Fall 2015

ROS - Lesson 9

Teaching Assistant: Roi Yehoshua
roiyeho@gmail.com

Agenda

- Create a 3D model of our robot
- URDF and Xacro files
- Joint State Publisher
- Watching the 3D model on rviz

URDF

- **Unified Robot Description Format** (**URDF**) is an XML format for representing a robot model
- In this file you define:
  - Kinematic and dynamic description of the robot
  - Visual representation of the robot
  - Collision model of the robot
- A list of robots described by URDF files can be found here: http://wiki.ros.org/urdf/Examples

URDF Format

- The description of a robot consists of a set of link (part) elements, and a set of joint elements connecting the links together
- A typical robot description looks like this:

```
<robot name="pr2">
  <link> ... </link>
  <link> ... </link>
  <link> ... </link>

  <joint>  ....  </joint>
  <joint>  ....  </joint>
  <joint>  ....  </joint>
</robot>
```

URDF Turorials

- We’re going to build a visual model of a robot that vaguely looks like R2D2
- All of the robot models and launch files mentioned here can be found in the urdf_tutorial package (already installed with ROS)

One Shape

- First, we're just going to create one simple shape
- Look at 01-myfirst.urdf:

```
<?xml version="1.0"?>
<robot name="myfirst">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
    </visual>
  </link>
</robot>
```

- This creates a robot with the name myfirst, that contains only one link, whose visual component is just a cylinder 0.6 meters long with a 0.2 meter radius.

Watching the 3D Model in rviz

display.launch (in urdf_tutorial package)

```
<launch>
    <arg name="model" />
    <arg name="gui" default="False" />
    <param name="robot_description" textfile="$(arg model)" />
    <param name="use_gui" value="$(arg gui)"/>
    <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher"></node>
    <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />
    <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
urdf_tutorial)/urdf.rviz" />
</launch>
```

Watching the 3D Model in rviz

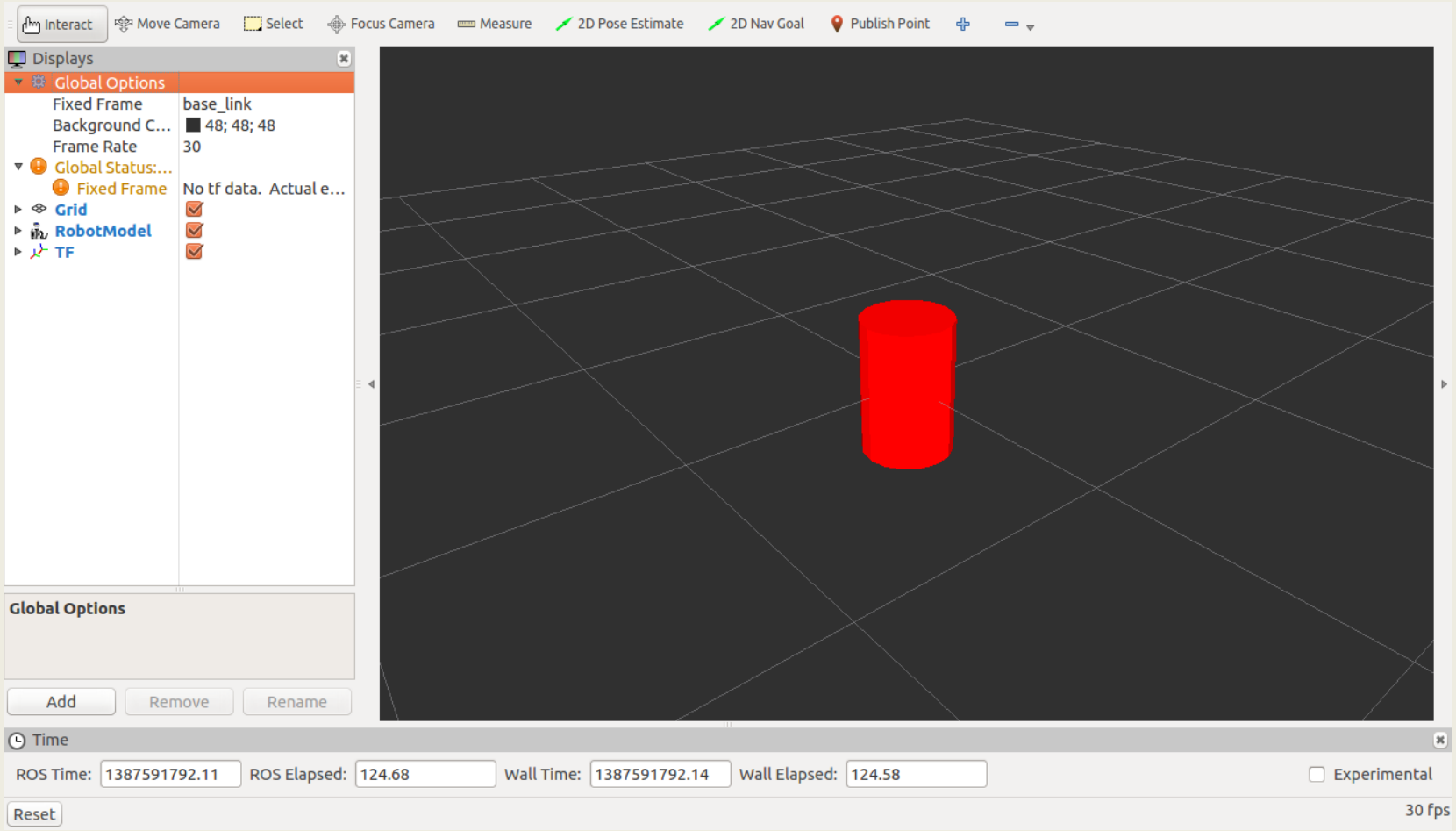
- This launch file does three things:
  - Loads the specified model into the parameter server
  - Runs nodes to publish the JointState and transforms
  - Starts rviz with a configuration file
- To run the launch file type
  - If you are in the same directory where the urdf file is:

```
$ roslaunch urdf_tutorial display.launch model:=01-myfirst.urdf
```

  - If you are in a different directory:

```
$ roslaunch urdf_tutorial display.launch model:='$(find
urdf_tutorial)/urdf/01-myfirst.urdf'
```

Watching the 3D Model in rviz



Multiple Links

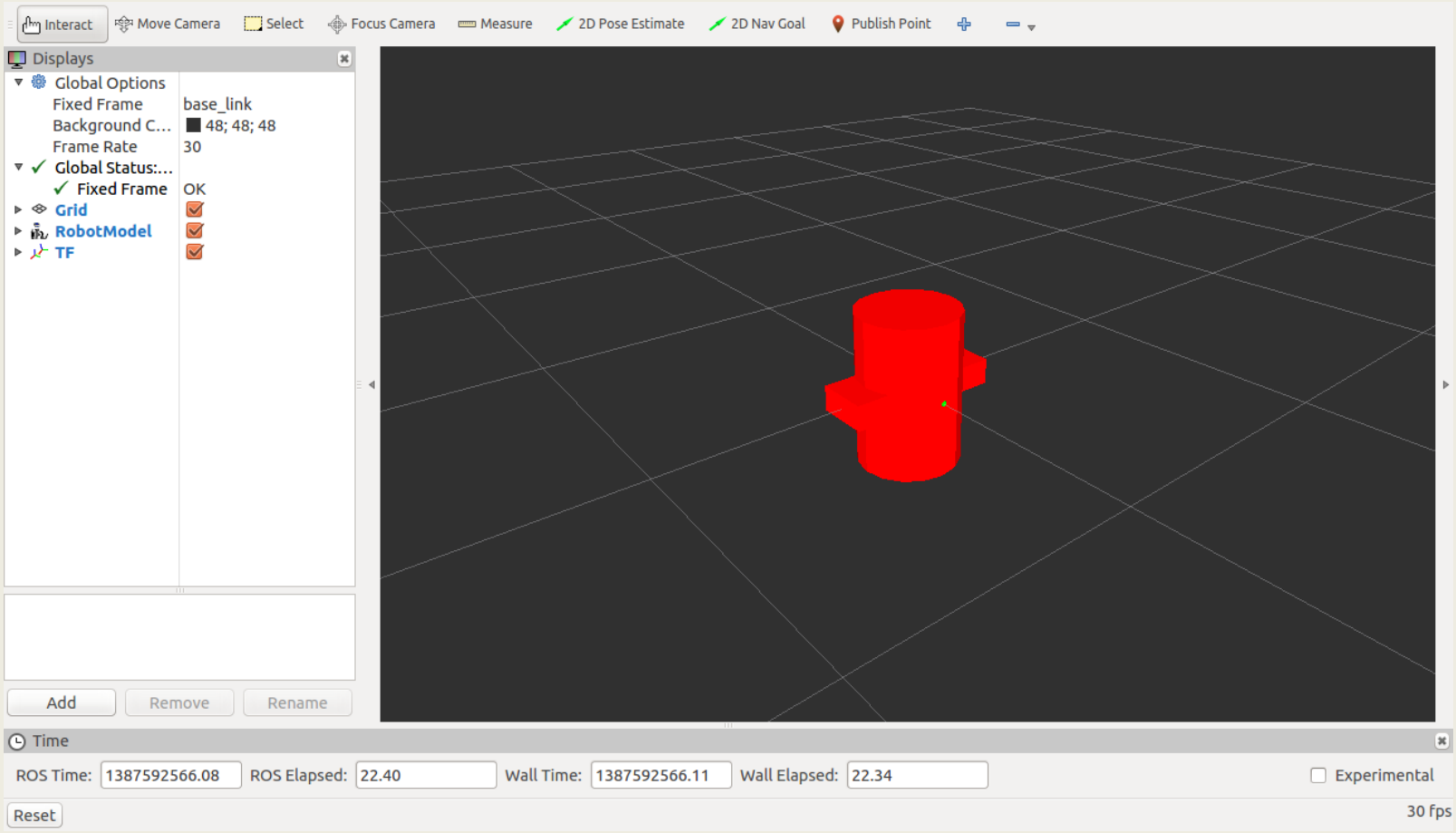
- In order to add more link elements to the urdf, you need to define joints
- The joint is connects a parent and a child link
- There are different types of joints
- We'll start with fixed joints

