
Gazebo Simulator

Unit 2: Build a Robot

- Summary -

Estimated time to completion: **4 hours**

This unit is about creating robots for Gazebo simulations using the **URDF** language. First, you will go through the necessary elements of URDF to achieve the creation of the robots. You can check the complete URDF documentation here: <http://wiki.ros.org/urdf> (<http://wiki.ros.org/urdf>), and full XML specifications here: <http://wiki.ros.org/urdf/XML> (<http://wiki.ros.org/urdf/XML>).

- End of Summary -

Why URDF instead of SDF?

Gazebo adopts the **SDF** format to describe **worlds**, **models**, **robots**, and other necessary components to build entire simulations. So why are you building robots with URDF?

ROS tools are built for URDF, for example, **RVIZ**. Gazebo converts URDF files to SDF under the hood whenever you spawn a URDF-defined robot in a Gazebo simulation.

It will be covered in the following unit when you spawn the robot into a Gazebo simulation and make it work with ROS.

The official reference can be found here: http://gazebosim.org/tutorials?tut=ros_urdf (http://gazebosim.org/tutorials?tut=ros_urdf).

2.1 Creating a Mobile Robot

To create a new robot model, have the robot definition files and folders in a specific structure.

Besides the robot definition file, a **URDF** file, you will use launch files to debug the robot creation.

Create a package and new folders in it:

► Execute

```
In [ ]: cd ~/catkin_ws/src/
```


```
In [ ]: catkin_create_pkg robot_description rospy urdf xacro
```

```
In [ ]: mkdir -p robot_description/urdf
```

```
In [ ]: touch robot_description/urdf/robot.urdf
```

The **URDF** folder will contain the robot description files.

Paste the content below to the **robot.urdf** file:

 Copy and paste

```
In [ ]: <?xml version="1.0" ?>

<robot name="robot">
  <!-- Link - chassis -->
  <link name="link_chassis">
    <inertial>
      <mass value="10" />
      <origin xyz="0 0 0.3" rpy="0 0 0" />
      <inertia ixx="1.5417" ixy="0" ixz="0" iyy="3.467" iyz="0" izz="4.7" />
    </inertial>

    <collision>
      <geometry>
        <box size="2 1.3 0.4" />
      </geometry>
    </collision>

    <visual>
      <geometry>
        <box size="2 1.3 0.4" />
      </geometry>
      <material name="Red">
        <color rgba="1 0 0 1" />
      </material>
    </visual>
  </link>
</robot>
```

Create a new launch file in the package, using the commands below:

► Execute

```
In [ ]: cd ~/catkin_ws/src/
```

```
In [ ]: mkdir -p robot_description/launch
```

```
In [ ]: touch robot_description/launch/rviz.launch
```

The **launch** folder will contain the necessary launch files to debug this robot model using **RVIZ**.

Copy and paste the following content:

 Copy and paste

```
In [ ]: <?xml version="1.0"?>
<launch>

  <param name="robot_description" command="cat '$(find robot_description)/urdf

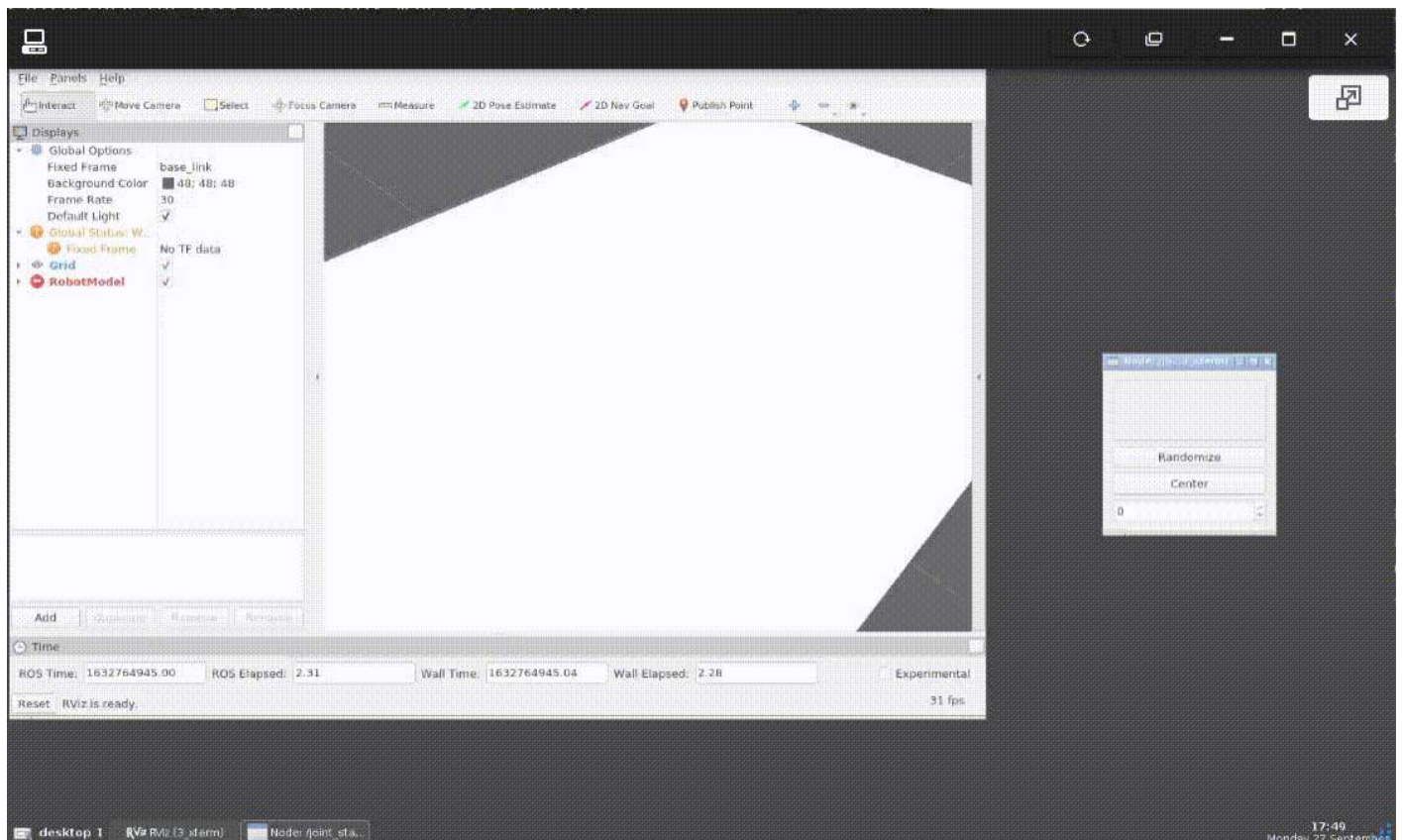
  <!-- send fake joint values -->
  <node name="joint_state_publisher_gui" pkg="joint_state_publisher_gui" type=

  <!-- Combine joint values -->
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_s

  <!-- Show in Rviz -->
  <node name="rviz" pkg="rviz" type="rviz" />

</launch>
```

Launch that file and wait for the **Graphical Tools** to open. You must have the result below. Next, change the **fixed frame** in RVIZ to **link_chassis**. This will make the base link of the robot the reference for the **RVIZ** model visualizer. Save that configuration in your workspace. The next time, the robot model will be shown correctly on RVIZ startup.



- End of Creating folders and files -

2.2 Links and Joints

- Links -

Links

You have created your first robot model with a single element, just a link. What composes a link?

Every link must have the following attributes:

Inertial

It contains **mass (kg)**, **origin (m)** and **inertia matrix (kg . m²)**.

```
<inertial>
  <mass value="10" />
  <origin
    xyz="0 0 0.3"
    rpy="0 0 0" />
  <inertia
    ixx="1.5417"
    ixy="0"
    ixz="0"
    iyy="3.467"
    iyz="0"
    izz="4.742" />
</inertial>
```

Collision and Visual

Collision and visual attributes are identical in that case, but they have different purposes. The simulator uses **Collision** to calculate physical link interactions. Visual is only for rendering purposes. They do not affect how a link interacts with others.

