Experimental Robotics Laboratory

Carmine Tommaso Recchiuto

Experimental Robotics Laboratory

Why?

- ✓ Hands-on experiences are required for engineering education.
- ✓ By working on robotic simulations and with real robots, you are going to put into practice many of the theoretical aspects that you have seen (or that you are seeing this semester) in different other courses.
- ✓ ROS 2!

Course Overview

- Robot modelling (URDF, XACRO, ROS control)
- OpenCV and ARuco
- ROSbots: assignment 1
- ROSPlan and PlanSys2
- ROSbots: assignment 2
- Localization and Mapping:
- Rest API
- ROSbots: assignment 2

How to use ROS in this course

- I strongly suggest installing Docker and pull the Docker image of the course
- Alternatively you may use:
  - a Native Linux System, with ROS 2 installed
  - a Virtual Machine, with Linux-ROS 2

✓ Any Ubuntu version could be used (20, 22, 24). I will use Ubuntu 24 (and ROS2 Jazzy)

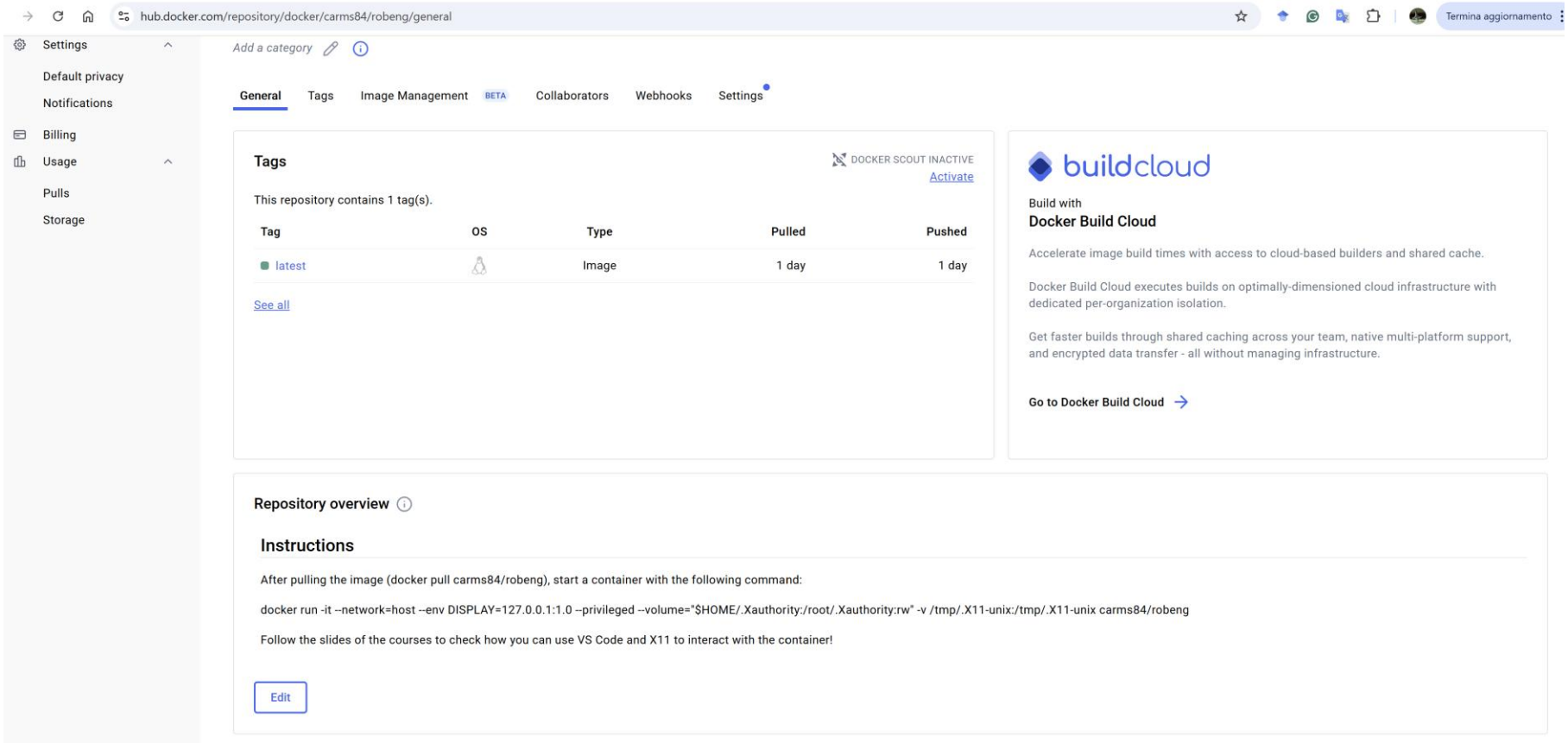
Research Track - Docker Image

✓ Install Docker
- Follow instructions from: https://docs.docker.com/install/

✓ Get the image
- Run `docker pull carms84/robeng`

✓Run a container Follow instructions from https://hub.docker.com/repository/docker/carms84/robeng/general

✓https://hub.docker.com/repository/docker/carms84/robeng/general



MAC USERS! Use the command:

docker run -it --network=host --platform linux/amd64 --env DISPLAY=127.0.0.1:1.0 --privileged --volume="$HOME/.Xauthority:/root/.Xauthority:rw" -v /tmp/.X11-unix:/tmp/.X11-unix carms84/robeng

How to use the Docker Image

The suggested strategy for using Docker is using an X11 server, and VsCode.

- Concerning X11, I suggest installing Xlaunch (or Xquartz on MacOS). Select *Disable Access Control* when configuring it.

- Install VsCode, by also adding the extension Dev Containers and Remote – SSH

Please refer to https://code.visualstudio.com/docs/devcontainers/containers for additional info.