02-multipleshapes.urdf

```
<?xml version="1.0"?>
<robot name="multipleshapes">
 <link name="base_link">
  <visual>
   <geometry>
    <cylinder length="0.6" radius="0.2"/>
   </geometry>
  </visual>
 </link>

 <link name="right_leg">
  <visual>
   <geometry>
    <box size="0.6 .2 .1"/>
   </geometry>
  </visual>
 </link>

 <joint name="base_to_right_leg" type="fixed">
  <parent link="base_link"/>
  <child link="right_leg"/>
 </joint>
</robot>
```

Multiple Links

```
$ roslaunch urdf_tutorial display.launch model:=02-multipleshapes.urdf
```



Origins

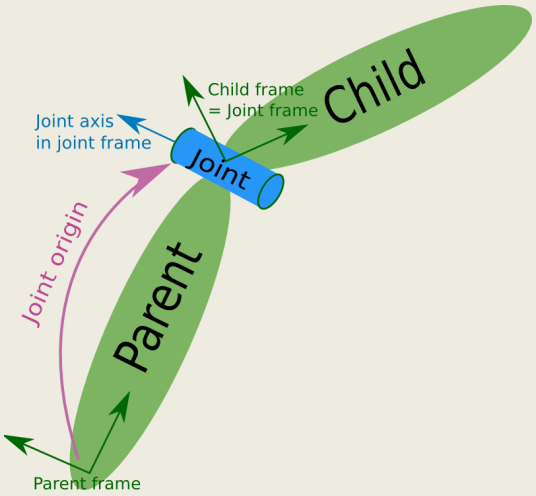
- Both of the shapes overlap with each other, because they share the same origin
- If we want them not to overlap we must define more origins
- Both the link and the joint elements may contain origin tags

Link's Origin Tag

- **<origin>** - defines where the center of the visual element should be relative to its origin
  - *defaults to identity if not specified*
- **Properties:**
  - **xyz –** represents the offset
    - *defaults to zero vector*
  - **rpy** - represents the rotation (roll, pitch, yaw)
    - *defaults to identity*

Joint's Origin Tag

- **<origin>** - This is the transform from the parent link to the child link. The joint is located at the origin of the child link
  - *– defaults to identity if not specified)*
- Has the same properties like the link's origin tag (xyz and rpy)



03-origins.urdf

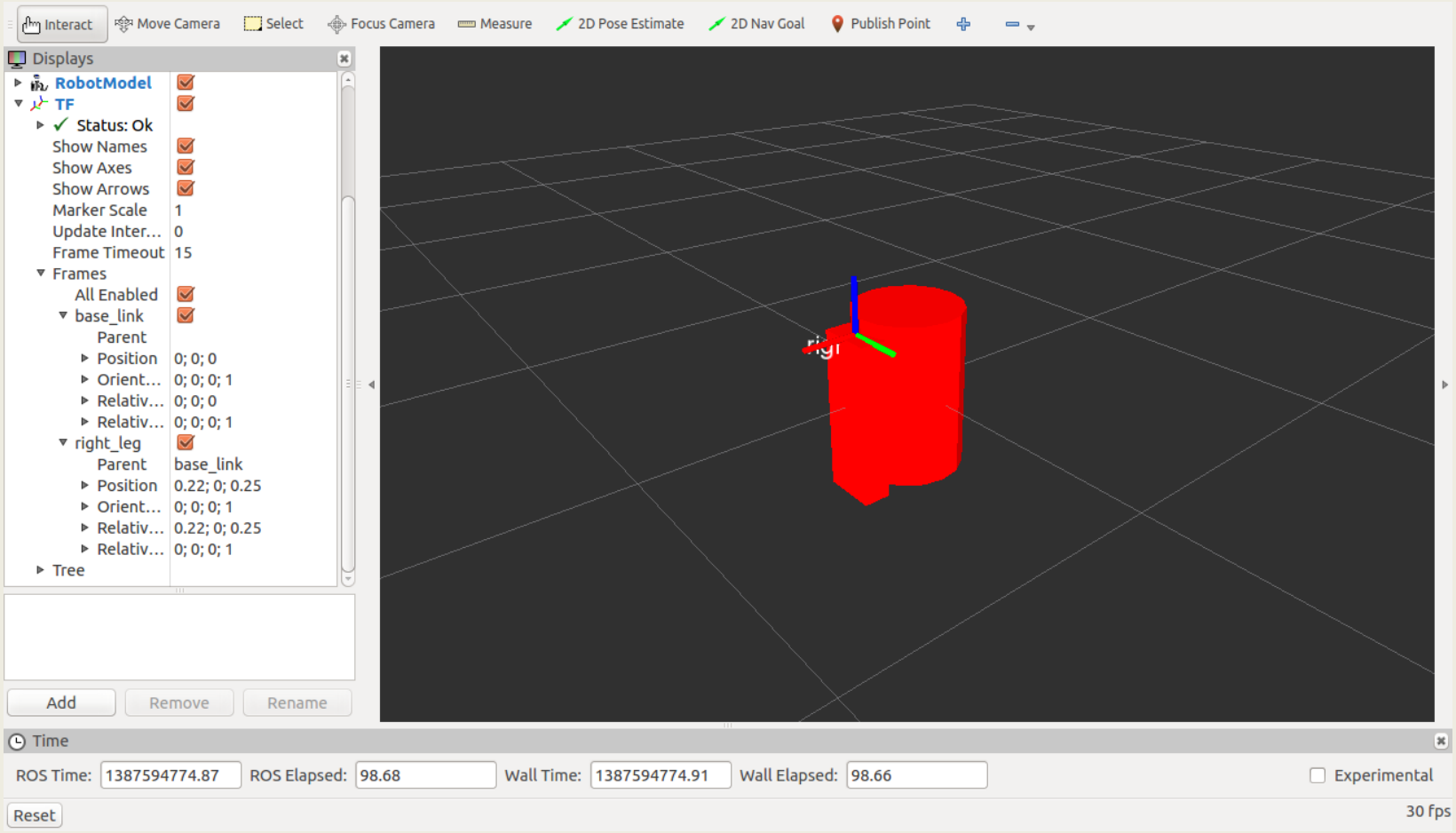
```
<?xml version="1.0"?>
<robot name="origins">
 <link name="base_link">
  <visual>
   <geometry>
    <cylinder length="0.6" radius="0.2"/>
   </geometry>
  </visual>
 </link>
 <link name="right_leg">
  <visual>
   <geometry>
    <box size="0.6 .2 .1"/>
   </geometry>
   <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
  </visual>
 </link>
 <joint name="base_to_right_leg" type="fixed">
  <parent link="base_link"/>
  <child link="right_leg"/>
  <origin xyz="0.22 0 .25"/>
 </joint>
</robot>
```

Origins

- The joint's origin is defined in terms of the parent's reference frame.
  - In this case the origin for the child link will be .22 meters in the x direction (right) and .25 meters in the z direction (up)
- The leg's visual origin defines where the center of the visual element should be relative to its origin
  - Here we want the leg to attach at the top, so we offset the origin down by setting the z offset to be -.3 meters
  - And since we want the long part of the leg to be parallel to the z axis, we rotate the visual part PI/2 around the Y axis