You may want to have different collision and visual if you have a very complex visual mesh, but do not want to overcalculate physical interaction, because it is not needed, or to save computational power.


```
<collision>
  <geometry>
    <box size="2 1.3 0.4" />
  </geometry>
</collision>

<visual>
  <geometry>
    <box size="2 1.3 0.4" />
  </geometry>
</visual>
```

Joints

Joints are used to bind links and create a movement relation between them. In this unit, your goal is to build a mobile robot, so put a wheel in the chassis you have created.

Still in the **robot.urdf** file, add the following code after the **</link>** closing tag:

 Copy and paste

In []:

```
<!-- Joint - chassis / left wheel -->
<joint name="joint_chassis_left_wheel" type="continuous">
  <origin rpy="0 0 0" xyz="-0.5 0.65 0" />
  <child link="link_left_wheel" />
  <parent link="link_chassis" />
  <axis rpy="0 0 0" xyz="0 1 0" />
  <limit effort="10000" velocity="1000" />
  <joint_properties damping="1.0" friction="1.0" />
</joint>

<!-- Link - left wheel -->
<link name="link_left_wheel">
  <inertial>
    <mass value="1" />
    <origin xyz="0 0 0" rpy="0 0 0" />
    <inertia ixx="0.0025266666666667" ixy="0" ixz="0" iyy="0.0025266666666667"
  </inertial>

  <collision>
    <origin rpy="1.5707 0 0" xyz="0 0.18 0" />
    <geometry>
      <cylinder length="0.12" radius="0.4"/>
    </geometry>
  </collision>

  <visual>
    <origin rpy="1.5707 0 0" xyz="0 0.18 0" />
    <geometry>
      <cylinder length="0.12" radius="0.4"/>
    </geometry>
  </visual>

  <collision>
    <origin rpy="1.5707 0 0" xyz="0 0.06 0" />
    <geometry>
      <cylinder length="0.12" radius="0.08"/>
    </geometry>
  </collision>

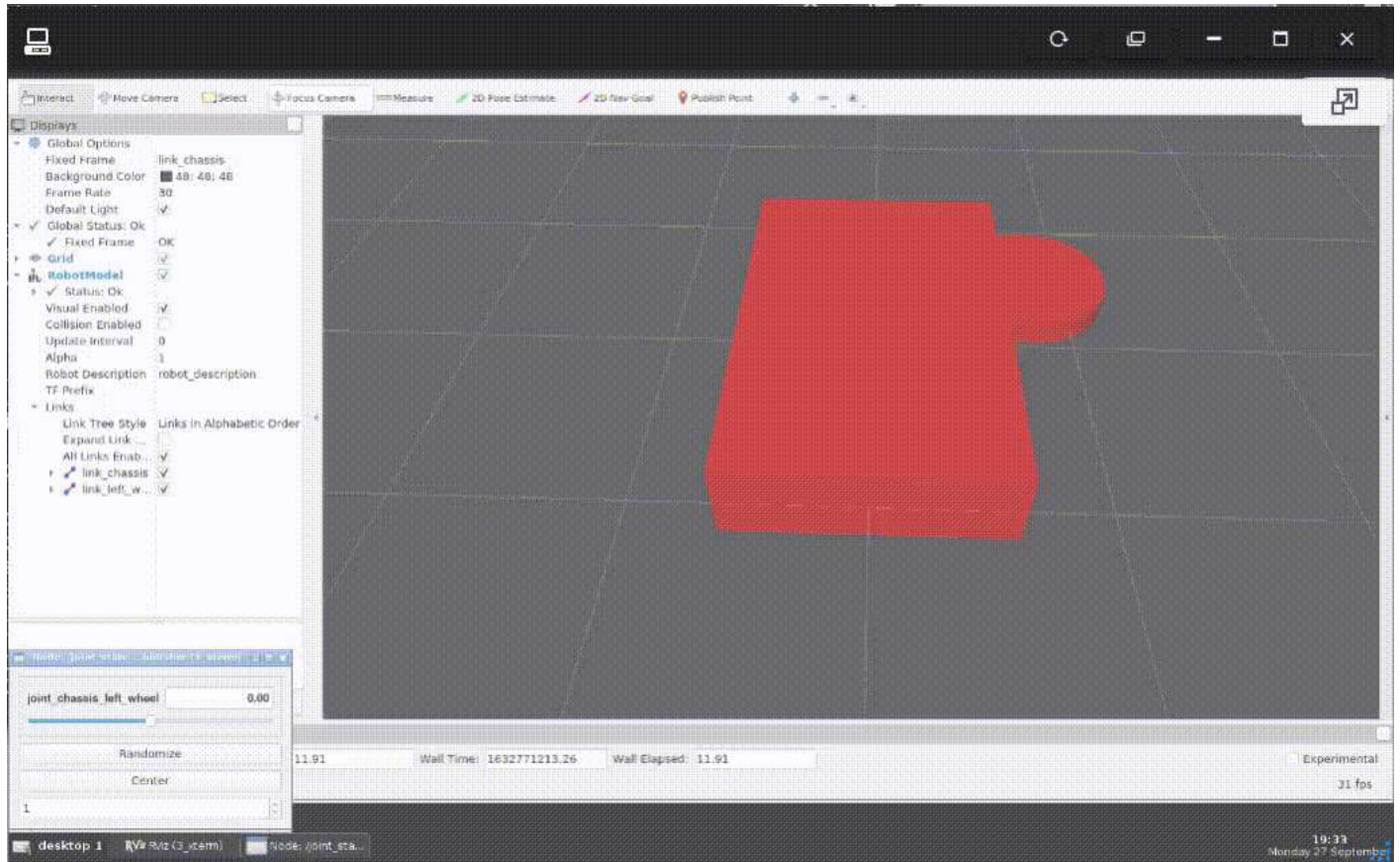
  <visual>
    <origin rpy="1.5707 0 0" xyz="0 0.06 0" />
    <geometry>
      <cylinder length="0.12" radius="0.08"/>
    </geometry>
  </visual>
</link>
```


Wheel link definition

You have two collision and visual tags. Each one has a different name. It does not matter how many you have for a single link. They will be displayed in relation to the joint position.

In that case, you created a pair of visual/collisions for the wheel and its axle.

Relaunch the **rviz.launch** file. You must have the following result:



Of course, this is not a wheel's desired position or rotation. What is wrong there?

