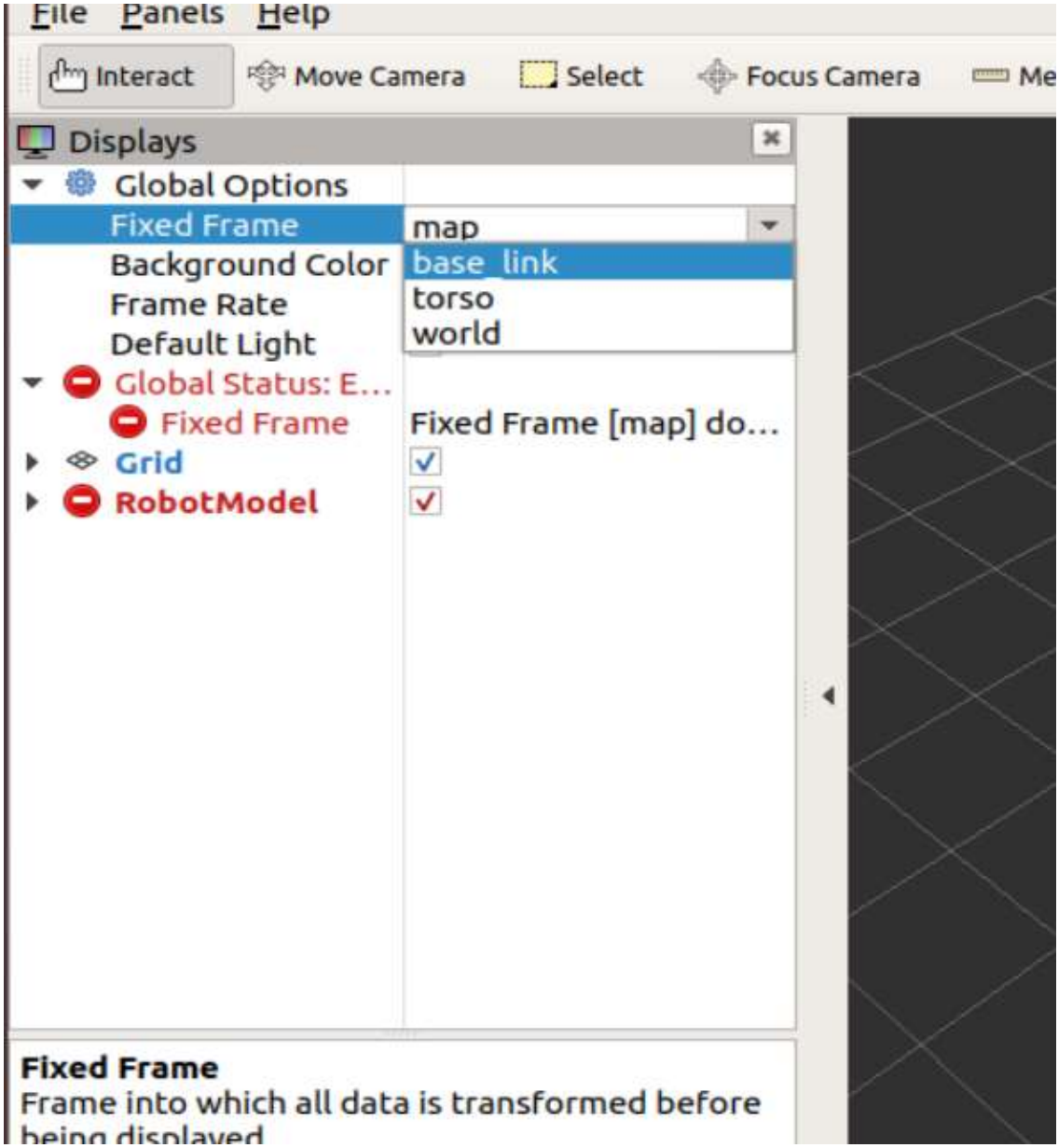
Lets run **RVIZ** and be amazed

```
rosrun rviz rviz
```

In rviz press the "add" button and select "RobotModel".