Evaluation

Assignments + Final evaluation: written exam (open-ended questions + practical exercises)

Assignments will have a (hard) deadline and they should submitted as a Git repository. If you don't participate to practical activities in lab you can still take the final exam, but you will have to do the assignments at least in simulation to take the exam. In this case however, the assignments will have a fixed evaluation (66/100). They will constitute the 35% of the final evaluation.

The final exam will have some questions related to the assignments done, and to general aspects seen during the course

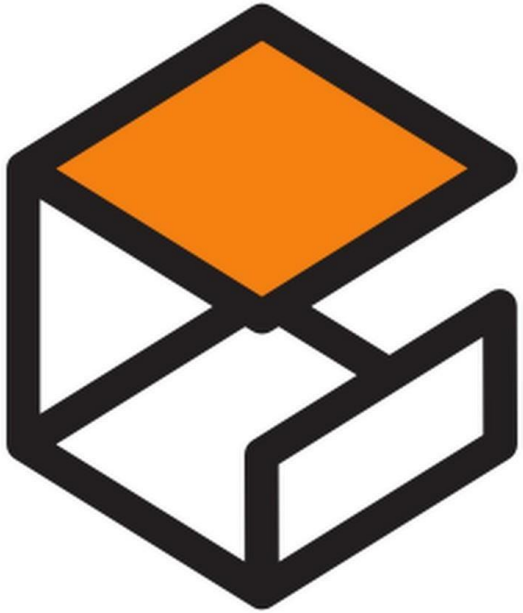
Robot modelling
URDF and XACRO
Integration with ROS and Gazebo

Carmine Tommaso Recchiuto

Robot Modelling: GAZEBO and URDF

Main features of Gazebo

- Dynamic simulation based on various physics engines (ODE, Bullet, Simbody and DART)
- Sensors (with noise) simulation
- Plugin to customize robots, sensors and the environment
- Realistic rendering of the environment and the robots
- Library of robot models
- ROS integration
- Advanced features
- Remote & cloud simulation Open source



CUSTOMIZATION

What kind of customization are we looking for in a simulator?

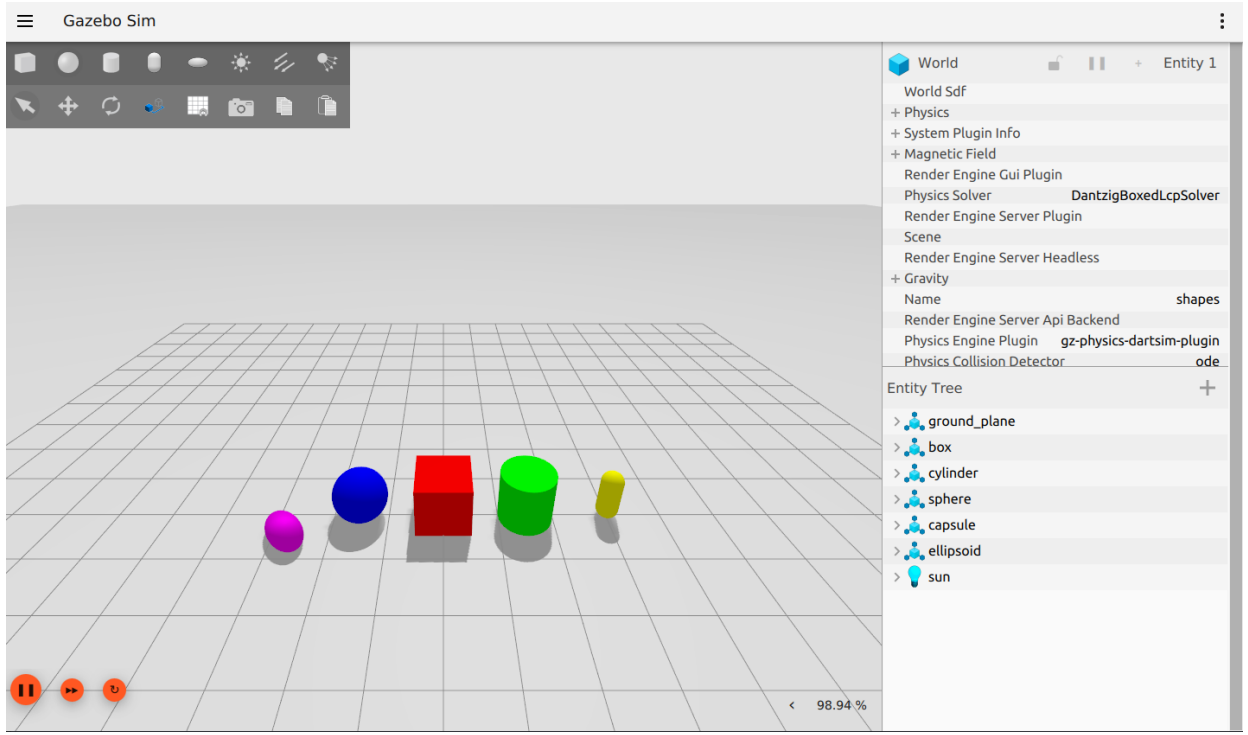
- Modifying existing robot or sensor models
- Building our own robot or sensor models
- Modifying the behavior of existing robotmodels
- Controlling and defining a behavior for our own robot models
- Creating specific environment compatible with our experiments