Examining the Model

```
$ roslaunch urdf_tutorial display.launch model:=03-origins.urdf
```



Material Girl

- That's very cute, but not all robots are red…
- The material tag of the visual element allows you to specify the material used by the link element
- Properties:
  - **name** – name of the material. Can refer to a previously defined material.
  - **<color>** *(optional)*
    - **rgba** The color of a material specified by set of four numbers representing red/green/blue/alpha, each in the range of [0,1].
  - **<texture>** *(optional)*
    - The texture of a material is specified by a **filename**

04-materials.urdf (1)

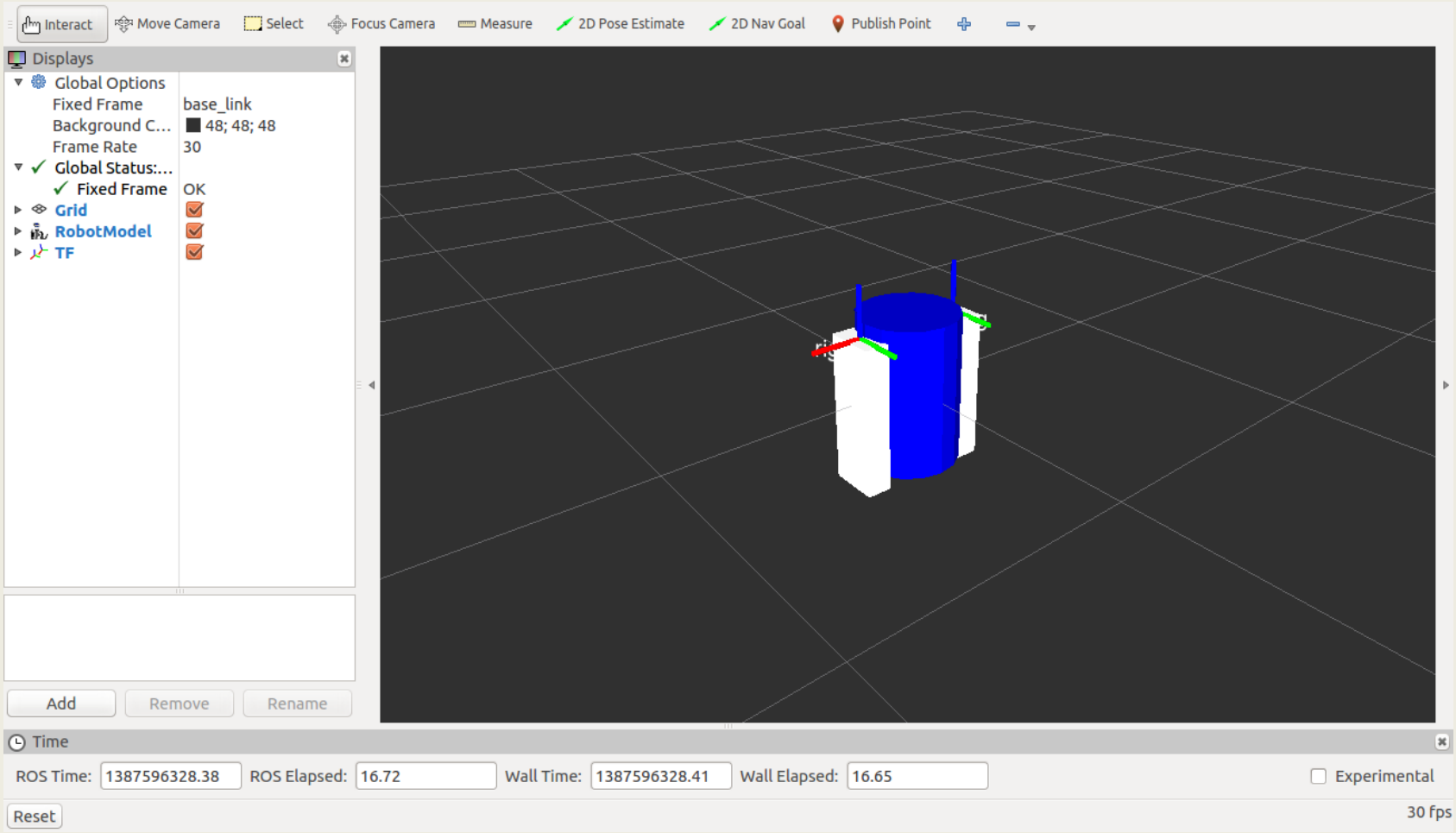
```
<?xml version="1.0"?>
<robot name="materials">
 <link name="base_link">
  <visual>
   <geometry>
    <cylinder length="0.6" radius="0.2"/>
   </geometry>
   <material name="blue">
    <color rgba="0 0 .8 1"/>
   </material>
  </visual>
 </link>

 <link name="right_leg">
  <visual>
   <geometry>
    <box size="0.6 .2 .1"/>
   </geometry>
   <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
   <material name="white">
    <color rgba="1 1 1 1"/>
   </material>
  </visual>
 </link>
```

04-materials.urdf (2)

```
 <joint name="base_to_right_leg" type="fixed">
  <parent link="base_link"/>
  <child link="right_leg"/>
  <origin xyz="0.22 0 .25"/>
 </joint>

 <link name="left_leg">
  <visual>
   <geometry>
    <box size="0.6 .2 .1"/>
   </geometry>
   <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
   <material name="white"/>
  </visual>
 </link>

 <joint name="base_to_left_leg" type="fixed">
  <parent link="base_link"/>
  <child link="left_leg"/>
  <origin xyz="-0.22 0 .25"/>
 </joint>

</robot>
```

Examining the Model

```
$ roslaunch urdf_tutorial display.launch model:=04-materials.urdf
```



Loading Meshes to Our Model

- Mesh files allow you to define more realistic elements than just using basic geometric objects/blocks
- It is possible to load meshes generated by us or to use meshes of other models.
- For our model we will use the PR2's gripper

Loading Meshes to Our Model

```
<link name="left_gripper">
  <visual>
    <origin rpy="0.0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://pr2_description/meshes/gripper_v0/l_finger.dae"/>
    </geometry>
  </visual>
</link>
```

pr2_description

- To borrow the meshes from PR2, you'll need to install the pr2_description package:

```
$ sudo apt-get install ros-hydro-pr2-description
```

- The package contains robot description files for PR2, organized into subdirectories as follows:
  - urdf/ contains urdf descriptions of various parts of the PR2 (arm, torso, etc.)
  - robots/ contains urdf descriptions of the full robot, that refer to the macros in urdf/
  - gazebo/ contains urdf descriptions of simulated PR2 components, like the simulated battery controller
  - meshes/ contains mesh files (.stl,.dae) for visualization and collision properties

Example of a Collada File

*l_finger.dae [Read-Only] (/opt/ros/hydro/share/pr2_description/meshes/gripper_v0) - gedit

Open Save Undo

*l_finger.dae

```
    <created>2010-04-30T15:44:27</created>
    <modified>2010-04-30T15:44:27</modified>
    <unit name="meter" meter="1"/>
    <up_axis>Y_UP</up_axis>
  </asset>
  <library_materials>
    <material id="lambert3" name="lambert3">
      <instance_effect url="#lambert3-fx"/>
    </material>
  </library_materials>
  <library_effects>
    <effect id="lambert3-fx">
      <profile_COMMON>
        <newparam sid="file3-surface">
          <surface type="2D">
            <init_from>file3</init_from>
          </surface>
        </newparam>
        <newparam sid="file3-sampler">
          <sampler2D>
            <source>file3-surface</source>
          </sampler2D>
        </newparam>
        <newparam sid="file2-surface">
          <surface type="2D">
            <init_from>file2</init_from>
          </surface>
        </newparam>
        <newparam sid="file2-sampler">
          <sampler2D>
            <source>file2-surface</source>
          </sampler2D>
```