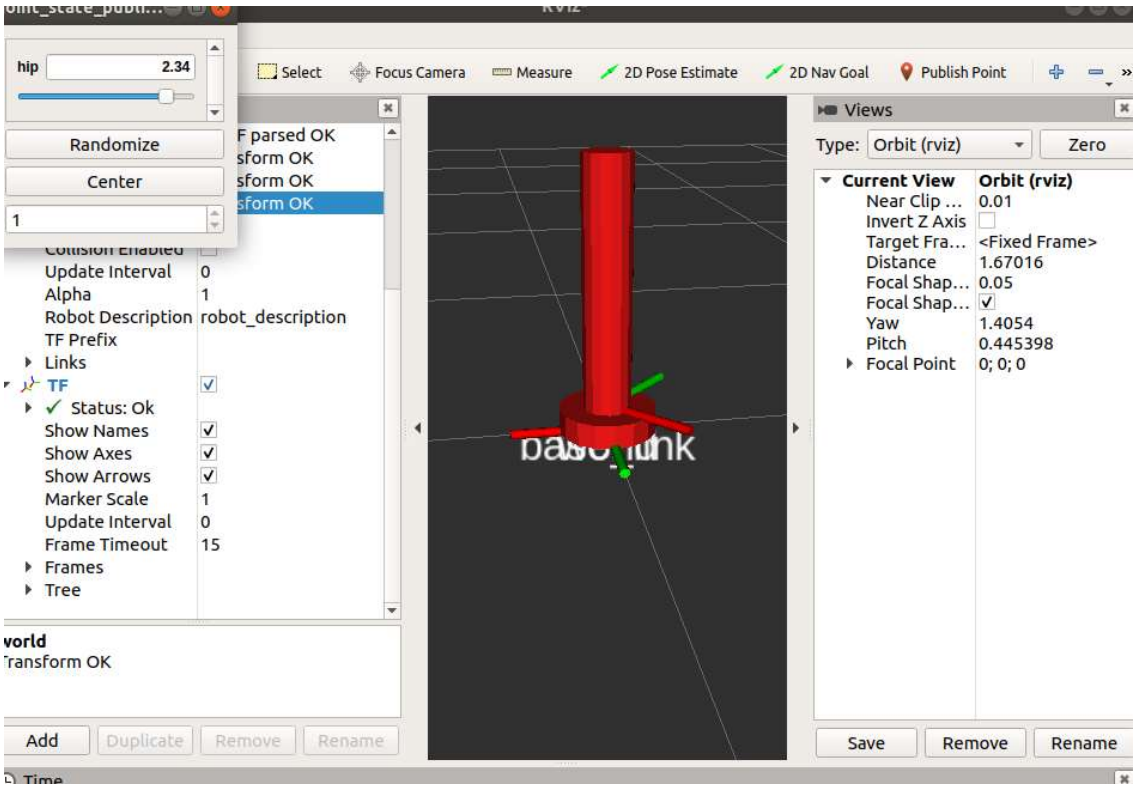


You might get an error here. This can be solved by setting "Fixed Frame" in the "Global Options" menu to "base_link":



Now the very simple robot should be visible in the rviz window. Open the joint_state_publisher window and play around with the slider. This should make the robot link rotate although it may be difficult to see.

Let's show the TF in rviz. Press the "add" button again and select "tf". Make sure the "show axes" box is ticked and you should be able to see the axes of the two frames. This will also make it easier to see when you rotate the torso.



Exercise 3.8 - URDF Let's build it up a bit

Exercise 3.8 - URDF Let's build it up a bit

We are still creating our robot from scratch. Let's add another part to our robot.

We do that by creating a file named **hello_ros_robot_3**.urdf and save it to **~/catkin_ws/src/hello_ros/urdf/hello_ros_robot_3.urdf:**

```
<?xml version="1.0"?>
<robot name="hello_ros_robot">
  <link name="world"/>
  <link name="base_link">
```

```
  <visual>
    <geometry>
      <cylinder length="0.05" radius="0.1"/>
    </geometry>
    <material name="silver">
      <color rgba="0.75 0.75 0.75 1"/>
    </material>
    <origin rpy="0 0 0" xyz="0 0 0.025"/>
  </visual>
</link>
<joint name="fixed" type="fixed">
  <parent link="world"/>
  <child link="base_link"/>
</joint>
<link name="torso">
  <visual>
    <geometry>
      <cylinder length="0.5" radius="0.05"/>
    </geometry>
    <material name="silver">
      <color rgba="0.75 0.75 0.75 1"/>
    </material>
    <origin rpy="0 0 0" xyz="0 0 0.25"/>
  </visual>
</link>
<joint name="hip" type="continuous">
  <axis xyz="0 0 1"/>
  <parent link="base_link"/>
  <child link="torso"/>
  <origin rpy="0 0 0" xyz="0.0 0.0 0.05"/>
</joint>
<link name="upper_arm">
  <visual>
    <geometry>
      <cylinder length="0.4" radius="0.05"/>
    </geometry>
    <material name="silver"/>
    <origin rpy="0 0 0" xyz="0 0 0.2"/>
  </visual>
</link>
<joint name="shoulder" type="continuous">
  <axis xyz="0 1 0"/>
  <parent link="torso"/>
```

```
    <child link="upper_arm"/>
    <origin rpy="0 1.5708 0" xyz="0.0 -0.1 0.45"/>
  </joint>
</robot>
```