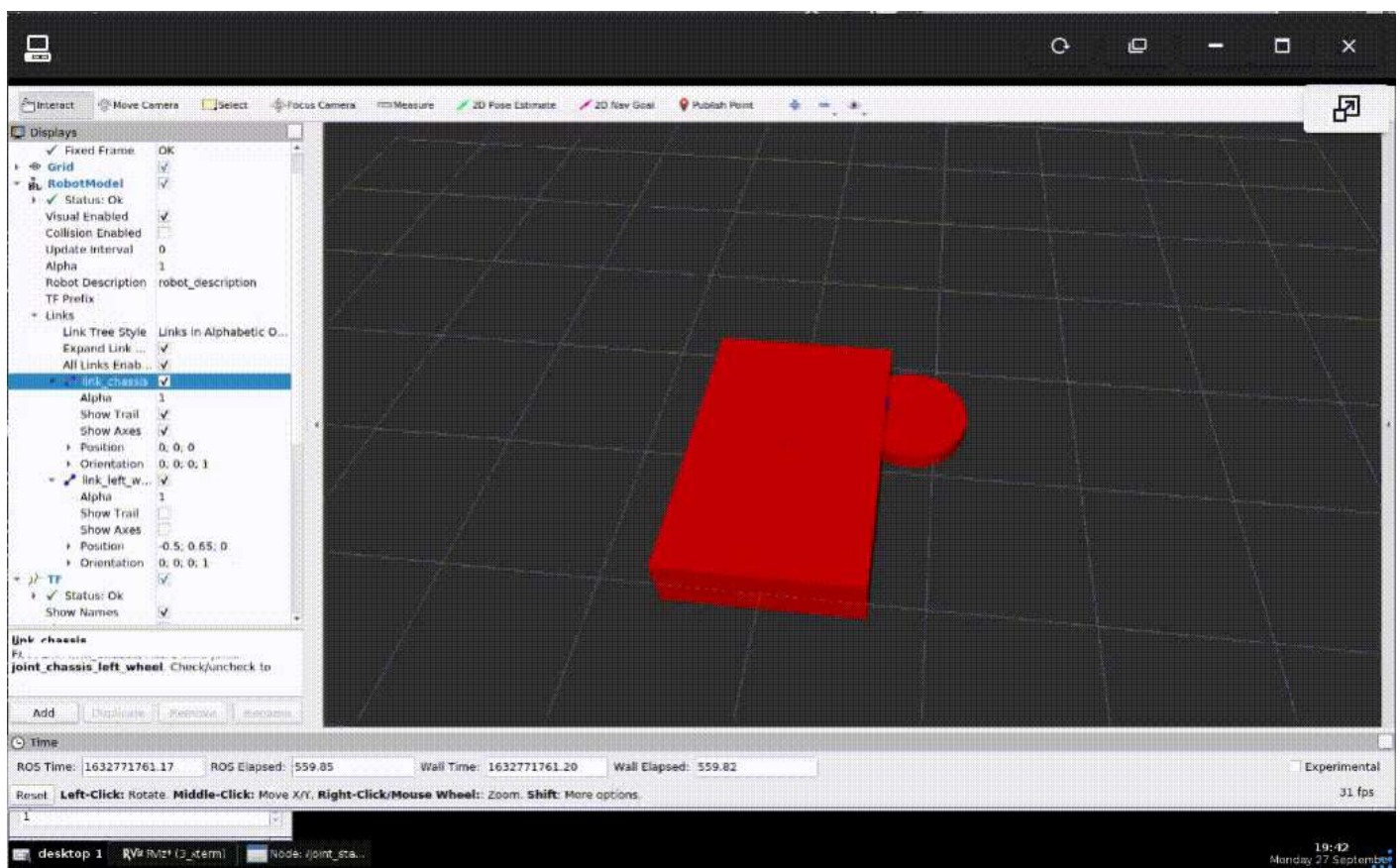
Joint definition

Check the joint definition:

```
<joint name="joint_chassis_left_wheel" type="continuous">
  <origin rpy="0 0 0" xyz="-0.5 0.88 0" />
  <child link="link_left_wheel" />
  <parent link="link_chassis" />
  <axis rpy="0 0 0" xyz="0 1 0" />
  <limit effort="10000" velocity="1000" />
  <joint_properties damping="1.0" friction="1.0" />
</joint>
```

The first attribute **type** defines the kind of movement the link is allowed to perform. Some types are: **continuous**, **revolute**, **prismatic**, and **fixed**. There are others, and you can check the complete reference here:

<http://wiki.ros.org/urdf/XML/joint> (<http://wiki.ros.org/urdf/XML/joint>).



The **origin** attribute has to be defined where the joint will be placed with respect to the chassis. The chassis link starts at **0 0 0**, and it has a length of 2 meters and a width of 1.3 meters. The chassis increases in both directions, which means the border of its side is at $\pm(1.3/2)$, which is **0.65 meters**.

Then, define the **child** and **parent** links. Skip the **axis**, for instance. You have the **limit** to define maximum effort and velocity. This is not important right now. Here you are designing the shape of the robot. And **joint properties**, which are also related to the movement dynamic, will be taken into account further.

Finally, go to the **axis** definition. This is where you define around which axis the link will rotate. In that case, you want the link to rotate around the Y-axis. This is correct.

Fix link definition to respect joint rotation

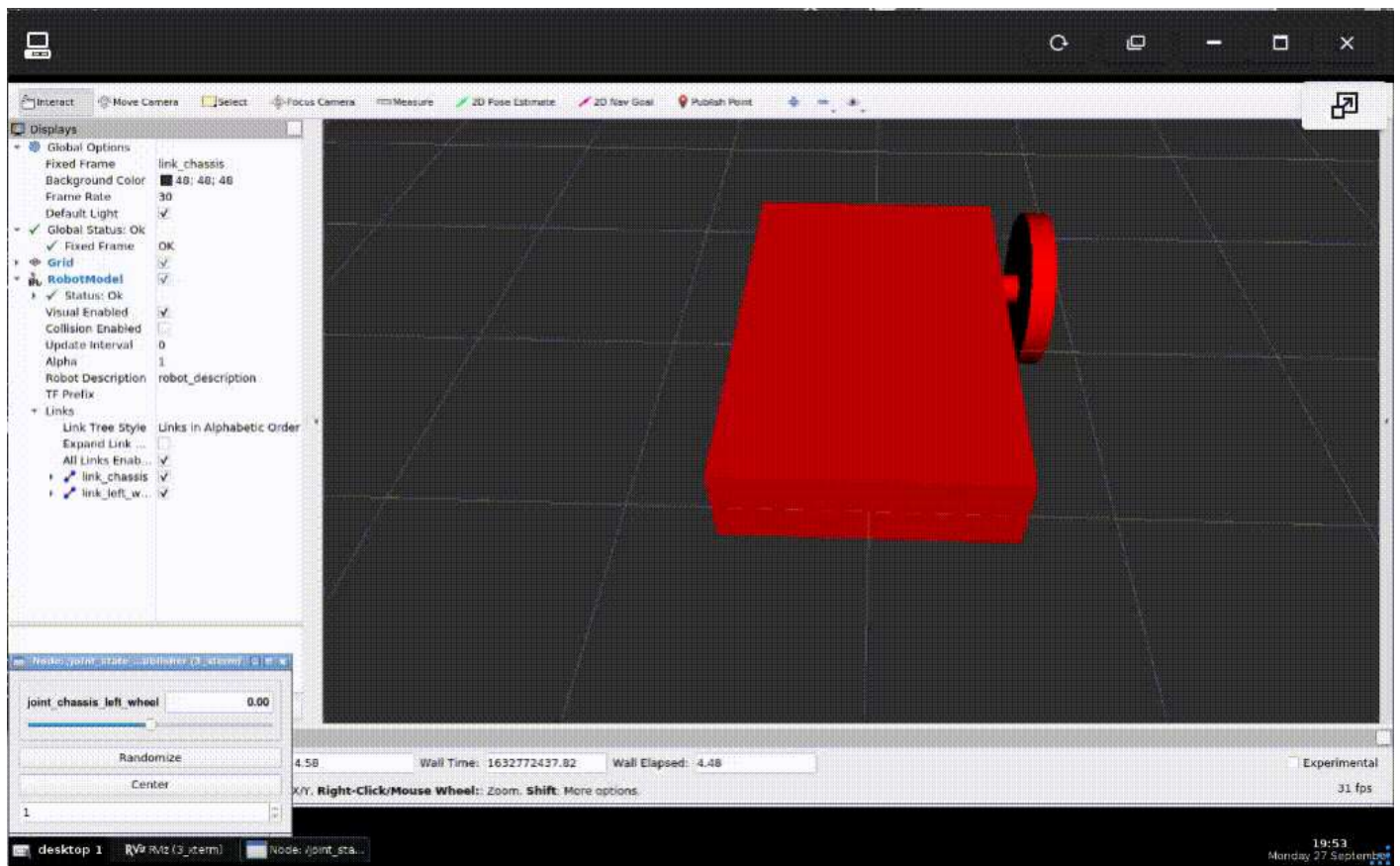
It turns out that your **link** cylinder geometry (length and radius) is defined with respect to the Z-axis. So do a rotation around X to fix it.