Using the GUI

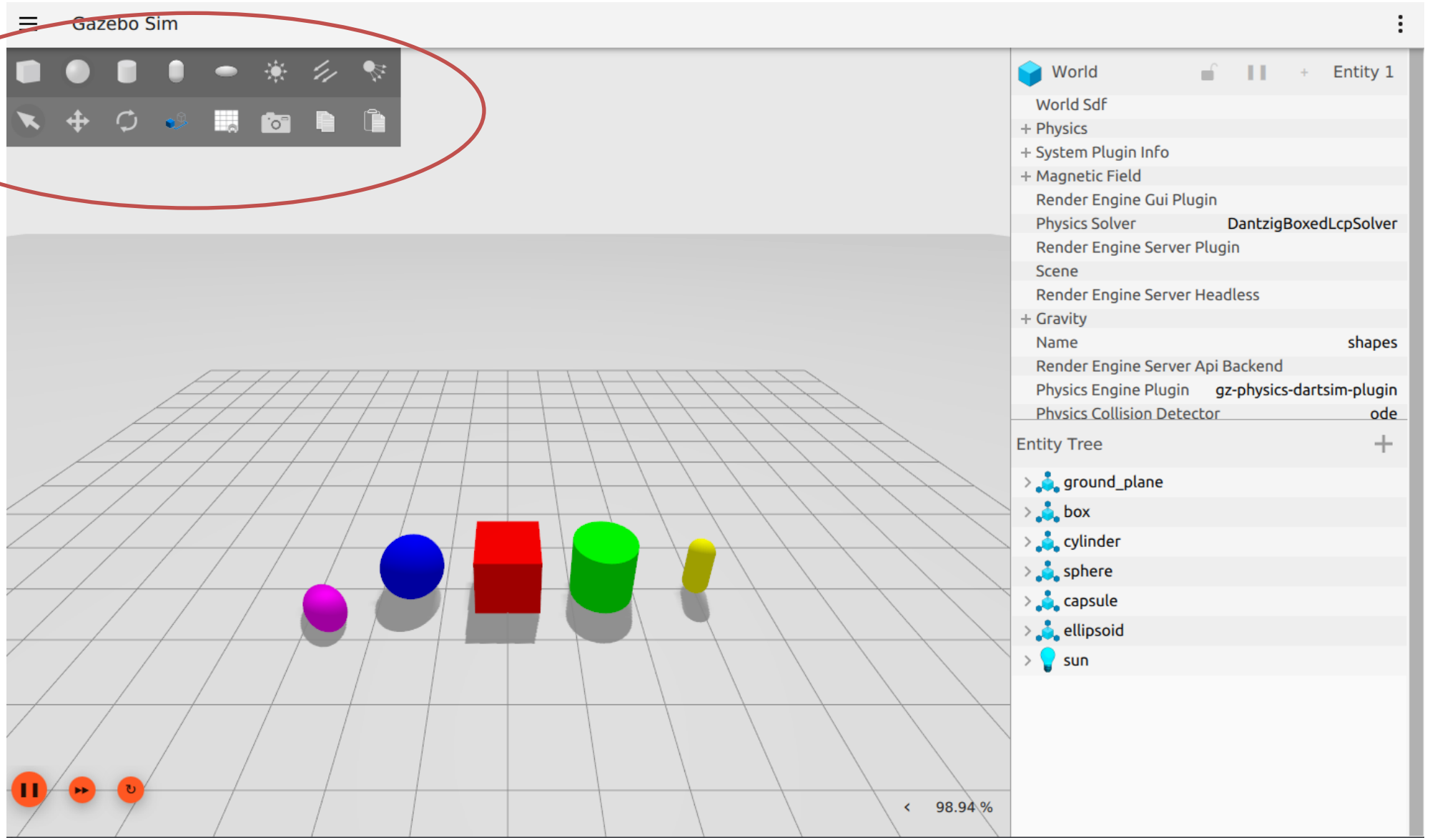
You can start Gazebo by running a simple word:

gz sim shapes.sdf

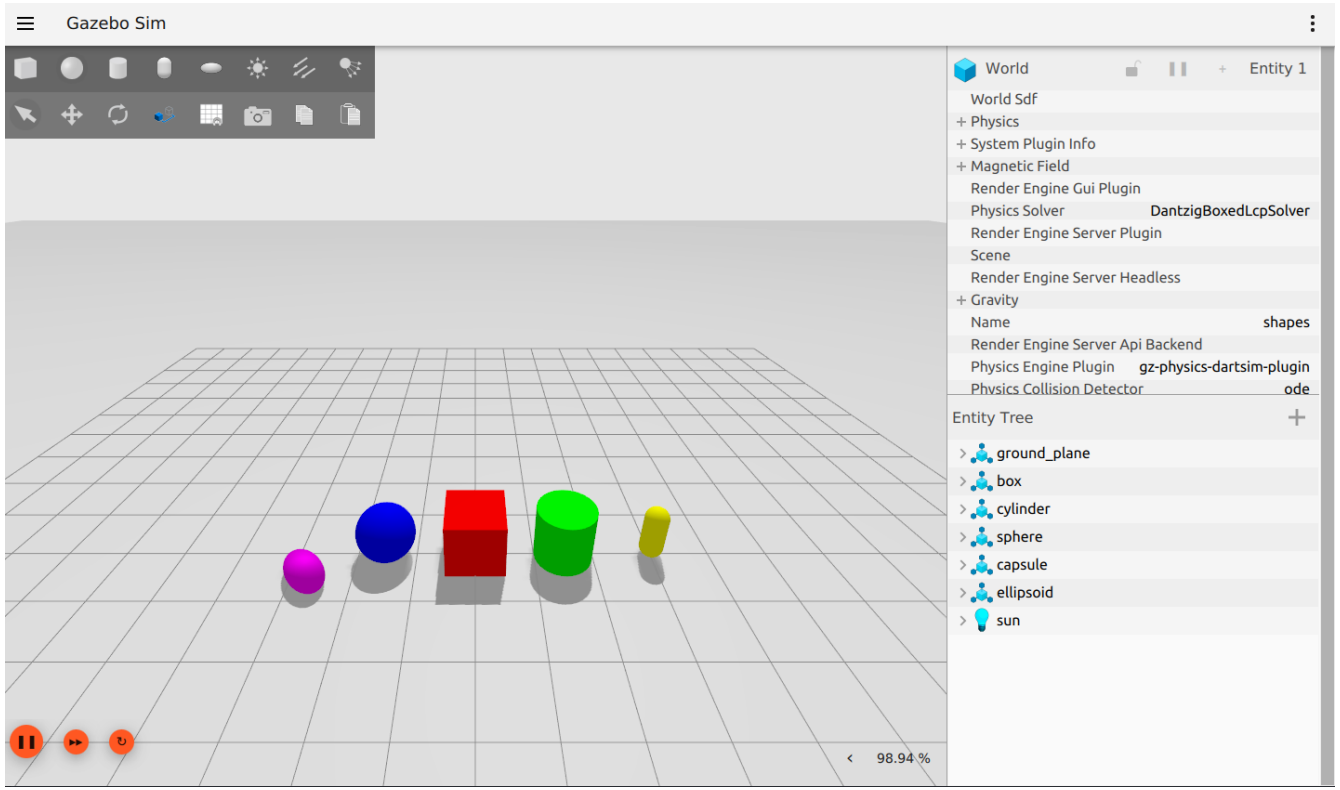
Notice that by doing that you are starting the simulator environment without using the ROS2 framework.