Lets make sure that it has no errors:

```
roscd hello_ros/urdf/
check_urdf hello_ros_robot_3.urdf
```

In order to load the model of our robot we need to use the:

joint_state_publisher and robot_state_publisher

Let's create the ros launch file: **urdf_launch_3.launch** to run this:

```
<launch>
  <!-- Load the hello_ros_robot URDF model into the parameter server -->
    <param name="robot_description" textfile="$(find
hello_ros)/urdf/hello_ros_robot_3.urdf" />
  <param name="use_gui" value="true" />
  <node name="joint_state_publisher" pkg="joint_state_publisher"
        type="joint_state_publisher"/>
  <node name="robot_state_publisher" pkg="robot_state_publisher"
        type="robot_state_publisher"/>
</launch>
```

This time when you run the launch file, there should be a new window opening with two sliders.

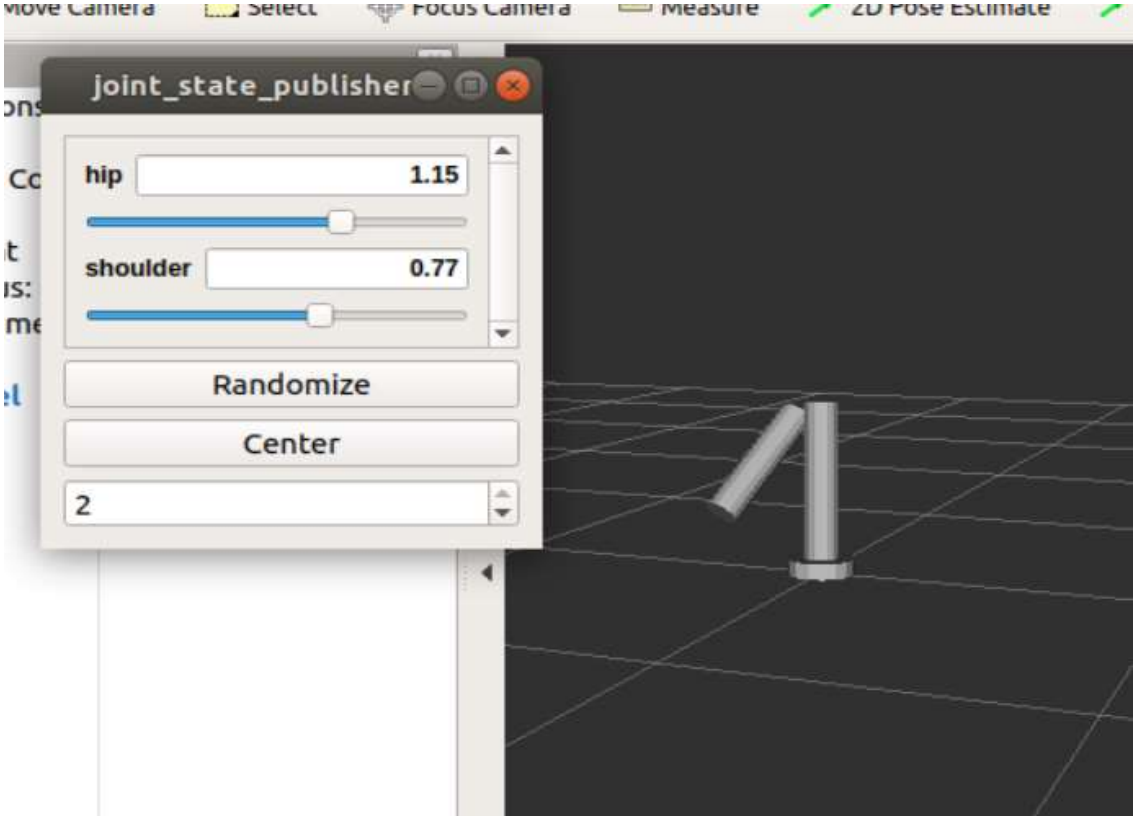
Lets see what tf says:

```
rosrun tf view_frames
evince frames.pdf
```

Lets run **RVIZ** and be amazed once again.

```
rosrun rviz rviz
```

Load the model as you did before. Play around with the sliders again. This time it should be easy to see the robot arm moving.