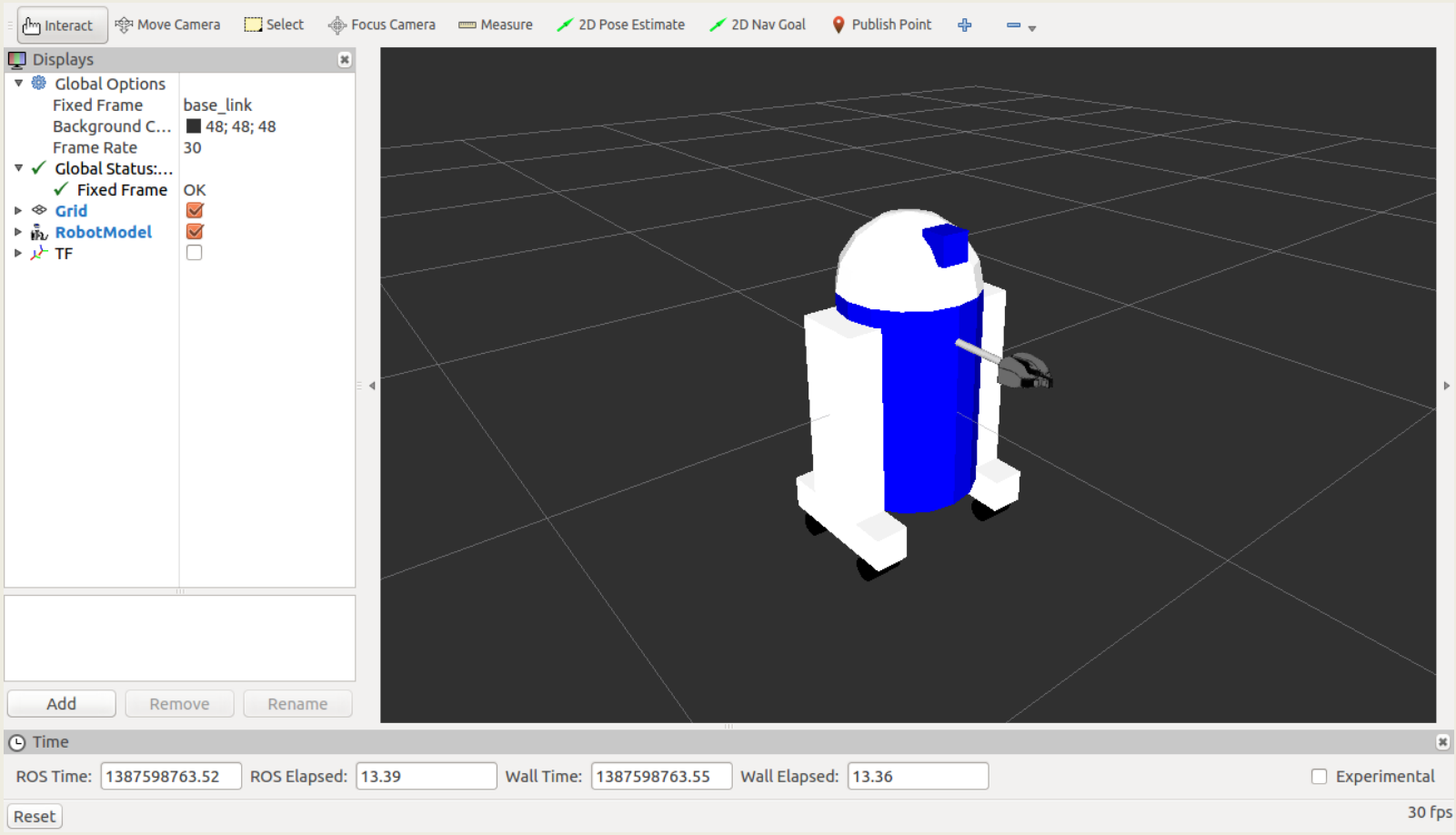
XML Tab Width: 8 Ln 11, Col 102 INS

Finishing the Model

- Let's finish the design by adding some parts: left and right bases, four wheels and an arm with a gripper
- The final urdf file is located at 05-visual.urdf

Finishing the Model

```
$ roslaunch urdf_tutorial display.launch model:=05-visual.urdf
```



Building a Movable Robot Model

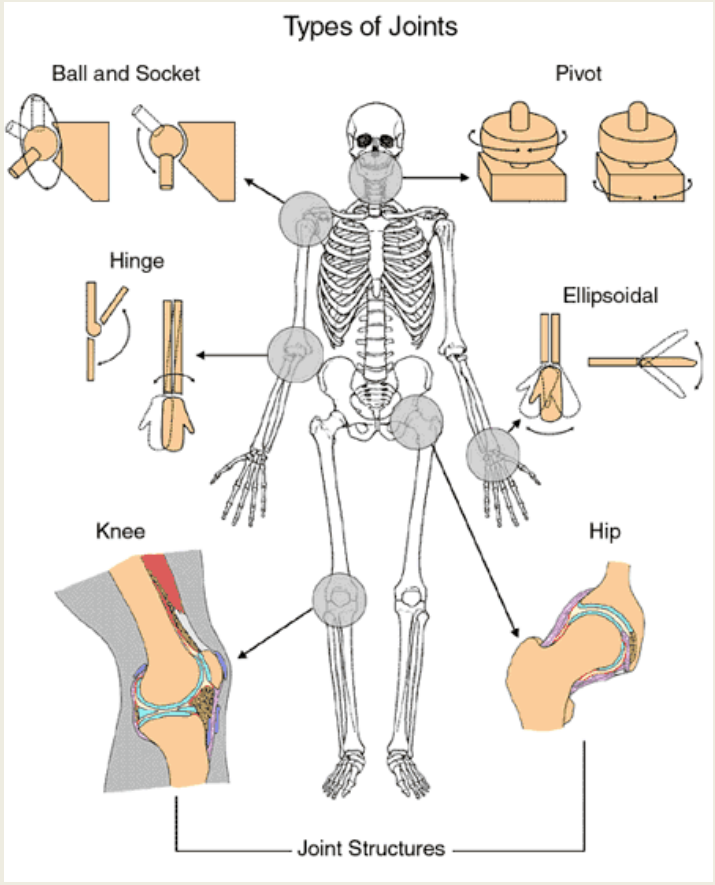
- To convert the model into a robot that can actually move, the only thing you have to do is take care of the type of the joints it uses
- In the previous model, all of the joints were fixed
- Now we are going to explore different types of flexible joints

Joint Types

- **revolute** - a hinge joint that rotates along the axis and has a limited range specified by the upper and lower limits
- **continuous** - a continuous hinge joint that rotates around the axis and has no upper and lower limits
- **prismatic** - a sliding joint that slides along the axis, and has a limited range specified by the upper and lower limits
- **fixed** - This is not really a joint because it cannot move
- **floating** - This joint allows motion for all 6 degrees of freedom
- **planar** - This joint allows motion in a plane perpendicular to the axis

Joint Types

- Compare to joint types in the human body:



The Head

```
<joint name="head_swivel" type="continuous">
   <parent link="base_link"/>
   <child link="head"/>
   <axis xyz="0 0 1"/>
   <origin xyz="0 0 0.3"/>
</joint>
```

- The connection between the body and the head is a continuous joint, meaning that it can take on any angle from -∞ to +∞
- The only additional information we have to add is the axis of rotation (the <axis> tag), which specifies a vector around which the head will rotate.
  - the vector "0 0 1" means it will rotate around the z axis

The Wheels

```
<joint name="right_front_wheel_joint" type="continuous">
   <axis xyz="0 0 1"/>
   <parent link="right_base"/>
   <child link="right_front_wheel"/>
   <origin rpy="0 1.57075 0" xyz="0 0.133333333333 -0.085"/>
   <limit effort="100" velocity="100"/>
   <dynamics damping="0.0" friction="0.0"/>
</joint>
```

- The wheels are also modeled as continuous joints, so that they can roll in both directions forever

The Gripper

```
<joint name="left_gripper_joint" type="revolute">
   <axis xyz="0 0 1"/>
   <limit effort="1000.0" lower="0.0" upper="0.548" velocity="0.5"/>
   <origin rpy="0 0 0" xyz="0.2 0.01 0"/>
   <parent link="gripper_pole"/>
   <child link="left_gripper"/>
</joint>
```

- The right and the left gripper joints are modeled as revolute joints
- This means that they rotate in the same way that the continuous joints do, but they have strict limits
- Hence, we must include the <limit> tag specifying the upper and lower limits of the joint (in radians)
- We also must specify a maximum velocity and effort for this joint

The Gripper Arm

```
<joint name="gripper_extension" type="prismatic">
   <parent link="base_link"/>
   <child link="gripper_pole"/>
   <limit effort="1000.0" lower="-0.38" upper="0" velocity="0.5"/>
   <origin rpy="0 0 1.57075" xyz="0 0.19 .2"/>
</joint>
```

- The gripper arm is modeled as a prismatic joint
  - This means that it moves along an axis, not around it
- This translational movement is what allows our robot model to extend and retract its gripper arm
- The limits of the prismatic arm are specified in the same way as a revolute joint, except that the units are meters, not radians