Using the GUI



Using the GUI



In the right panel there are two plugins:

- the Component Inspector
- the Entity Tree

Other plugins can be added:

- Align Tool
- Plot3D
- ...

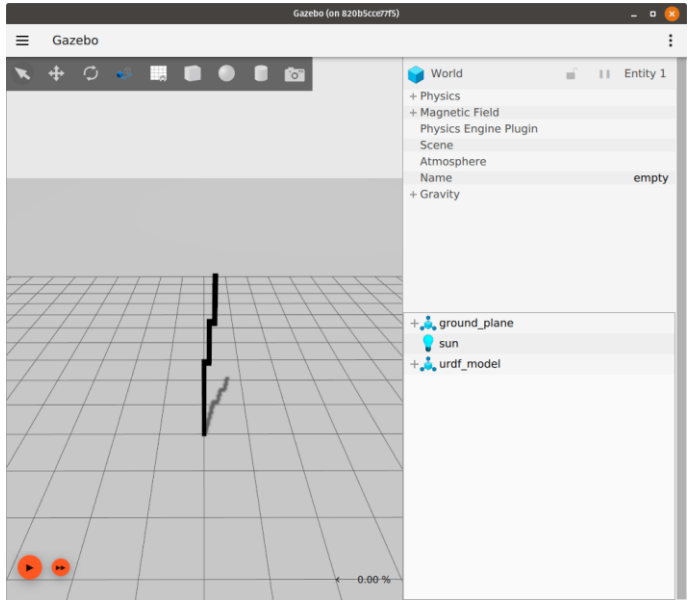
Using the GUI

Alternatively, it is possible to start from an empty world and spawn a model

gz sim empty.sdf

Example:

gz service -s /world/empty/create --reqtype gz.msgs.EntityFactory --reptype gz.msgs.Boolean --timeout 1000 --req 'sdf_filename: "/opt/ros/jazzy/share/dummy_robot_bringup/launch/single_rrbot.urdf", name: "urdf_model"'



Gazebo services and topics

Gazebo (gz) services are methods for requesting and executing tasks on a running Gazebo simulation, e.g. adding entities, while Gazebo (gz) topics are named channels for broadcasting and subscribing to real-time data, such as sensor readings or model states.

`/world/<world_name>/reset`: Services to reset the simulation state.

`/cmd_vel`: A common topic for sending velocity commands to a robot model.

Gazebo services and topics

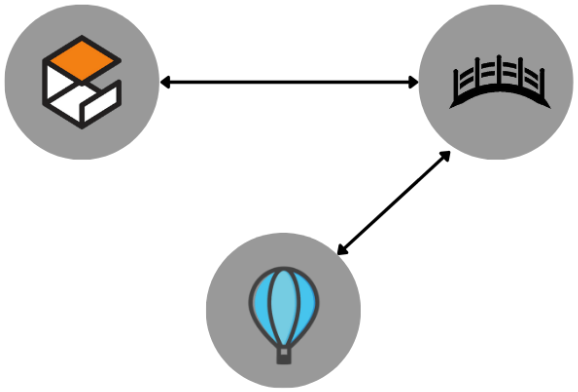
In principle, you could use only GZ (without ROS2, by leveraging the gz_transport APIs)

Es.

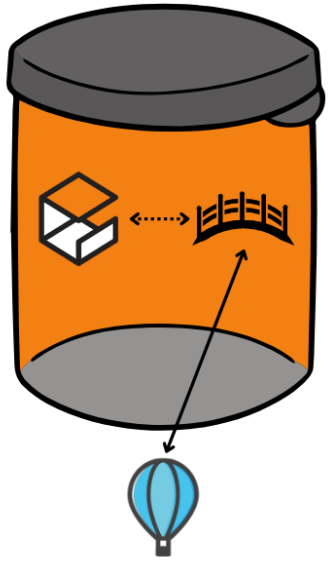
```
int main(int argc, char **argv)
{
// Create a transport node and advertise a topic.
gz::transport::Node node;
std::string topic = "/my_topic";
auto pub = node.Advertise<gz::msgs::StringMsg>(topic);
gz::msgs::StringMsg msg;
msg.set_data("HELLO");
pub.Publish(msg);
[…]
}
```