Exercise 3.9 - Gazebo there must be more to it

Exercise 3.9 - Gazebo There must be more to it

Ok, lets try to simulate our robot..

In order to do so we will need to create the mass, inertia, and collision parameters.

Lets create a file named **hello_gazebo_robot_1.urdf** and save it to **~/catkin_ws/src/hello_ros/urdf/hello_gazebo_robot_1.urdf:**

```
<?xml version="1.0"?>
<robot name="hello_ros_robot">
```

```
<link name="world"/>
<link name="base_link">
 <visual>
  <geometry>
   <cylinder length="0.05" radius="0.1"/>
  </geometry>
  <material name="silver">
   <color rgba="0.75 0.75 0.75 1"/>
  </material>
  <origin rpy="0 0 0" xyz="0 0 0.025"/>
 </visual>
 <collision>
  <geometry>
   <cylinder length="0.05" radius="0.1"/>
  </geometry>
  <origin rpy="0 0 0" xyz="0 0 0.025"/>
 </collision>
 <inertial>
  <mass value="1.0"/>
  <origin rpy="0 0 0" xyz="0 0 0.025"/>
  <inertia ixx="0.0027" iyy="0.0027" izz="0.005" ixy="0" ixz="0"
iyz="0"/>
 </inertial>
</link>
<joint name="fixed" type="fixed">
 <parent link="world"/>
 <child link="base_link"/>
</joint>
<link name="torso">
 <visual>
  <geometry>
   <cylinder length="0.5" radius="0.05"/>
  </geometry>
  <material name="silver">
   <color rgba="0.75 0.75 0.75 1"/>
  </material>
  <origin rpy="0 0 0" xyz="0 0 0.25"/>
 </visual>
 <collision>
  <geometry>
   <cylinder length="0.5" radius="0.05"/>
  </geometry>
  <origin rpy="0 0 0" xyz="0 0 0.25"/>
```

```
    </collision>
    <inertial>
      <mass value="1.0"/>
      <origin rpy="0 0 0" xyz="0 0 0.25"/>
      <inertia ixx="0.02146" iyy="0.02146" izz="0.00125" ixy="0" ixz="0"
iyz="0"/>
    </inertial>
  </link>
  <joint name="hip" type="continuous">
    <axis xyz="0 0 1"/>
    <parent link="base_link"/>
    <child link="torso"/>
    <origin rpy="0 0 0" xyz="0.0 0.0 0.05"/>
  </joint>
  <link name="upper_arm">
    <visual>
      <geometry>
        <cylinder length="0.4" radius="0.05"/>
      </geometry>
      <material name="silver"/>
      <origin rpy="0 0 0" xyz="0 0 0.2"/>
    </visual>
    <collision>
      <geometry>
        <cylinder length="0.4" radius="0.05"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0.2"/>
    </collision>
    <inertial>
      <mass value="1.0"/>
      <origin rpy="0 0 0" xyz="0 0 0.2"/>
      <inertia ixx="0.01396" iyy="0.01396" izz="0.00125" ixy="0" ixz="0"
iyz="0"/>
    </inertial>
  </link>
  <joint name="shoulder" type="continuous">
    <axis xyz="0 1 0"/>
    <parent link="torso"/>
    <child link="upper_arm"/>
    <origin rpy="0 1.5708 0" xyz="0.0 -0.1 0.45"/>
  </joint>
</robot>
```

Lets make sure that it has no errors:

```
roscd hello_ros/urdf/
check_urdf hello_gazebo_robot_1.urdf
```

In order to load the model of our robot we need to use the:

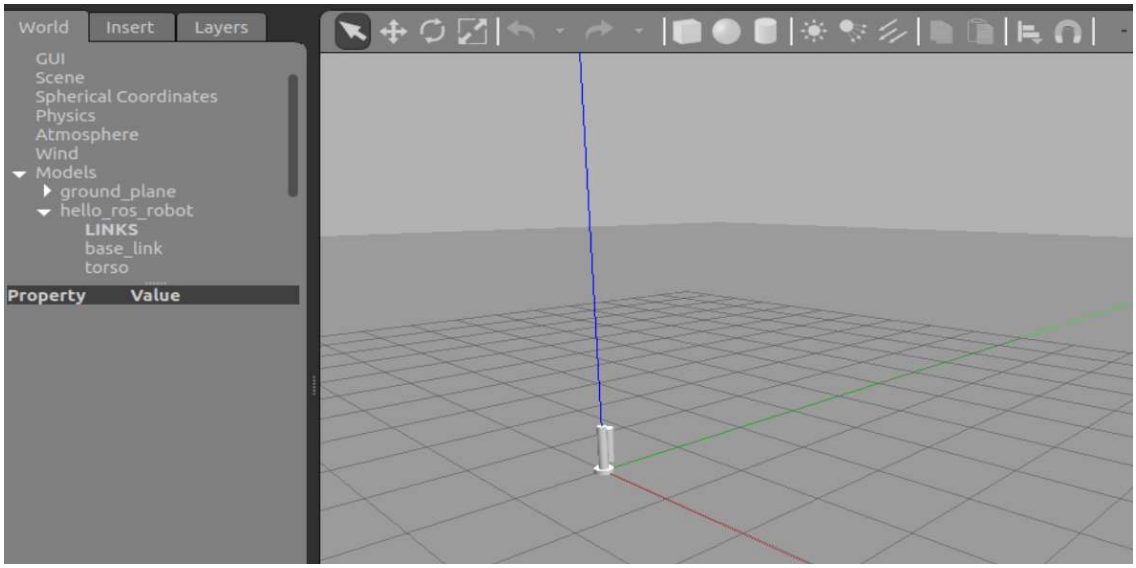
joint_state_publisher and robot_state_publisher

Let's create the ros launch file: **spawn_gazebo_1.launch** to run this. As you can read by the comments it loads our robot into the parameter server, opens an empty gazebo window and loads our robot from the parameter server:

```
<launch>
  <!-- Load the helo_ros_robot URDF model into the parameter server -->
  <param name="robot_description" textfile="$(find
hello_ros)/urdf/hello_gazebo_robot_1.urdf" />
  <!-- Start Gazebo with an empty world -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch"/>
  <!-- Spawn a helo_ros_robot in Gazebo, taking the description from the
  parameter server -->
  <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
    args="-param robot_description -urdf -model hello_ros_robot" />
</launch>
```

Let's launch it!!

Gazebo should now open and you should be able to see our robot in it:



Exercise 3.10 - Gazebo Let's control our robot

Exercise 3.10 - Gazebo Let's control our robot

So our robot is a numb pendulum?

We will add some motors and a controller and see where it gets us:

Lets create a file named **hello_gazebo_robot_2.urdf** and save it to **~/catkin_ws/src/hello_ros/urdf/hello_gazebo_robot_2.urdf:**

```
<?xml version="1.0"?>
<robot name="hello_ros_robot">
  <link name="world"/>
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.05" radius="0.1"/>
      </geometry>
      <material name="silver">
        <color rgba="0.75 0.75 0.75 1"/>
      </material>
      <origin rpy="0 0 0" xyz="0 0 0.025"/>
    </visual>
    <collision>
      <geometry>
        <cylinder length="0.05" radius="0.1"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0.025"/>
    </collision>
    <inertial>
      <mass value="1.0"/>
      <origin rpy="0 0 0" xyz="0 0 0.025"/>
      <inertia ixx="0.0027" iyy="0.0027" izz="0.005" ixy="0" ixz="0"
iyz="0"/>
    </inertial>
  </link>
```

```
<joint name="fixed" type="fixed">
  <parent link="world"/>
  <child link="base_link"/>
</joint>
<link name="torso">
  <visual>
    <geometry>
      <cylinder length="0.5" radius="0.05"/>
    </geometry>
    <material name="silver">
      <color rgba="0.75 0.75 0.75 1"/>
    </material>
    <origin rpy="0 0 0" xyz="0 0 0.25"/>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.5" radius="0.05"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0 0 0.25"/>
  </collision>
  <inertial>
    <mass value="1.0"/>
    <origin rpy="0 0 0" xyz="0 0 0.25"/>
    <inertia ixx="0.02146" iyy="0.02146" izz="0.00125" ixy="0" ixz="0"
iyz="0"/>
  </inertial>
</link>
<joint name="hip" type="continuous">
  <axis xyz="0 0 1"/>
  <parent link="base_link"/>
  <child link="torso"/>
  <origin rpy="0 0 0" xyz="0.0 0.0 0.05"/>
</joint>
<link name="upper_arm">
  <visual>
    <geometry>
      <cylinder length="0.4" radius="0.05"/>
    </geometry>
    <material name="silver"/>
    <origin rpy="0 0 0" xyz="0 0 0.2"/>
  </visual>
  <collision>
    <geometry>
```

```
      <cylinder length="0.4" radius="0.05"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0 0 0.2"/>
  </collision>
  <inertial>
    <mass value="1.0"/>
    <origin rpy="0 0 0" xyz="0 0 0.2"/>
    <inertia ixx="0.01396" iyy="0.01396" izz="0.00125" ixy="0" ixz="0"
iyz="0"/>
  </inertial>
 </link>
 <joint name="shoulder" type="continuous">
  <axis xyz="0 1 0"/>
  <parent link="torso"/>
  <child link="upper_arm"/>
  <origin rpy="0 1.5708 0" xyz="0.0 -0.1 0.45"/>
 </joint>
 <transmission name="tran0">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="hip">
    <hardwareInterface>PositionJointInterface</hardwareInterface>
  </joint>
  <actuator name="motor0">
    <hardwareInterface>PositionJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
 </transmission>
 <transmission name="tran1">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="shoulder">
    <hardwareInterface>PositionJointInterface</hardwareInterface>
  </joint>
  <actuator name="motor1">
    <hardwareInterface>PositionJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
 </transmission>
 <gazebo>
  <plugin name="control" filename="libgazebo_ros_control.so"/>
 </gazebo>
 <gazebo>
  <plugin name="joint_state_publisher"
filename="libgazebo_ros_joint_state_publisher.so">
```

```
    <jointName>hip, shoulder</jointName>
  </plugin>
 </gazebo>
</robot>
```

Let's make sure we have the necessary libraries:

```
sudo apt-get install ros-melodic-gazebo-ros-control
sudo apt-get install ros-melodic-joint-trajectory-controller
```

Let's make sure that it has no errors:

```
roscd hello_ros/urdf/
check_urdf hello_gazebo_robot_2.urdf
```

In order to use the controller we will need to create small configuration file like this:

Lets create a file named **controllers_simple.yaml** and save it to **~/catkin_ws/src/hello_ros/udrf/controllers_simple.yaml:**

```
arm_controller:
  type: "position_controllers/JointTrajectoryController"
  joints:
    - hip
    - shoulder
```

Let's create the ros launch file: **spawn_gazebo_2.launch** to run this:

```
<launch>
  <!-- Load the helo_ros_robot URDF model into the parameter server -->
  <param name="robot_description" textfile="$(find
hello_ros)/urdf/hello_gazebo_robot_2.urdf" />
  <!-- Start Gazebo with an empty world -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch"/>
  <!-- Spawn a helo_ros_robot in Gazebo, taking the description from the
  parameter server -->
  <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
    args="-param robot_description -urdf -model hello_ros_robot" />

  <rosparam file="$(find hello_ros)/urdf/controllers_simple.yaml"
```

```
command="load"/>
    <node name="controller_spawner" pkg="controller_manager"
type="spawner"
     args="arm_controller"/>
    <!-- Convert /joint_states messages published by Gazebo to /tf
messages,
     e.g., for rviz-->
    <node name="robot_state_publisher" pkg="robot_state_publisher"
     type="robot_state_publisher"/>
</launch>
```

Launch the file as you have done many times before. This will open Gazebo.

Now we have to command our robot to a position. Type the following into a new terminal and watch the arm in Gazebo move:

```
rostopic pub /arm_controller/command trajectory_msgs/JointTrajectory
'{joint_names: ["hip", "shoulder"], points: [{positions: [0.1, -0.5],
time_from_start:[1.0,0.0]}]}'
```

Add a lower arm and an elbow to the robot. **Hint:** You should be able to reuse a lot of the given material.

Finally, create a ROS node that publishes random trajectories to the robot.