Building a Movable Robot Model

- 06_flexible.urdf is the new urdf with flexible joints

Joint State Publisher

- http://wiki.ros.org/joint_state_publisher
- This package contains a tool for setting and publishing joint state values for a given URDF
- Publishes sensor_msgs/JointState messages for all the non-fixed joints of the robot
- Can be used in conjunction with the robot_state_publisher node to also publish transforms for all joint states

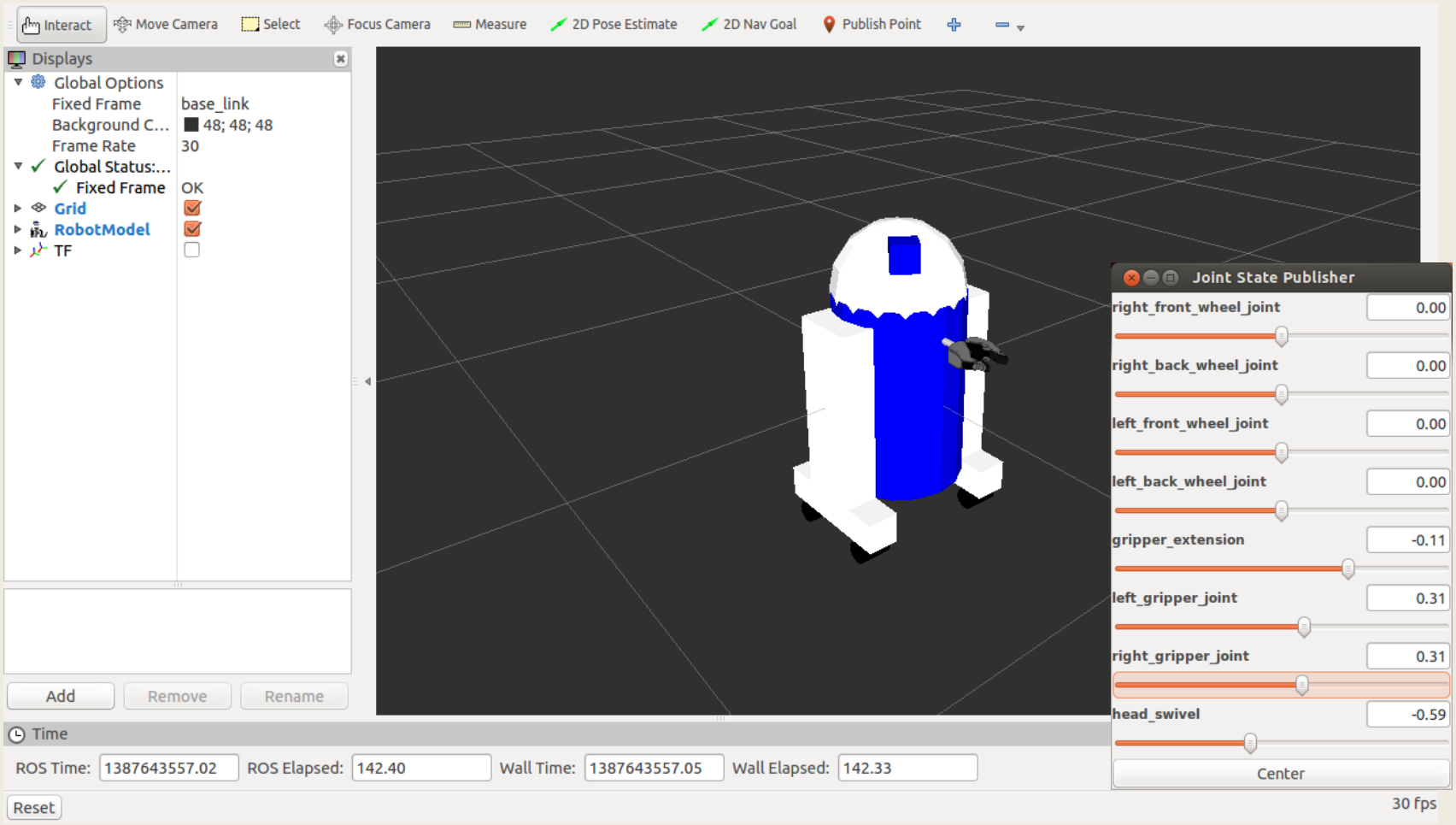
Joint State Publisher GUI

- To visualize and control this model, type:

```
$ roslaunch urdf_tutorial display.launch model:=06-flexible.urdf gui:=True
```

- This will also pop up a GUI that allows you to control the values of all the non-fixed joints
- Play with the model some and see how it moves

Joint State Publisher GUI



Adding Physical and Collision Properties

- If you want to simulate the robot on Gazebo or any other simulation software, it is necessary to add physical and collision properties
- We need to set on every link the dimension of the geometry to calculate the possible collisions, the weight that will give us the inertia, and so on.
- 07-physics.urdf is the new URDF with the physical properties

Collision

- The collision element is a direct subelement of the link object, at the same level as the visual tag
- The collision element defines its shape the same way the visual element does, with a geometry tag
- The format for the geometry tag is exactly the same here as with the visual.
- You can also specify an origin in the same way as a subelement of the collision tag (as with the visual)

Collision Element Example

```
<link name="base_link">
   <visual>
    <geometry>
     <cylinder length="0.6" radius="0.2"/>
    </geometry>
    <material name="blue">
     <color rgba="0 0 .8 1"/>
    </material>
   </visual>
   <collision>
    <geometry>
     <cylinder length="0.6" radius="0.2"/>
    </geometry>
   </collision>
</link>
```

Collision

- Usually the collision geometry is defined exactly the same as the visual geometry
- There are two main cases where you wouldn't:
  - Quicker Processing - Doing collision detection for two meshes is a lot more computational complex than for two simple geometries. Hence, you may want to replace the meshes with simpler geometries in the collision element.
  - Safe Zones - You may want to restrict movement close to sensitive equipment. For instance, if we didn't want anything to collide with R2D2's head, we might define the collision geometry to be a cylinder encasing his head to prevent anything from getting to near his head.

Inertia

- Inertia is the resistance of any physical object to any change in its state of motion, including changes to its speed and direction
- It is the tendency of objects to keep moving in a straight line at constant velocity
- The inertia tensor depends on both the mass and the distribution of mass of the object
  - A good explanation on intertia tensor can be found here
- The 3x3 rotational inertia matrix is specified with the inertia element

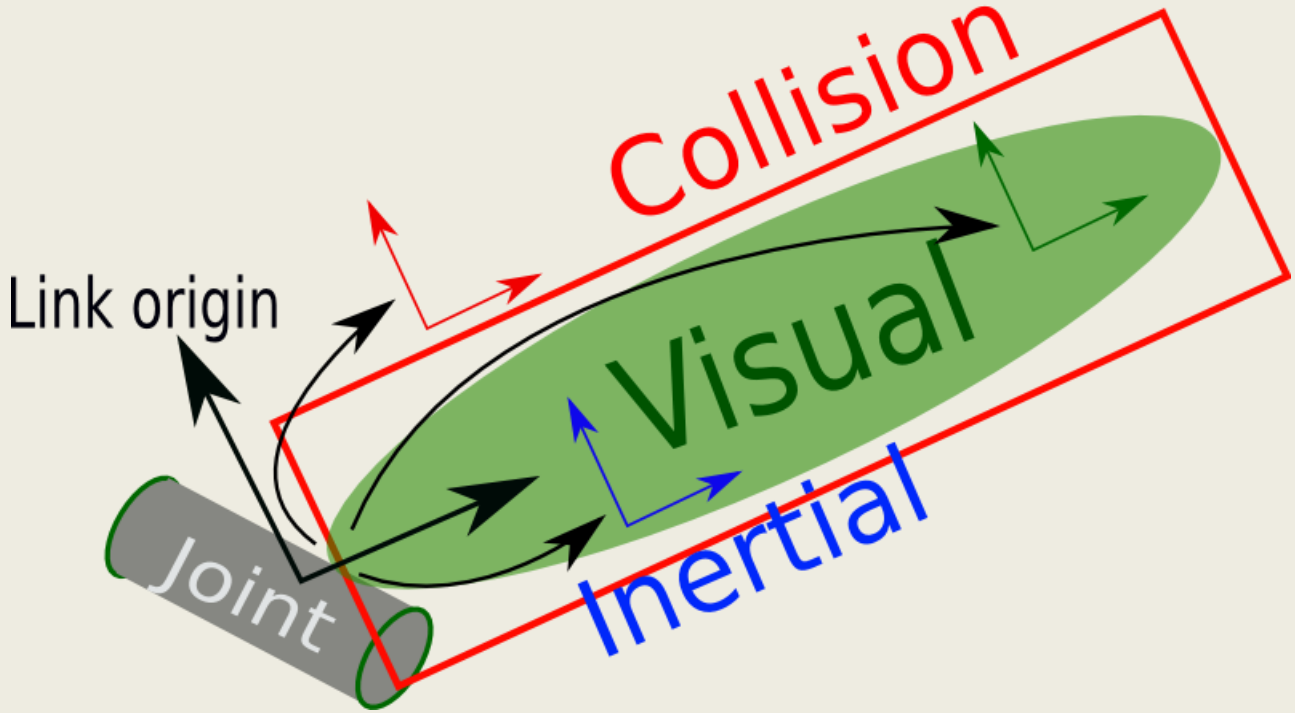
ixx	**ixy**	**ixz**
ixy	**iyy**	**iyz**
ixz	iyz	**izz**