ROS2 Integration

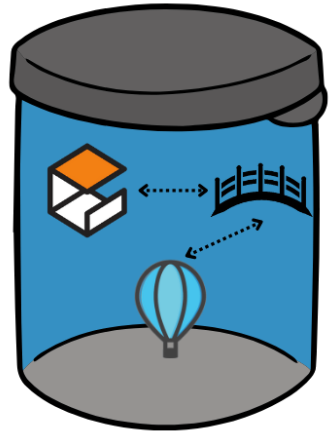
How is this integrated with ROS2? ROS2_gz BRIDGE! (Reference: https://gazebosim.org/docs/latest/ros2_overview/)



use_composition:=False



use_composition:=True
create_own_container:=True



use_composition:=True
create_own_container:=False

ROS2 Integration

Launch file to start GZ together with the bridge:

https://github.com/gazebosim/ros_gz/blob/ros2/ros_gz_sim/launch/ros_gz_sim.launch.py

We can then start again the simulation as before, but this time adding the bridge:

ros2 launch ros_gz_sim ros_gz_sim.launch.py
world_sdf_file:=.shapes.sdf
bridge_name:=ros_gz_bridge
config_file:=/home/ubuntu/test.yaml
use_composition:=True
create_own_container:=True

The file test.yaml should be something like this:

```
- ros_topic_name: "clock"
  gz_topic_name: "/world/empty/clock"
  ros_type_name: "rosgraph_msgs/msg/Clock"
  gz_type_name: "gz.msgs.Clock"
  subscriber_queue: 5        # Default 10 if
qos_profile is empty, otherwise not set by
default
  publisher_queue: 6         # Default 10 if
qos_profile is empty, otherwise not set by
default
  lazy: true                 # Default "false"
  direction: BIDIRECTIONAL   # Default
"BIDIRECTIONAL" - Bridge both directions
                             # "GZ_TO_ROS" -
Bridge Gz topic to ROS
                             # "ROS_TO_GZ" -
Bridge ROS topic to Gz
  qos_profile: SENSOR_DATA   # Default is a
default-constructed QoS with appropriate queue
size
```

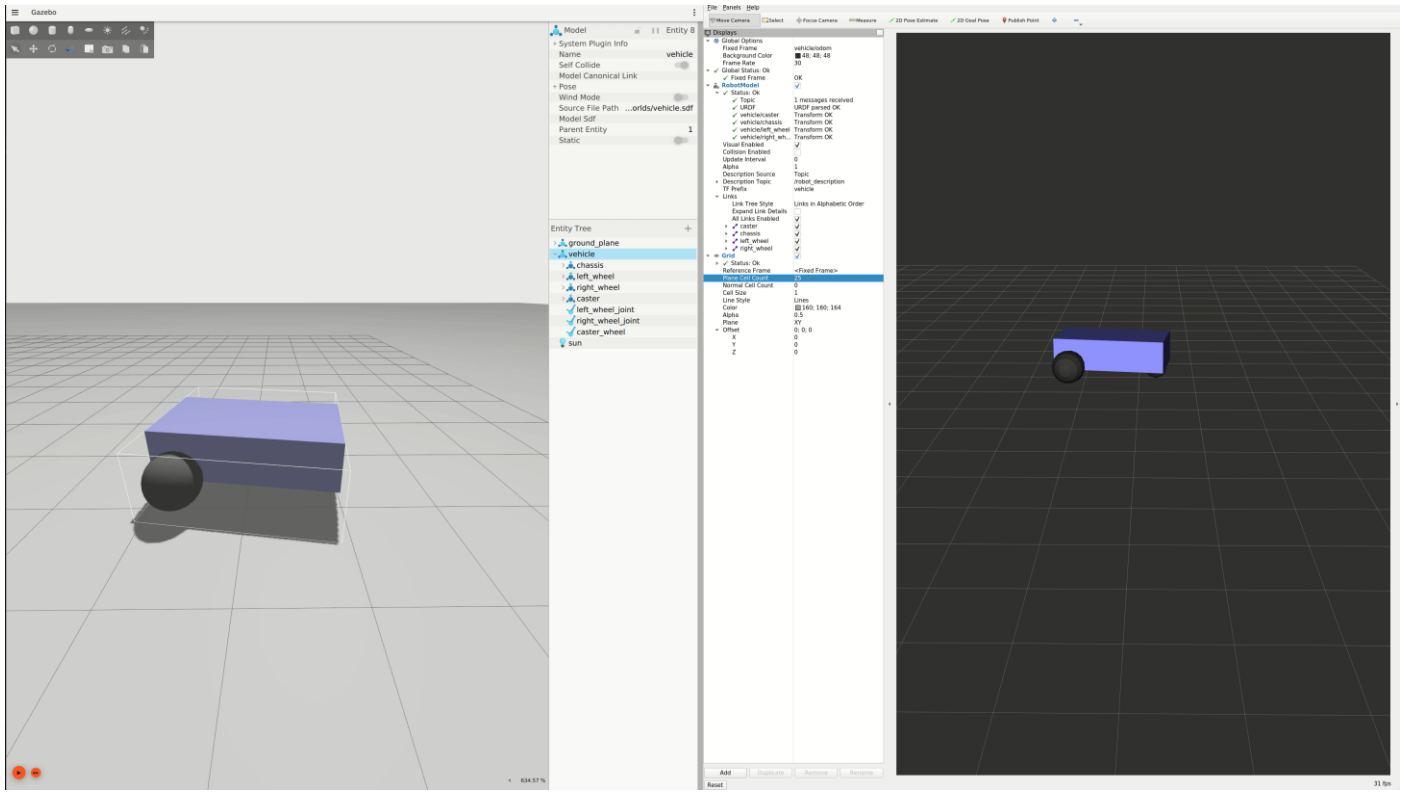
ROS2 Integration

As you can notice, gz_sim_launch only start the gz server + the bridge. If you also want the graphical interface, you could:

- run *gz sim –g* in another terminal

- Modify the launch file adding

```
gz_client = ExecuteProcess(
    cmd=['gz', 'sim', '-g'],
    output='screen'
)
```



ROS2 Integration

You can also start gz without the bridge:

```
ros2 launch ros_gz_sim gz_sim.launch.py gz_args:="shapes.sdf"
```

And then run the gz_bridge, by specifying the configuration!