Back to the left wheel joint definition. Change the **origin** attribute of collision and visual tags to rotate 90 degrees on the X-axis:

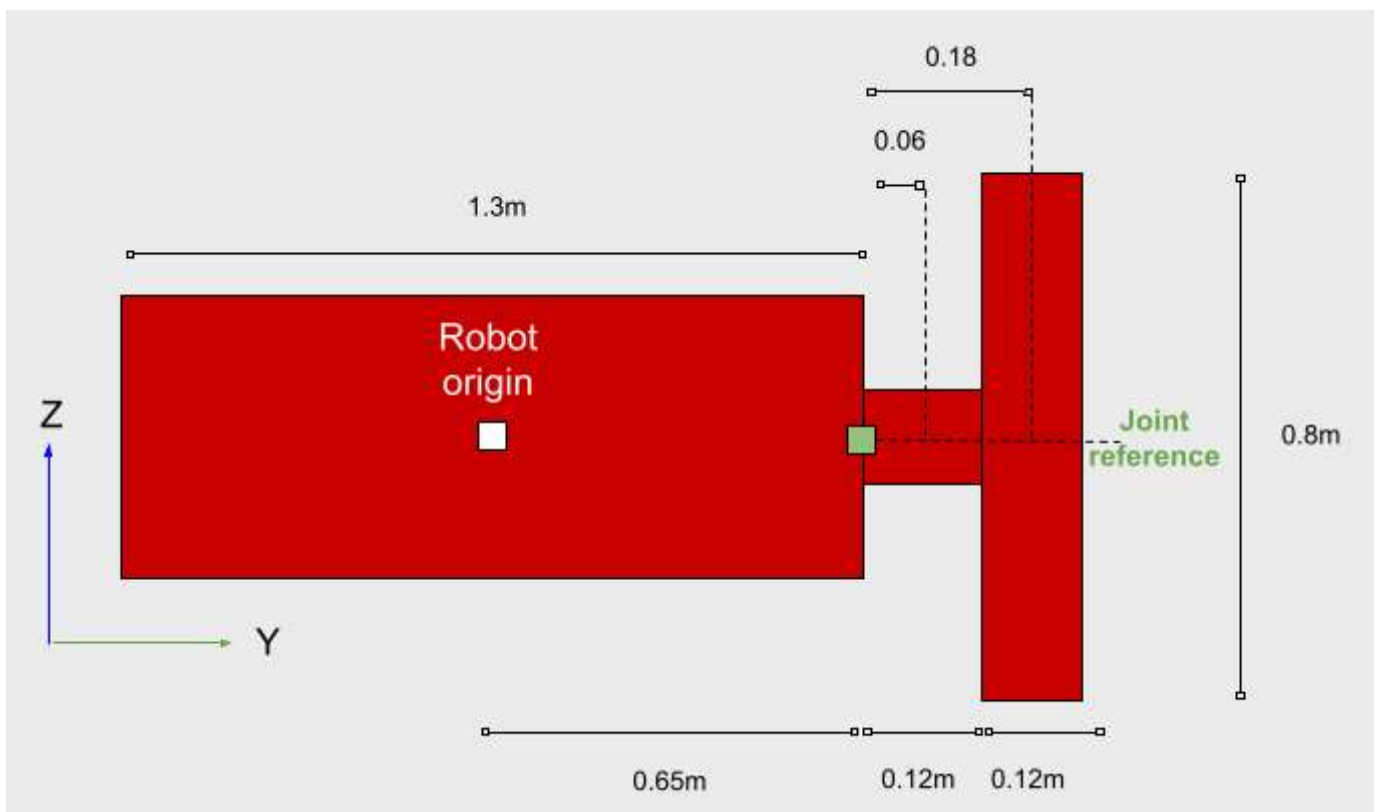
```
<link name="link_left_wheel">
  ...
  <collision>
    <origin rpy="1.5707 0 0" xyz="0 0.18 0" />
    <geometry>
      <cylinder length="0.12" radius="0.4"/>
    </geometry>
  </collision>
  <visual>
    <origin rpy="1.5707 0 0" xyz="0 0.18 0" />
    <geometry>
      <cylinder length="0.12" radius="0.4"/>
    </geometry>
  </visual>

  <collision>
    <origin rpy="1.5707 0 0" xyz="0 0.06 0" />
    <geometry>
      <cylinder length="0.12" radius="0.08"/>
    </geometry>
  </collision>
  <visual>
    <origin rpy="1.5707 0 0" xyz="0 0.06 0" />
    <geometry>
      <cylinder length="0.12" radius="0.08"/>
    </geometry>
  </visual>
  ...
```

Relaunch the **rviz.launch** file, and the expected result must be the following:



Check the dimensions of the robot below. You will understand better why the values are the ones used in the XML definition:

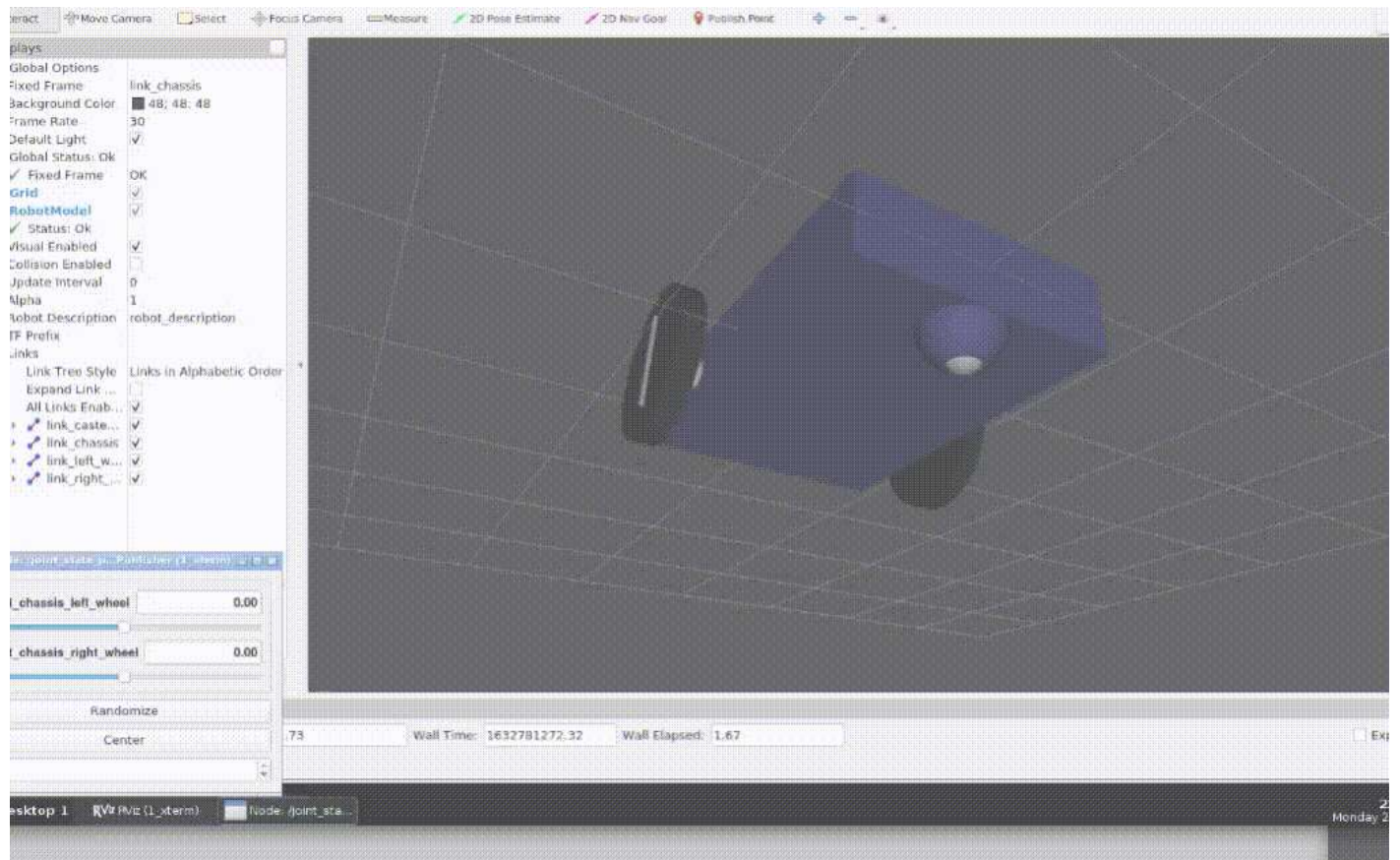


- End of Joints -

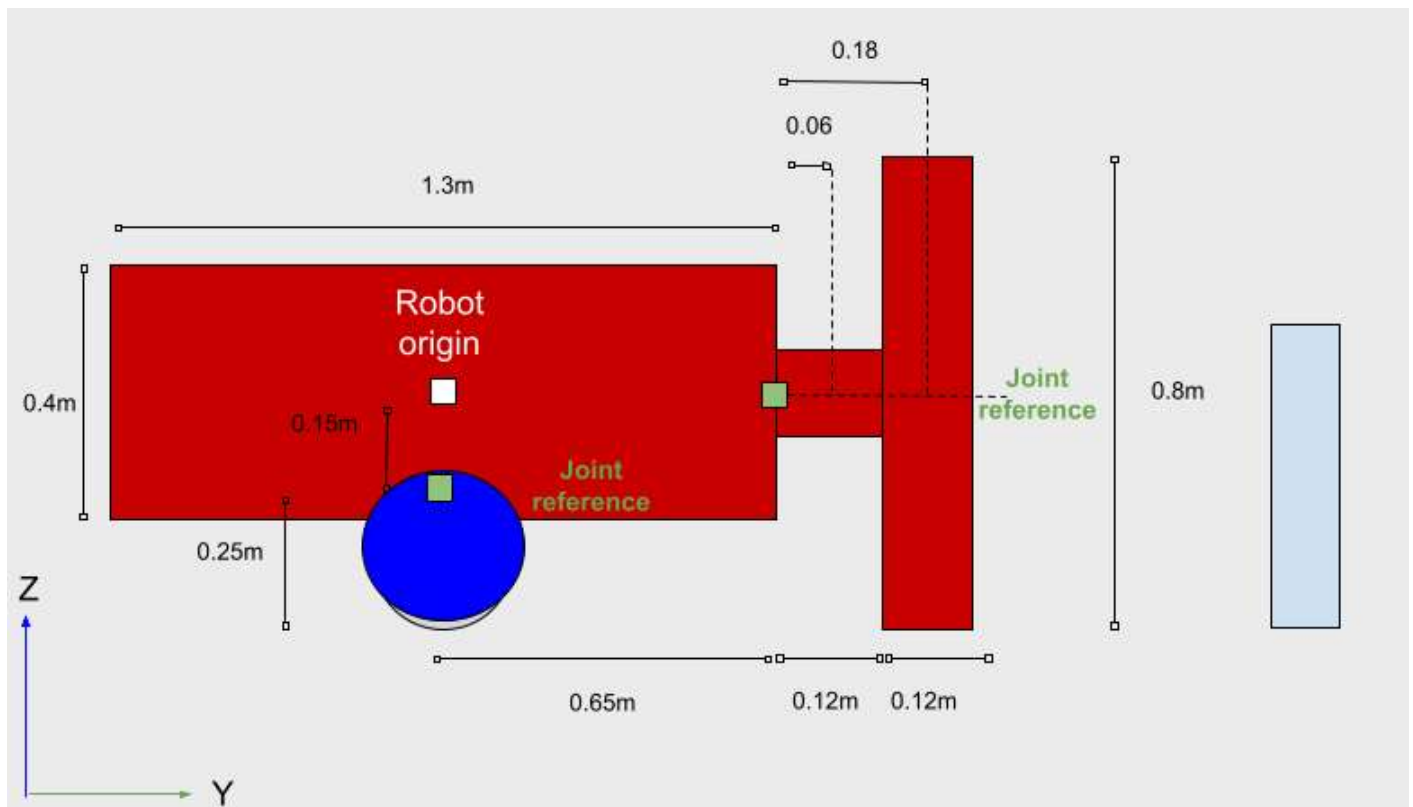
Improve the current model and finish it!

For this exercise, add a few more things to have the robot ready:

- Add the **right wheel**
- Add a new **visual** element to both wheels to make the spinning visible (i.e., hubcap)
- Add a **caster wheel** with a **fixed** joint to the front of the robot
- Change and add **material colors** of the chassis and wheels to make it look more or less like the image below



Use the image below as a reference to create the fixed joint and the visual components of the caster wheel.



- End of Exercise 2.2 -

- Solution for Exercise 2.2 -

Do not check the solution before trying to complete the exercise on your own! Instead, review the solution after completing the exercise. It is an excellent resource to compare your solution.

If you have problems solving the exercise, use our forum support!

[robot.urdf \(unit-02/robot-sol-ex2.2.urdf.txt\)](#)

- End of Solution for Exercise 2.2 -

- References -

You can have the full specification of links and joints here:

- <http://wiki.ros.org/urdf/XML/link> (<http://wiki.ros.org/urdf/XML/link>)
- <http://wiki.ros.org/urdf/XML/joint> (<http://wiki.ros.org/urdf/XML/joint>)

- End of References -

2.3 Meshes

- Meshes -

Meshes can be used to create more complex 3D geometric shapes. Imagine you have a real robot, the entire project was done using CAD software, and you need to simulate such a robot. You would have the model files. URDF supports **.stl** and **.dae** files. The last one is recommended for the best resolution.

See how to use a mesh in the robot you are working with.

Go to the **visual** section of the **chassis**, replace the code:

```
<geometry>
  <box size="2 1.3 0.4" />
</geometry>
```

By:

 Copy and paste

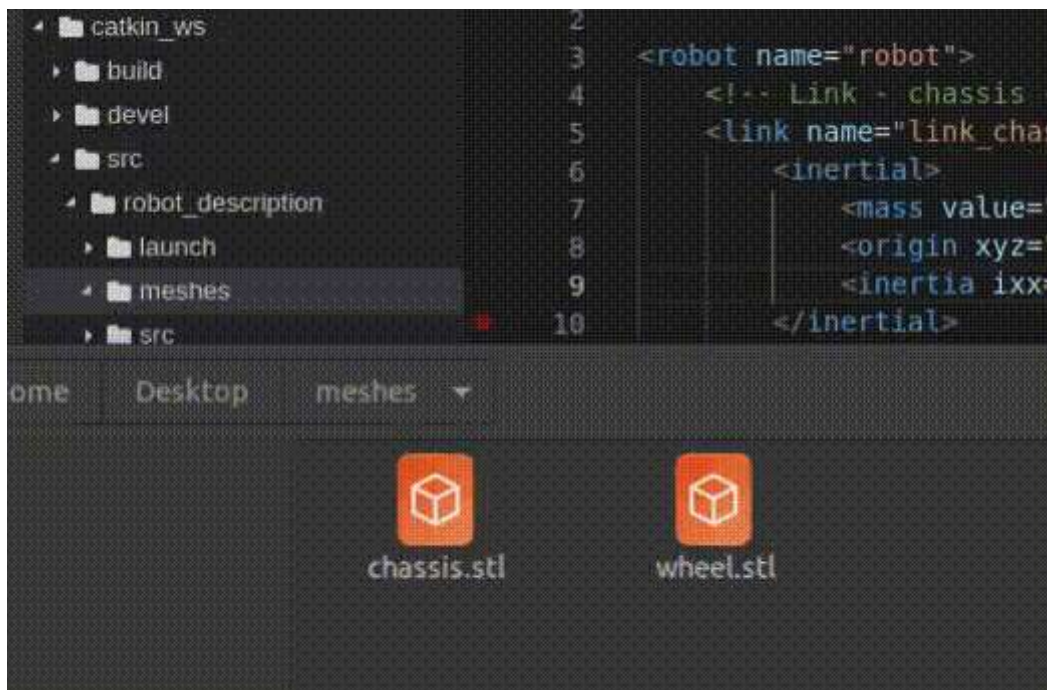
```
In [ ]: <geometry>
        <mesh filename="package://robot_description/meshes/chassis.stl" />
</geometry>
```

URDF understands the `package://` notation, which makes it easier to find files using their relative path. That means you must have a `meshes` folder in your package. Create this folder and upload the file below:

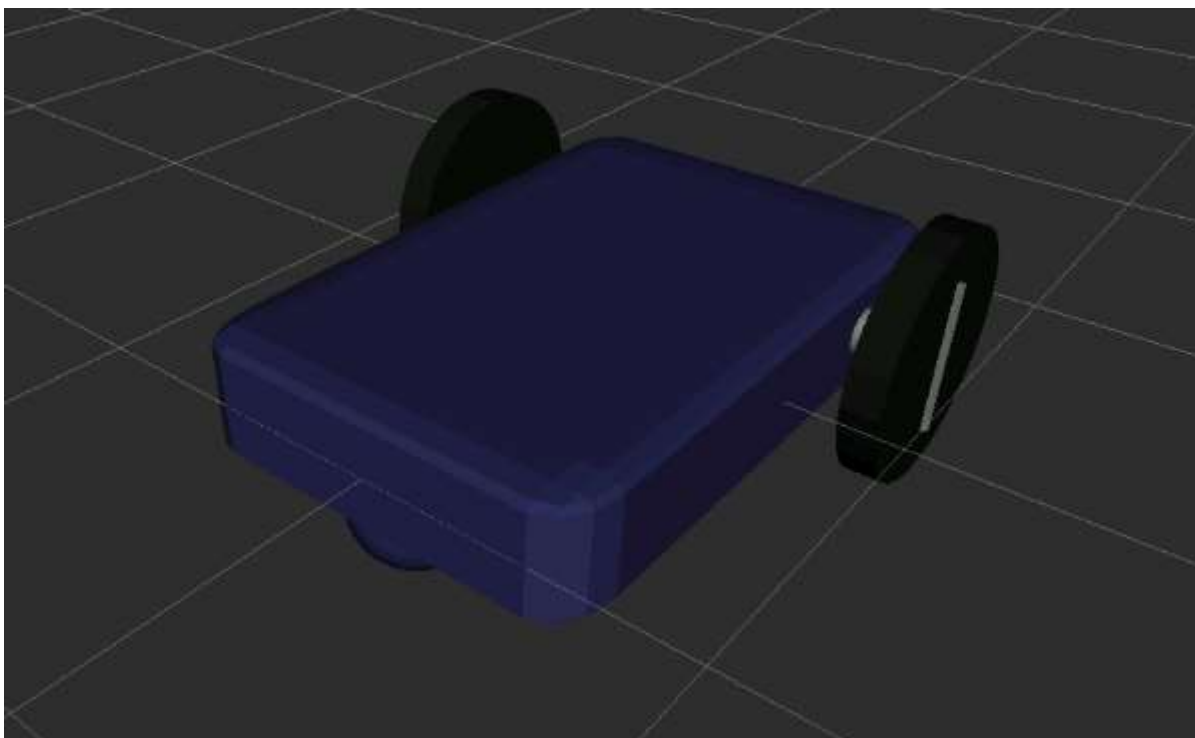
► Execute

```
In [ ]: mkdir -p ~/catkin_ws/src/robot_description/meshes/
```


[chassis.stl](https://s3.eu-west-1.amazonaws.com/readme.theconstructsim.com/_others_/course_gazebo_intro/chassis.stl) (https://s3.eu-west-1.amazonaws.com/readme.theconstructsim.com/_others_/course_gazebo_intro/chassis.stl)



Relaunch `rviz.launch` file, and the result must be the following:



Notice the **.stl** file uses the same material color set to the basic geometry shape.

- End of Meshes -

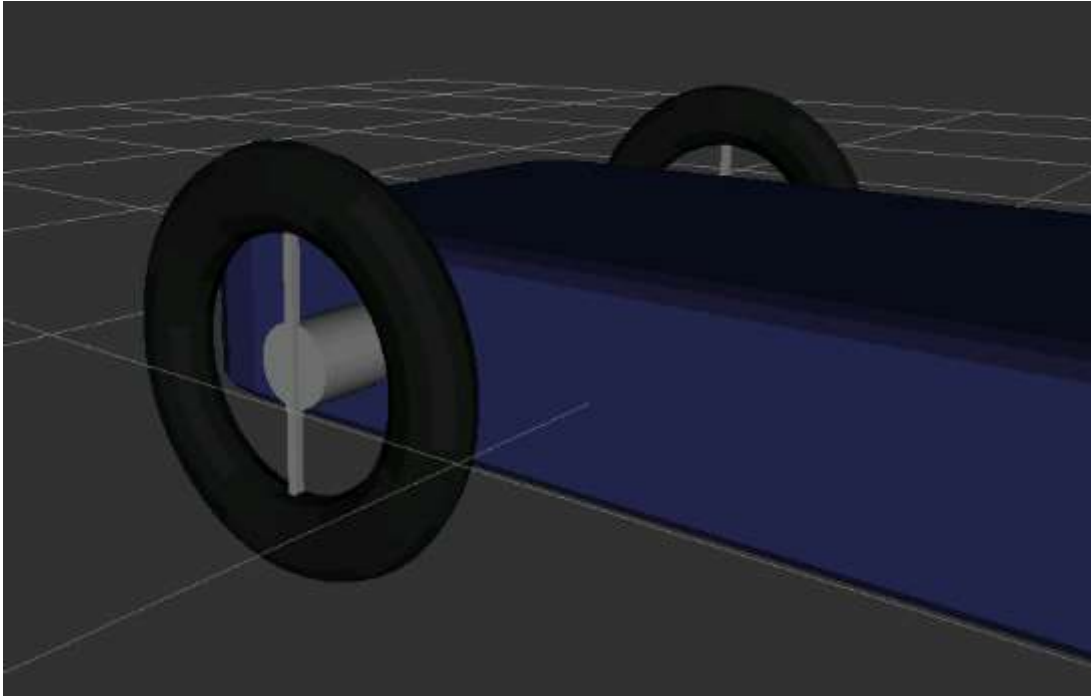
- Exercise 2.3 -

Add a mesh file to the tires.

Use the file below:

- [wheel.stl \(https://s3.eu-west-1.amazonaws.com/readme.theconstructsim.com/_others_/course_gazebo_intro/wheel.stl\)](https://s3.eu-west-1.amazonaws.com/readme.theconstructsim.com/_others_/course_gazebo_intro/wheel.stl)

Adjust the hubcap visual to make the robot look like the below:



- End of Exercise 2.3 -

- Solution for Exercise 2.3 -

Do not check the solution before trying it on your own! Instead, review the solution when you have completed the exercise. It is a good resource to compare your solution.

If you have problems completing the exercise, use our forum support!

[robot.urdf \(unit-02/robot-sol-ex2.3.urdf.txt\)](#)

- End of Solution for Exercise 2.3 -

2.4 Spawn the Robot in Gazebo

You have built the robot, and it has a chassis and wheels. Therefore, it should be ready to simulate! See how to put it in a Gazebo simulation.

Create a new file **empty_world.launch** in the **launch** folder and paste the content.

► Execute

```
In [ ]: touch ~/catkin_ws/src/robot_description/launch/empty_world.launch
```

📄 Copy and paste

```
In [ ]: <launch>
        <include file="$(find gazebo_ros)/launch/empty_world.launch">
          <arg name="paused" value="false"/>
          <arg name="use_sim_time" value="true"/>
          <arg name="gui" value="true"/>
          <arg name="headless" value="false"/>
          <arg name="debug" value="false"/>
        </include>
      </launch>
```

Start the simulation:

► Execute

```
In [ ]: roslaunch robot_description empty_world.launch
```

You currently have an empty world. Spawn the robot in a separate file. It is a better approach while you are developing because it allows you to remove the robot and re-spawn without restarting the whole simulation.

Create a new file **spawn.launch** and paste the content below:

```
In [ ]: <launch>
        <param name="robot_description" command="cat '$(find robot_description)/ur

        <arg name="x" default="0"/>
        <arg name="y" default="0"/>
        <arg name="z" default="0.5"/>

        <node name="mybot_spawn" pkg="gazebo_ros" type="spawn_model" output="screen"
              args="-urdf -param robot_description -model my_robot -x $(arg x) -y

      </launch>
```

You are allowed three arguments with the CLI: X, Y, and Z. And run this **gazebo_ros/spawn_model** node, which is in charge of converting the URDF model and inserting it into the running Gazebo simulation.