Inertia Example

```
<link name="base_link">
  <visual>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
    <material name="blue">
      <color rgba="0 0 .8 1"/>
    </material>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="10"/>
    <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
  </inertial>
 </link>
```

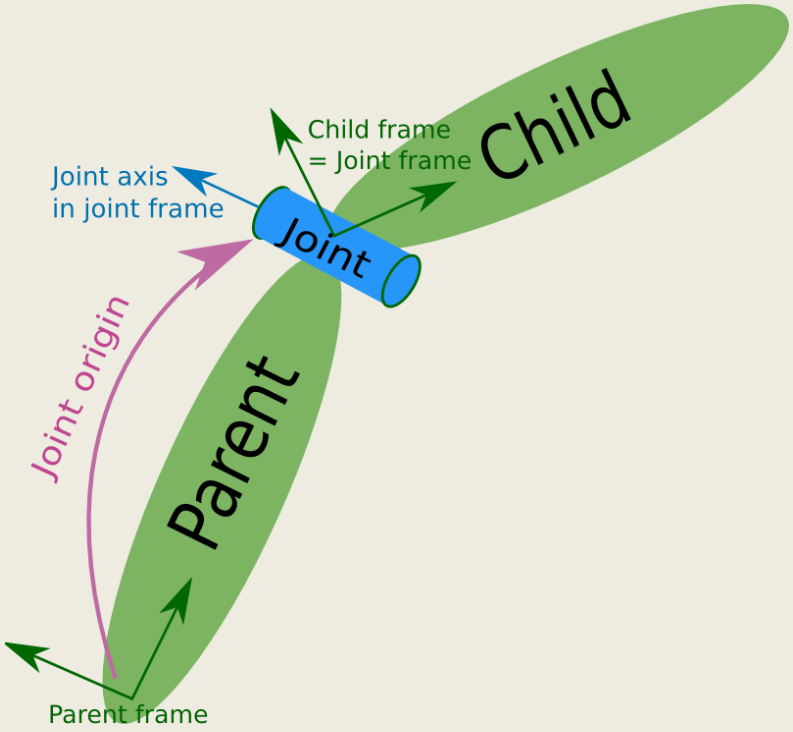
Link Element Summary

- The link element describes a rigid body with an inertia, visual and collision features



Joint Element Summary

- The joint element describes the kinematics and dynamics of the joint and also specifies the safety limits of the joint



Xacro

- Notice the size of the 07-physics.urdf file
- It has 414 lines of code to define our robot (!)
- Imagine if you start to add cameras and other geometries, the file will start to increase and its maintenance will become even more complicated
- Xacro helps to reduce the overall size of the URDF file and makes it easier to read and maintain.
- It also allows us to create modules and reuse them to create repeated structures such as several arms or legs

Using Xacro

- As its name implies, xacro is a macro language.
- The xacro program runs all of the macros and outputs the result as a urdf file
- Typical usage:

```
$ rosrun xacro xacro.py model.xacro > model.urdf
```

- You can also automatically generate the urdf in a launch file:

```
<param name="robot_description" command="$(find xacro)/xacro.py
'$(find pr2_description)/robots/pr2.urdf.xacro'" />
```

Using Xacro

- In the xacro file, you must specify a namespace in order for the file to parse properly.
- For example, these are the first two lines of a valid xacro file:

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="robot1_xacro">
```

Constants

- We can use xacro to declare constant values
- This way we can avoid putting the same value in a lot of lines and will make it easier to maintain changes.
- You can define constants using <xacro:property> tag, usually located at the top of the file
- Then to use the constant's value, you can write ${name_of_variable}

Constants Example

```
<xacro:property name="width" value=".2" />
<xacro:property name="bodylen" value=".6" />
<link name="base_link">
    <visual>
        <geometry>
            <cylinder radius="${width}" length="${bodylen}"/>
        </geometry>
        <material name="blue">
            <color rgba="0 0 .8 1"/>
        </material>
    </visual>
    <collision>
        <geometry>
            <cylinder radius="${width}" length="${bodylen}"/>
        </geometry>
    </collision>
</link>
```

Math

- You can build up arbitrarily complex expressions in the ${} construct using the four basic operations (+,-,*,/), the unary minus, and parenthesis.
- Exponentiation and modulus are not supported.
- Examples:

```
<cylinder radius="${wheeldiam/2}" length=".1"/>
<origin xyz="${reflect*(width+.02)} 0 .25" />
```

Macros

- Macros are the most useful component of the xacro package
- Example for a simple macro:

```
<xacro:macro name="default_origin">
      <origin xyz="0 0 0" rpy="0 0 0"/>
</xacro:macro>
<xacro:default_origin />
```

- This code will generate the following:

```
<origin xyz="0 0 0" rpy="0 0 0"/>
```

Parameterized Macros

- You can also parameterize macros so that they don't generate the same exact text every time
- When combined with the math functionality, this is even more powerful.
- An inertia macro example:

```
<xacro:macro name="default_inertial" params="mass">
    <inertial>
        <mass value="${mass}" />
        <inertia ixx="1.0" ixy="0.0" ixz="0.0"
                 iyy="1.0" iyz="0.0"
                 izz="1.0" />
    </inertial>
</xacro:macro>
<xacro:default_inertial mass="10"/>
```

Example of a Leg Macro

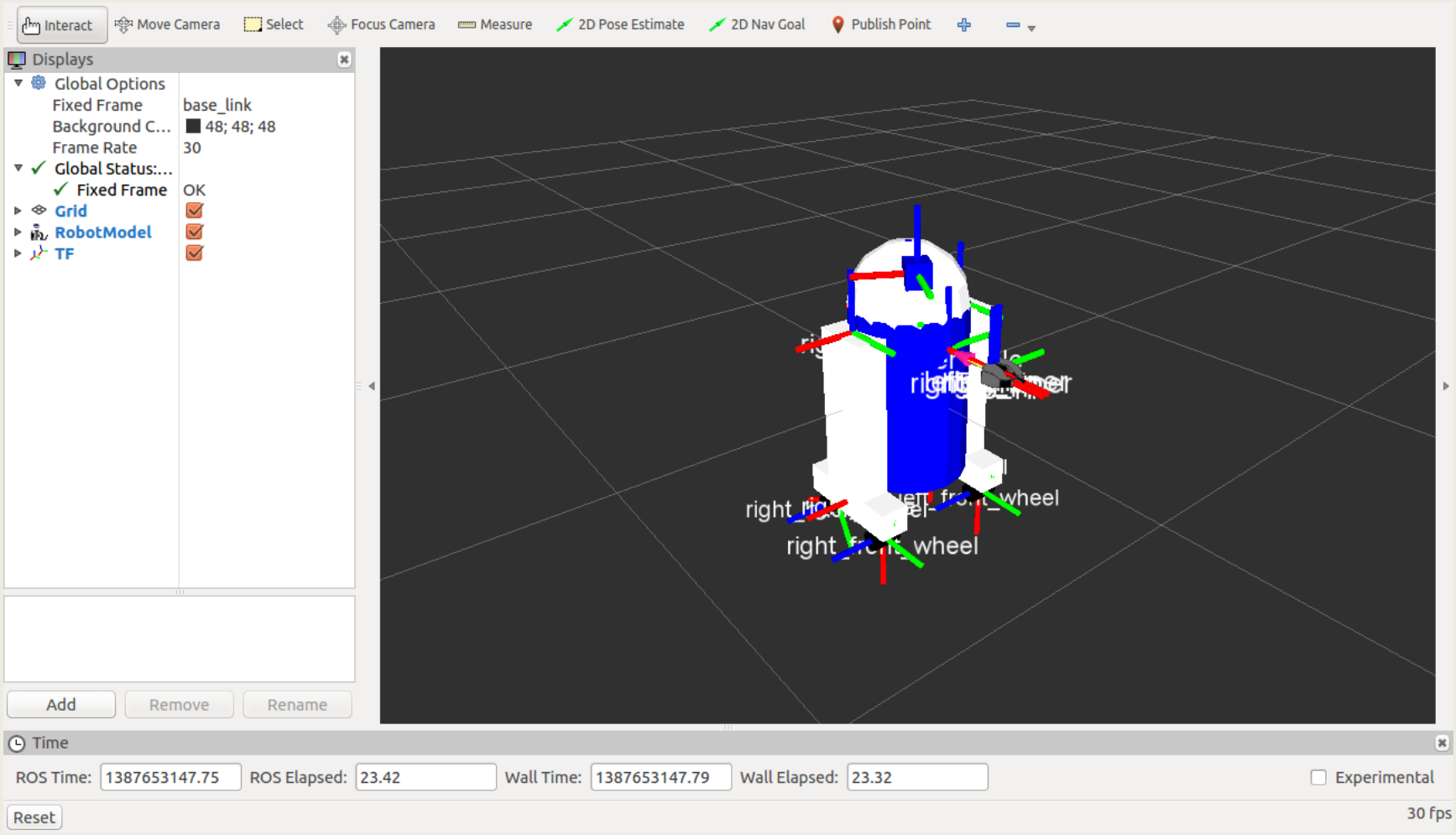
```
<xacro:macro name="leg" params="prefix reflect">
    <link name="${prefix}_leg">
        <visual>
            <geometry>
                <box size="${leglen} .2 .1"/>
            </geometry>
            <origin xyz="0 0 -${leglen/2}" rpy="0 ${pi/2} 0"/>
            <material name="white">
                <color rgba="1 1 1 1"/>
            </material>
        </visual>
        <collision>
            <geometry>
                <box size="${leglen} .2 .1"/>
            </geometry>
            <origin xyz="0 0 -${leglen/2}" rpy="0 ${pi/2} 0"/>
        </collision>
        <xacro:default_inertial mass="10"/>
    </link>
    <joint name="base_to_${prefix}_leg" type="fixed">
        <parent link="base_link"/>
        <child link="${prefix}_leg"/>
        <origin xyz="${reflect*(width+.02)} 0 .25" />
    </joint>
    <!-- A bunch of stuff cut -->
</xacro:macro>
<xacro:leg prefix="right" reflect="1" />
<xacro:leg prefix="left" reflect="-1" />
```

Examining the Xacro

- Here is the xacro used in the R2D2 model
- Number of lines was reduced from 414 to 236
- To see the model generated by the xacro file, run:

```
$ roslaunch urdf_tutorial xacrodisplay.launch model:=08-macroed.urdf.xacro
```

Examining the Xacro



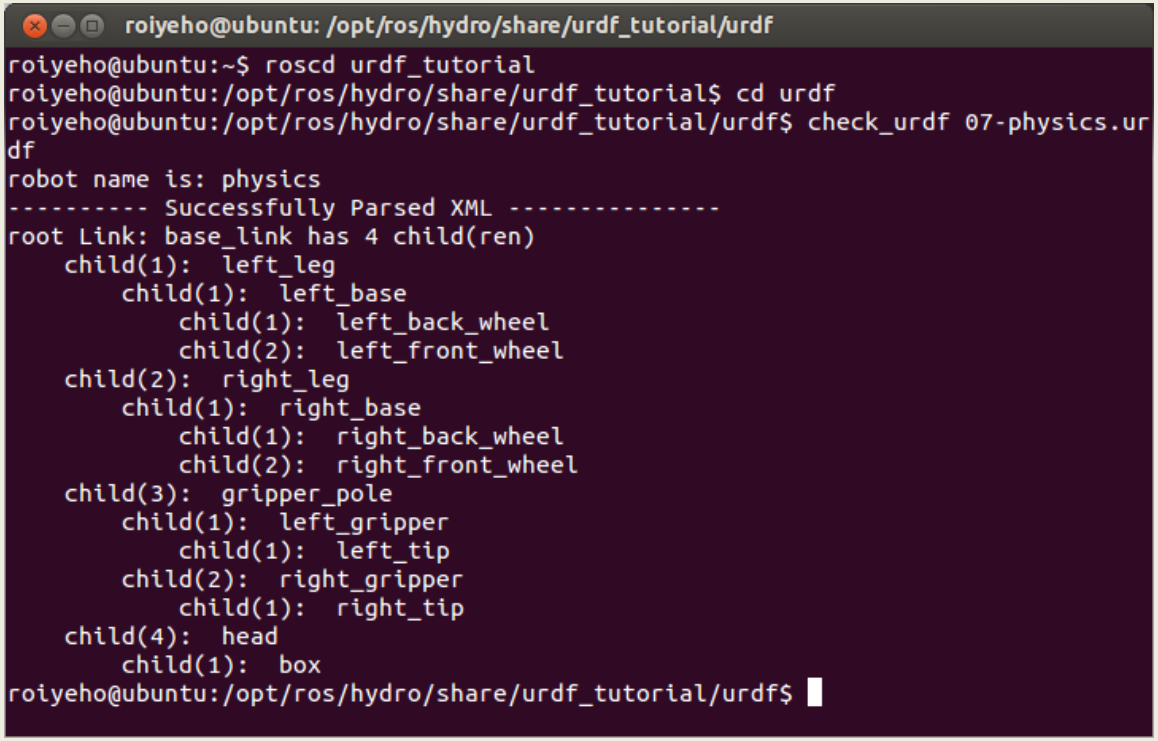
URDF Verification

- The robot model stack includes the command line tool **check_urdf**
- It's called with a single command line argument naming a file
- It attempts to parse the file as a URDF description, and either prints a description of the resulting kinematic chain, or an error message

URDF Verification

- For example, to run this tool on the R2D2 urdf:

```
$ check_urdf 07-physics.urdf
```



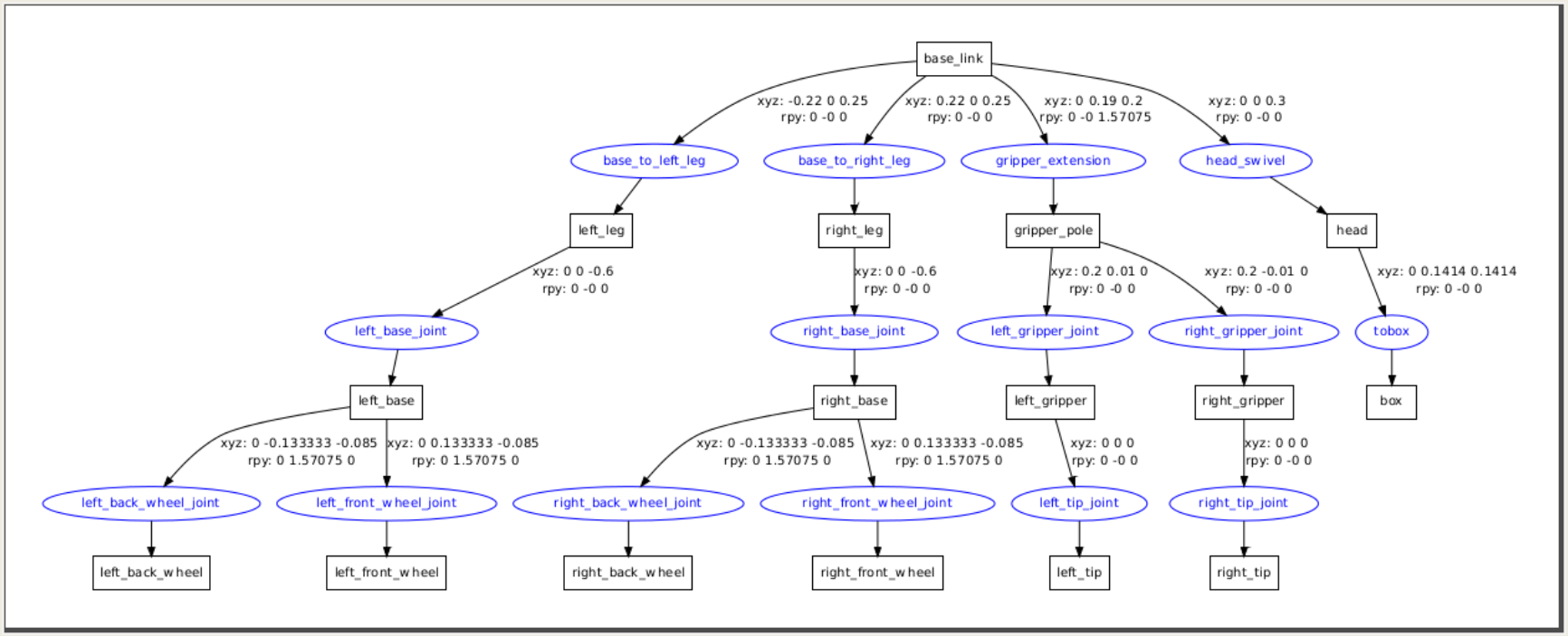
URDF Visualization

- To get a graphviz diagram of your urdf file, type:

```
$ cp 07-physics.urdf /tmp
$ urdf_to_graphiz /tmp/07-physics.urdf
$ evince /tmp/physics.pdf
```

- The result is a file called physics.pdf that looks something like this:

URDF Visualization



PR2 URDF

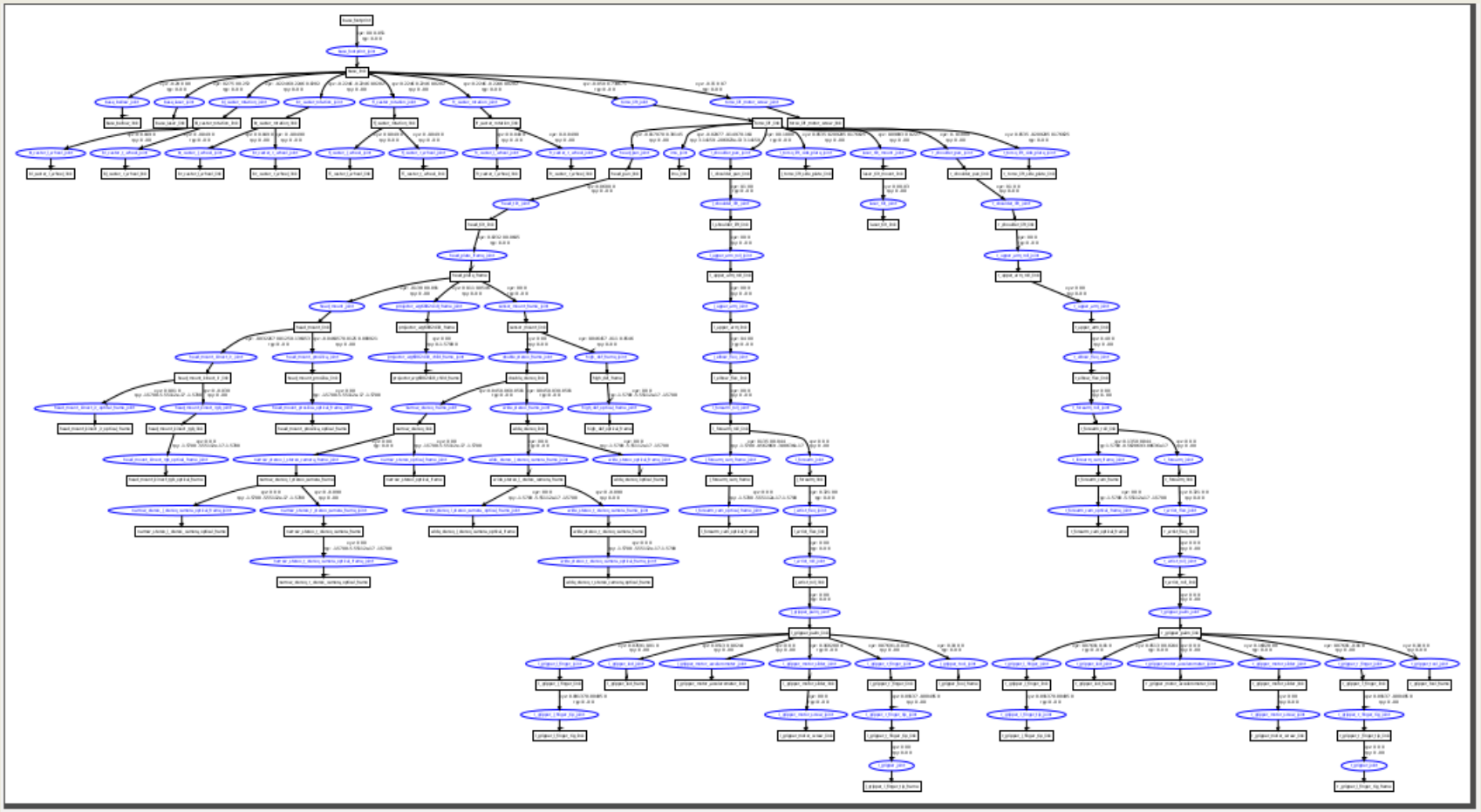
- To see the PR2 URDF graphically, first create the urdf file by running:

```
$ rosrun xacro xacro.py `rospack find
pr2_description`/robots/pr2.urdf.xacro > /tmp/pr2.urdf
```

- Then run the urdf_to_graphiz tool:

```
$ rosrun urdfdom urdf_to_graphiz /tmp/pr2.urdf
$ evince /tmp/pr2.urdf
```

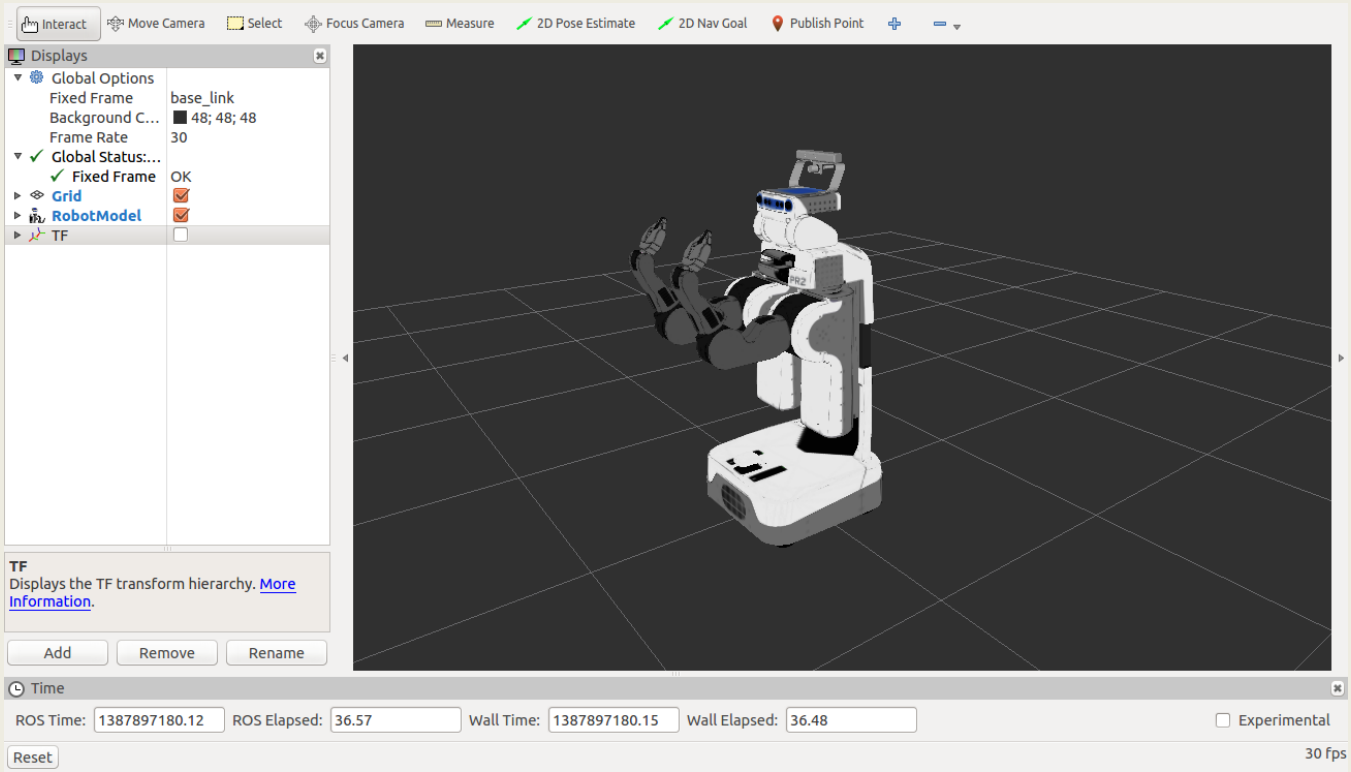
PR2 URDF



Watch PR2 in rviz

- To watch PR2's urdf in rviz, follow these steps:

```
$ cd /tmp
$ roslaunch urdf_tutorial display.launch model:=pr2.urdf gui:=True
```



Homework (not for submission)

- Create a URDF model for the following robot
  - The robot contains 4 links (base, left and right wheels and a caster) and 3 joints that connect the wheels and the caster to the base
  - Attach to the robot's body a mesh taken from the Pioneer robot's model (model://pioneer2dx/meshes/chassis.dae)