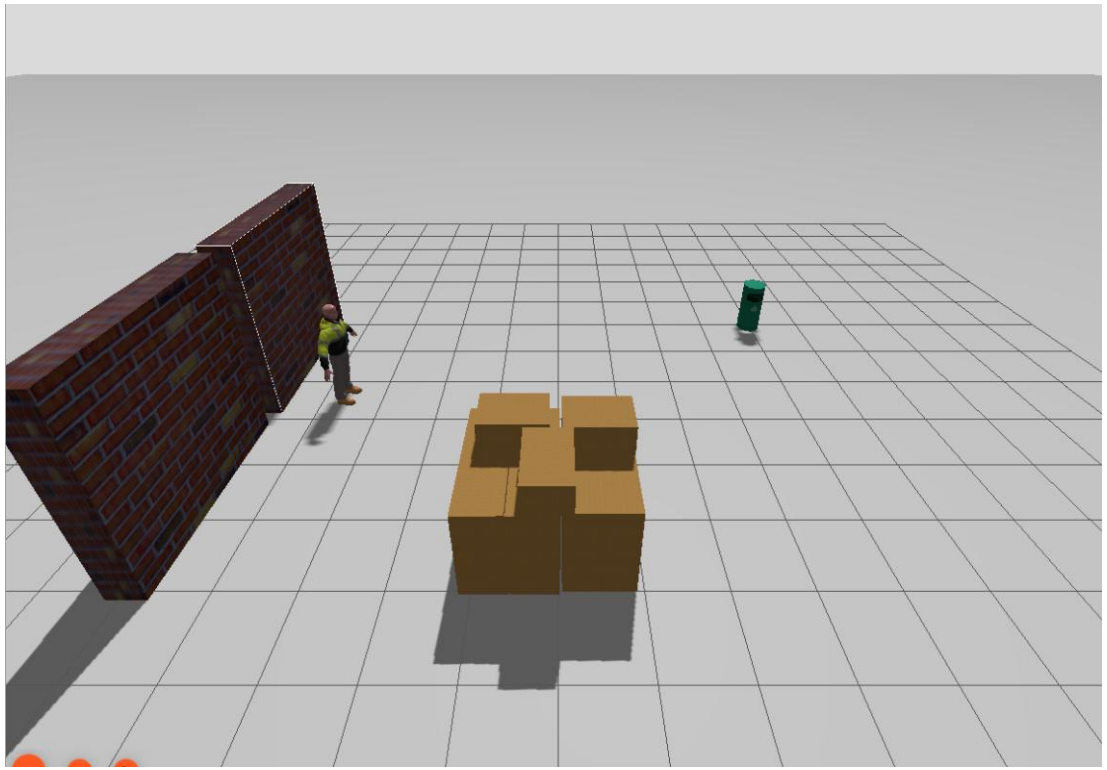
```
ros2 run ros_gz_bridge parameter_bridge clock@rosgraph_msgs/msg/Clock@gz.msgs.Clock
/keyboard/keypress@std_msgs/msg/Int32@gz.msgs.Int32
```

Building your own world

Let's create a new package: `ros2 pkg create --build-type ament_cmake erl1`

Let's start again from our empty world: `gz sim empty.sdf`

We can add resources the world, and save it into the worlds folder of our package



Building your own world

After we created the new world file, we would like to launch Gazebo and load the world using ROS.
We can create a launch file to this aim:

Also add to the CMakeLists.txt:

```
install(DIRECTORY
  launch
  worlds
  DESTINATION share/${PROJECT_NAME}
)
```

```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.substitutions import  LaunchConfiguration, PathJoinSubstitution, TextSubstitution


def generate_launch_description():

    world_arg = DeclareLaunchArgument(
        'world', default_value='my_world.sdf',
        description='Name of the Gazebo world file to load'
    )

    pkg_erl1= get_package_share_directory('erl1')
    pkg_ros_gz_sim = get_package_share_directory('ros_gz_sim')

    # Add your own gazebo library path here
    gazebo_models_path = "/home/ubuntu/gazebo_models"
    os.environ["GZ_SIM_RESOURCE_PATH"] += os.pathsep + gazebo_models_path


    gazebo_launch = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(pkg_ros_gz_sim, 'launch', 'gz_sim.launch.py'),
        ),
        launch_arguments={'gz_args': [PathJoinSubstitution([
            pkg_erl1,
            'worlds',
            LaunchConfiguration('world')
        ]),
        #TextSubstitution(text=' -r -v -v1 --render-engine ogre')],
        TextSubstitution(text=' -r -v -v1')],
        'on_exit_shutdown': 'true'}.items()
    )

    launchDescriptionObject = LaunchDescription()

    launchDescriptionObject.add_action(world_arg)
    launchDescriptionObject.add_action(gazebo_launch)

    return launchDescriptionObject
```

Building your own robot

Two standard, SDF from Gazebo and URDF from ROS

SDF

- xml
- Developed as part of Gazebo
- Describe robots and Scene
- Visual design environment
- SDF file in gazebo can interact with ROS code

URDF

- xml
- Developed as part of ROS
- High ROS integration
- It only describes robots
- No "official" tools, just RVIZ to visualize the robot
- URDF files can be imported in Gazebo

SIMULATION DESCRIPTION FORMAT

As any XML file is composed by tags, but differently from some XML files the structure is quite simple

Tag structure:

- **sdf**
  - **world**
  - **model**
  - **actor**
  - **light**

```
<?xml version='1.0'?>
<sdf version='1.6'>
  <world name='default'>
    ...
  </world>
</sdf>
```

```
<?xml version='1.0'?>
<sdf version='1.6'>
  <actor name='act'>
    ...
  </actor>
</sdf>
```

```
<?xml version='1.0'?>
<sdf version='1.6'>
  <model name='model'>
    ...
  </model>
</sdf>
```

```
<?xml version='1.0'?>
<sdf version='1.6'>
  <light name='light'>
    ...
  </light>
</sdf>
```

Building your own robot

URDF is Universal Robot Description Format, it's an XML format for representing a robot model commonly used in ROS, RViz and Gazebo

A robot description (3D model) in URDF is built up as the tree of links and joints that connects links together. A parent link can have multiple children, but a link can only have a single parent.