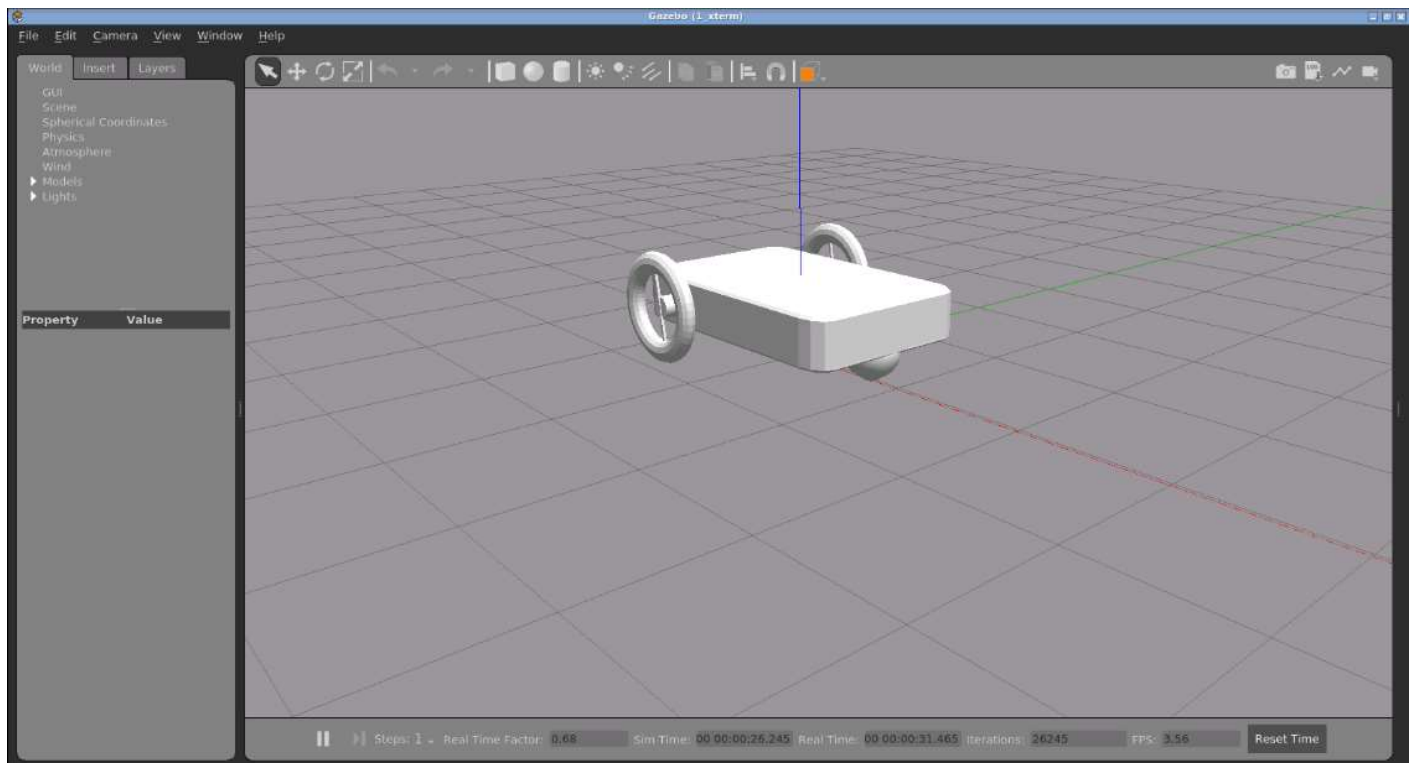
Spawn the robot:

► Execute

```
In [ ]: roslaunch robot_description spawn.launch
```



The result must be like the image below:



Gazebo is not applying the colors you set up! That is because you are not using special **<Gazebo>** tags to set it. You will learn how to do that in the next unit. Check the **World** tree, the robot model is there, and it contains all the links and joints defined in the URDF file, except for the fixed joint. For SDF, it is not necessary. It merges both links.



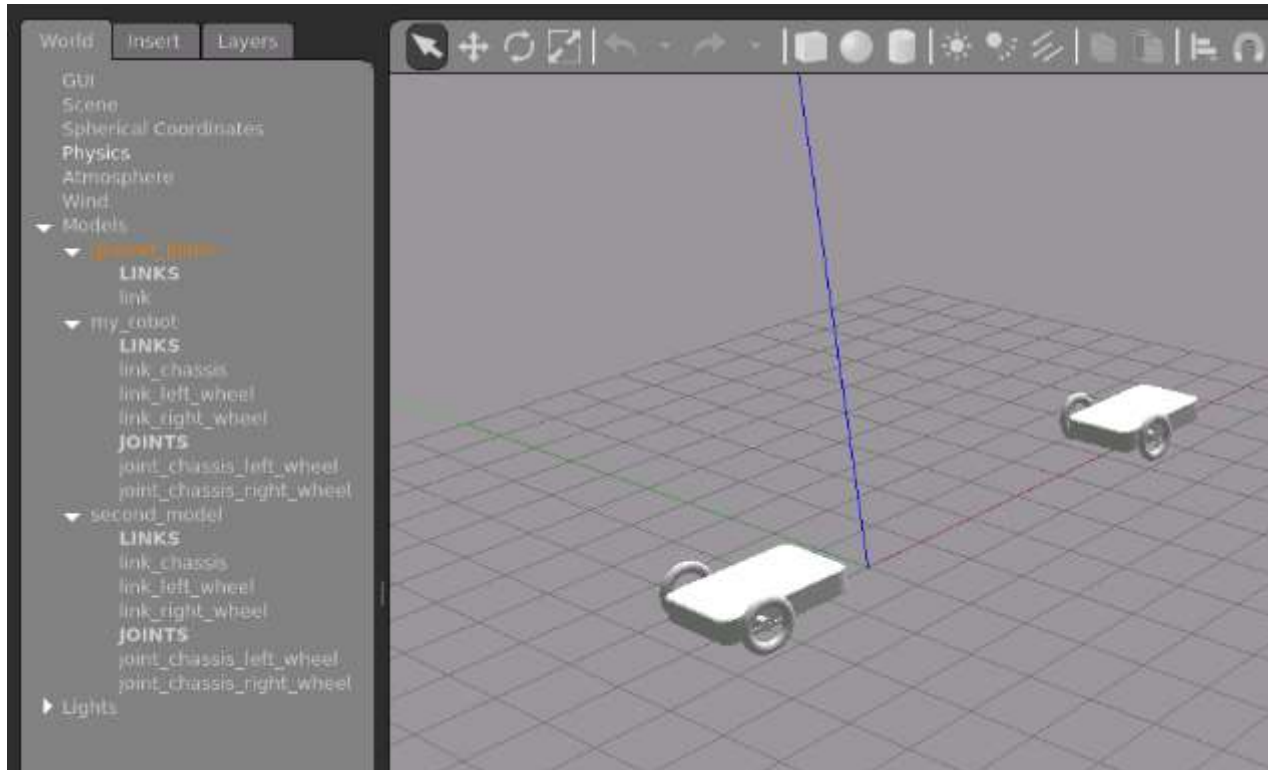
- Exercise 2.4 -

Add a new argument to the launch file that setup the name of the model to be inserted. It must replace the status **my_robot** name with the one provided in the CLI.

After that, spawn the robot you created twice using different names and check the world tree to ensure it worked correctly.

The expected result is to execute the command below and have the following:

```
roslaunch robot_description spawn.launch x:=6 model_name:=second_model
```



- End of Exercise 2.4 -

- Solution for Exercise 2.4 -

Do not check the solution before trying it on your own! Instead, review the solution when you have completed the exercise. It is a good resource to compare your solution.

If you have problems completing the exercise, use our forum support!