In the links we can define the mechanical parameters of the link (weight, inertia), the collision shape for the physical simulation and it's visual properties (e.g. detailed 3D models).

In the joints we define the parent and child links and we tell to the simulation what kind of joint do we have (fixed, rotation or linear).

Types of joints

revolute - a hinge joint that rotates around 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. All degrees of freedom are locked. This type of joint does not require the axis, calibration, dynamics, limits or safety_controller.

floating - This joint allows motion for all 6 degrees of freedom.

planar - This joint allows motion in a plane perpendicular to the axis.

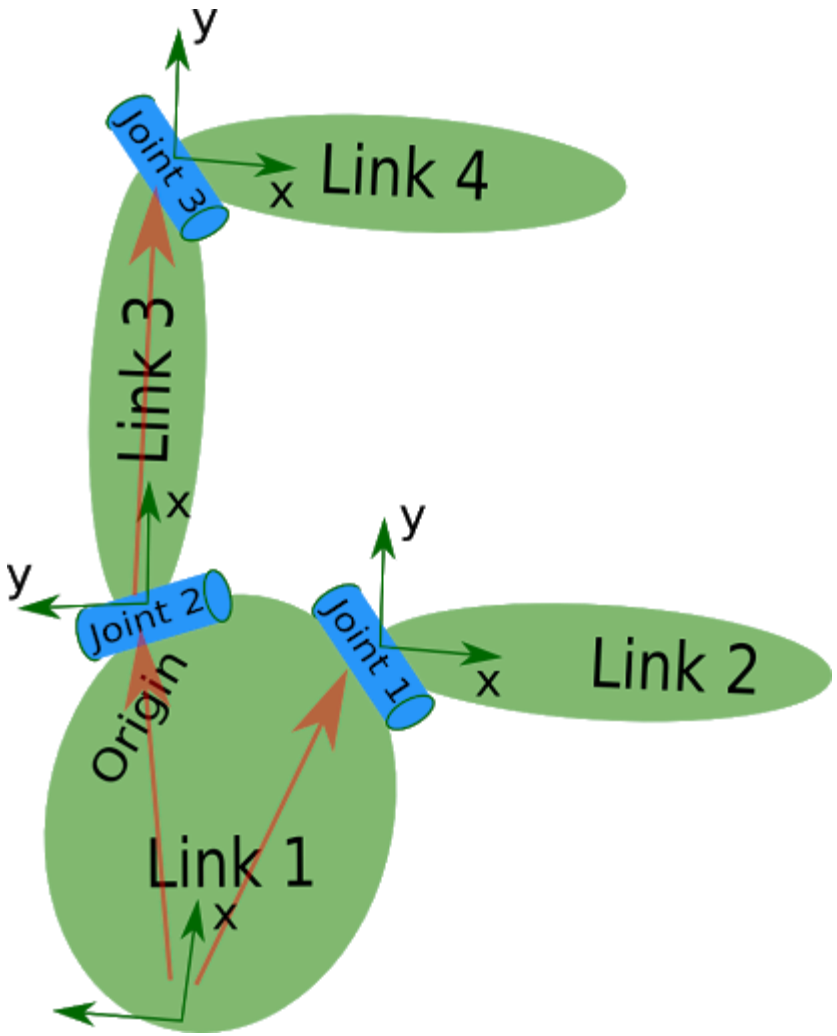
URDF: Unified Robot Description Format

Another XML specification to describe a robot.

Limitation: only tree structures can be represented, ruling out all parallel robots. Also, the specification assumes the robot consists of rigid links connected by joints; flexible elements are not supported.

This format is used in ROS to describe all elements of a robot. To use a URDF file in Gazebo, some additional simulation-specific tags must be added to work properly with Gazebo

The resulting description is a tree defined by the connection between links (nodes) and joints (edges).



Creating a URDF file

First of all let's create our robot's URDF in the urdf folder with erl1.urdf name. To start from the bare minimum let's add the following xml code to the file.

```
<?xml version='1.0'?>

<robot name="erl1" xmlns:xacro="http://www.ros.org/wiki/xacro">

<link name="link_chassis"></link>

</robot>
```

Now we only have the very first link of our robot without any mechanical, collision or visual properties, there is nothing to see yet.

URDF file: LINKS

We can then specify the visual properties of the link. This element specifies the shape of the object (box, cylinder, etc.) for visualization purposes. Note: multiple instances of <visual> tags can exist for the same link. The union of the geometry they define forms the visual representation of the link.

name (optional) Specifies a name for a part of a link's geometry. This is useful to be able to refer to specific bits of the geometry of a link.

<origin> (optional) The reference frame of the visual element with respect to the reference frame of the link

xyz (optional: defaults to zero vector)
Represents the position of the visual frame (i.e. of the geometrical center of the geometry) with respect to the origin of the geometry

rpy (optional: defaults to zero vector)
Represents the orientation of the visual frame with respect to the world frame.

URDF file: LINKS

<geometry> (required) The shape of the visual object. This can be one of the following:

<box> size attribute contains the three side lengths of the box. The origin of the box is in its center.