[spawn.launch \(unit-02/spawn-ex2.4.launch.txt\)](#)

- End of Solution for Exercise 2.4 -

2.5 Applying Force/Torque

Applying force/torque

There is a feature in Gazebo that allows the application of external force or torque to a specific link of a model. It is a good way to test links and joints.

Prepare the empty world with a single robot.

► Execute in Shell #1

```
In [ ]: roslaunch robot_description empty_world.launch
```

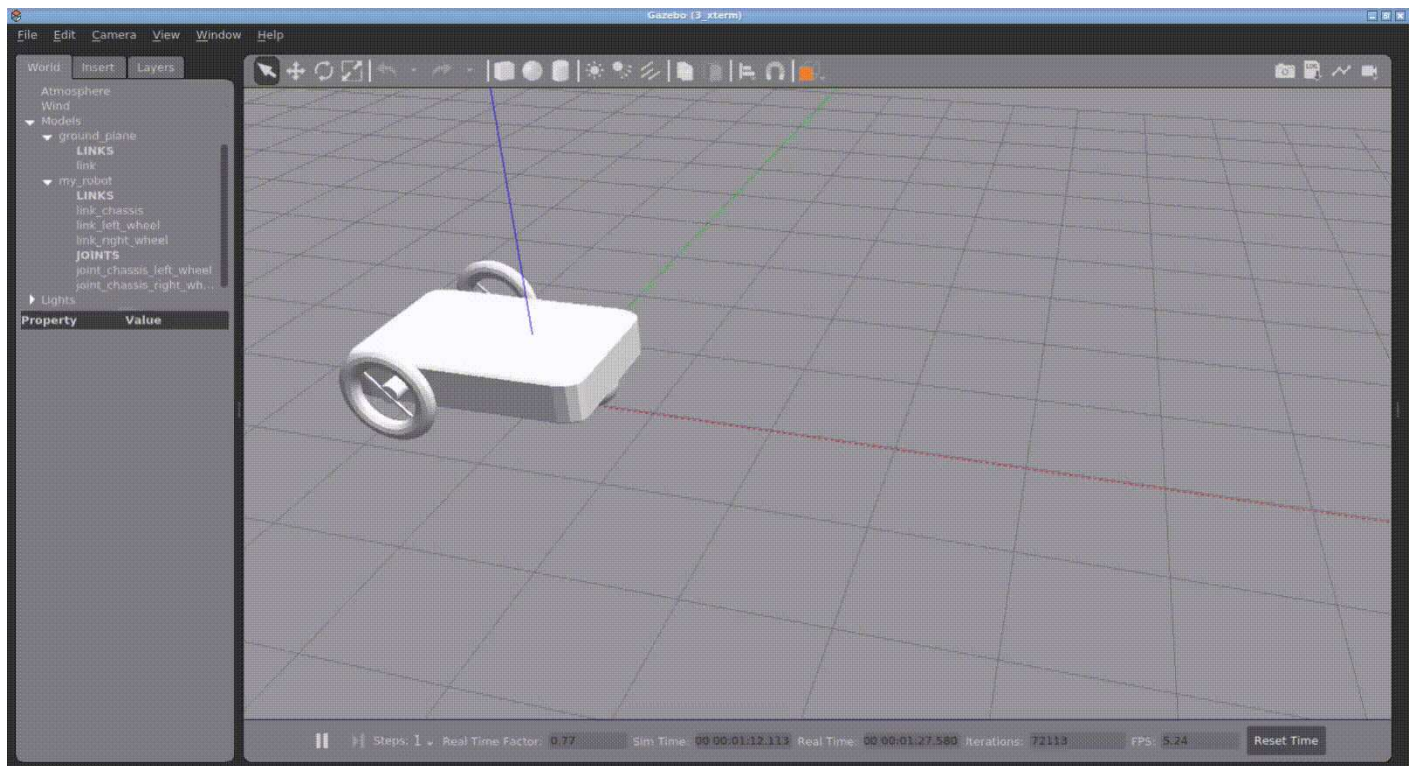


► Execute in Shell #2

```
In [ ]: roslaunch robot_description spawn.launch
```

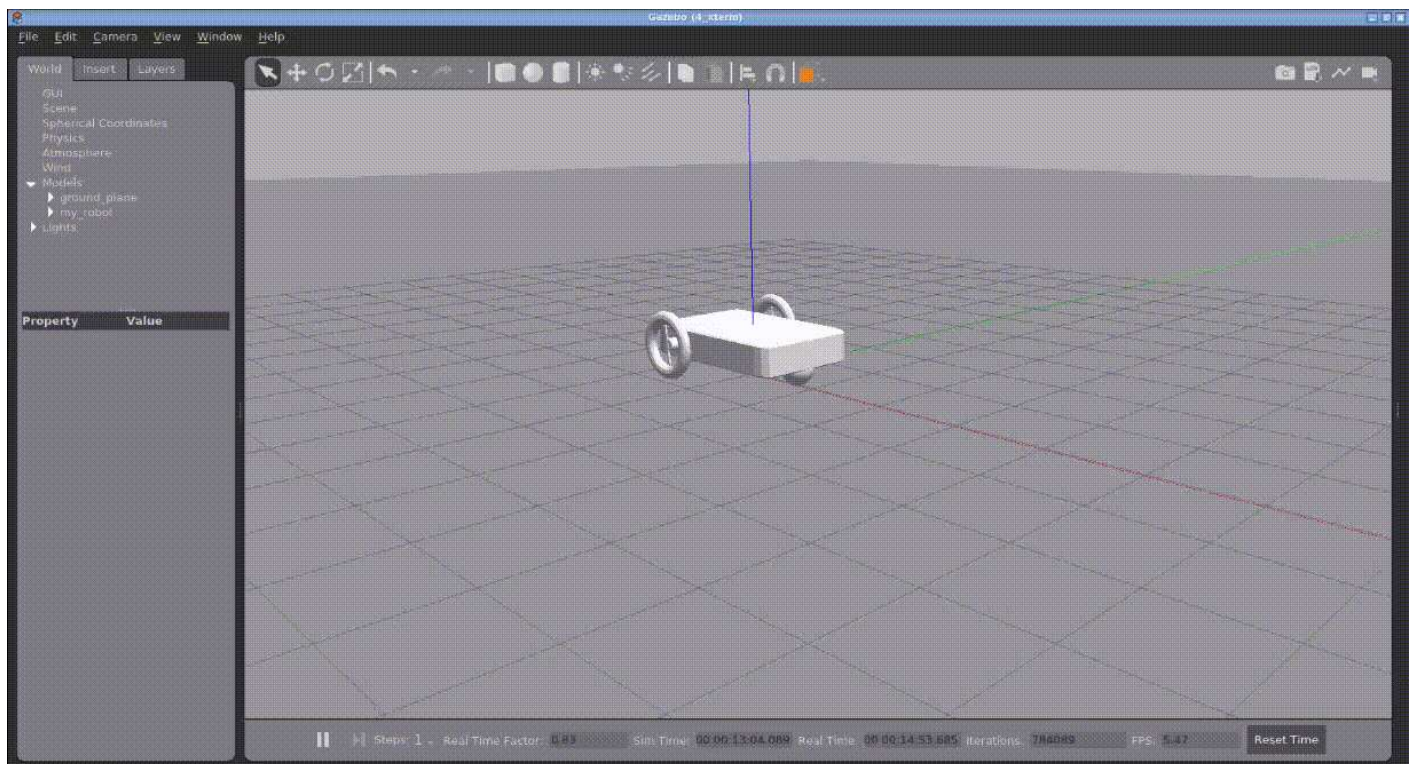


With Gazebo open, click the desired link to select and click the link again in the 3D viewer with the right button:



Monitoring model response

It is also possible to monitor the response of the model body. It's really helpful to debug the model inertia, for example. Check the example below:



- Practice 2.5 -

Use the same interface to apply **Torque** to the robot. Use the **Plot** window to monitor the angular acceleration of the model to understand the relation and how the inertia of the model is related to that.

Note: There is no solution for this exercise, so use the exercise to practice what you have learned.

- End of Practice 2.5 -