<cylinder> Specify the radius and length. The origin of the cylinder is in its center.

<sphere> Specify the radius. The origin of the sphere is in its center.

<mesh> A mesh element specified by a filename, and an optional scale that scales the mesh's axis-aligned-bounding-box. Any geometry format is acceptable but specific application compatibility is dependent on implementation. The recommended format for best texture and color support is Collada .dae files. The mesh file must be a local file. Prefix the filename with package://<packagename>/<path> to make the path to the mesh file relative to the package <packagename>

URDF file: LINKS

<material> (optional) The material of the visual element. It is allowed to specify a material element outside of the 'link' object, in the top level 'robot' element. From within a link element you can then reference the material by name.

name name of the material

<color> (optional)

rgba The color of a material specified by set of four numbers representing /green/blue/alpha, each in the range of [0,1].

<texture> (optional) The texture of a material is specified by a filename

N.B. The material tag has only effect on Rviz!

Physical and Collision Models

- ✓ If you want to simulate the robot on Gazebo or any other simulation software, it is necessary to add physical and collision properties.

- ✓ This means that we need to set the dimension of the geometry to calculate the possible collisions, for example, the weight that will give us the inertia, and so on

- ✓ Indeed, so far, we've only specified our links with a single sub-element, visual, which defines (not surprisingly) what the robot looks like. However, in order to get collision detection to work or to simulate the robot in something like Gazebo, we need to define a collision element as well.

  Es. robot1_physics.urdf

Physical and Collision Models

<collision> (optional) The collision properties of a link. Note that this can be different from the visual properties of a link, for example, simpler collision models are often used to reduce computation time. Note: multiple instances of <collision> tags can exist for the same link. The union of the geometry they define forms the collision representation of the link.

name (optional) Specifies a name for a part of a link's geometry.

<origin> (optional) The reference frame of the collision element, relative to the reference frame of the link.

xyz (optional: defaults to zero vector) Represents the offset.

rpy (optional: defaults to zero vector) Represents the fixed axis roll, pitch and yaw angles in radians.

<geometry> See the geometry description in the above visual element.

Physical and Collision Models

Example of Collision TAG in robot1_physics.urdf

```
<collision name="collision_chassis">
    <geometry>
        <box size="0.5 0.3 0.07"/>
    </geometry>
</collision>
```

Physical and Collision Models

In order to get your model to be properly simulated, you need to define several physical properties of your robot, i.e. the properties that a physics engine like Gazebo would need. Every link element being simulated needs an inertial tag

The 3x3 rotational inertia matrix is specified with the inertia element. Since this is symmetrical, it can be represented by only 6 elements, as such:

ixx, ixy, ixz, iyy, iyz,izz

This information can be provided to you by modeling programs such as MeshLab. The inertia of geometric primitives (cylinder, box, sphere) can bealso computed using Wikipedia's list of moment of inertia tensors (and is used in the current example).

Physical and Collision Models

<inertial> (optional: defaults to a zero mass and zero inertia if not specified) The inertial properties of the link.

<origin> (optional) This is the pose of the inertial reference frame, relative to the link reference frame. The origin of the inertial reference frame needs to be at the center of gravity. The axes of the inertial reference frame do not need to be aligned with the principal axes of the inertia.

xyz (optional: defaults to zero vector) Represents the offset.

rpy (optional: defaults to zero vector) Represents the fixed axis roll, pitch and yaw angles in radians.

<mass> The mass of the link is represented by the value attribute of this element

<inertia> The 3x3 rotational inertia matrix, represented in the inertia frame. Because the rotational inertia matrix is symmetric, only 6 above-diagonal elements of this matrix are specified here, using the attributes ixx, ixy, ixz, iyy, iyz, izz. These can be found for some primitive shapes

Physical and Collision Models

Example of Inertia TAG in robot1_physics.urdf

```
<inertial>
    <mass value="5"/>
    <origin rpy="0 0 0" xyz="0 0 0.1"/>
    <inertia ixx="0.0395416666667" ixy="0" ixz="0" iyy="0.106208333333"
             iyz="0" izz="0.106208333333"/>
</inertial>
```

Using XACRO

- Problem: adding inertial and collision properties, the file may become very long and difficult to read
- Links and Joints (i.e., the 2 wheels in our model) can be similar: it should be possible to reduce the size of the file describing the robot by using some tricks.
- The XACRO format helps in reducing the overall size of URDF files and makes it easier to read and maintain the packages.
- It allows us to create modules and reuse them, in case of repeated structures

Using XACRO

- Files should have the extension .xacro instead of .urdf
- XACRO stands for "XML Macros": you can construct shorter and more readable XML files, by using macros that expand to larger XML expressions.
- Typical first lines of a valid XACRO files:

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

- **Identifier of the namespace**: Our robot will refer to the namespace *http://www.ros.org/wiki/xacro,* and its name will be *robot1_xacro*

Using CONSTANTS

- With *xacro* we can declare constant values; hence, you can avoid putting the same value in a lot of lines.

- Examples: the two wheels have the same values for length and radius. We can define a property

```
<xacro:property name="length_wheel" value="0.04" />
<xacro:property name="radius_wheel" value="0.1" />
```

- Once defined these variables, you only have to change the old value with the new value:

```
<geometry>
        <cylinder length="${lenght_wheel}" radius="${radius_wheel}"/>
</geometry>
```

Using MATH

- You can build up even complex expression, by using the ${} construct and the four basic operations.

```
<cylinder lenght=".1" radius="${radius_wheel/2}" />
<origin xyz="${test*(witdh+0.02)} ${0.5-0.3} 0.2" />
```

- By using math, you can resize the model by only changing a value

Using MACROS

- Macros are the most useful component of the xacro package.
- Example: a macro for *inertial!*

Using MACROS

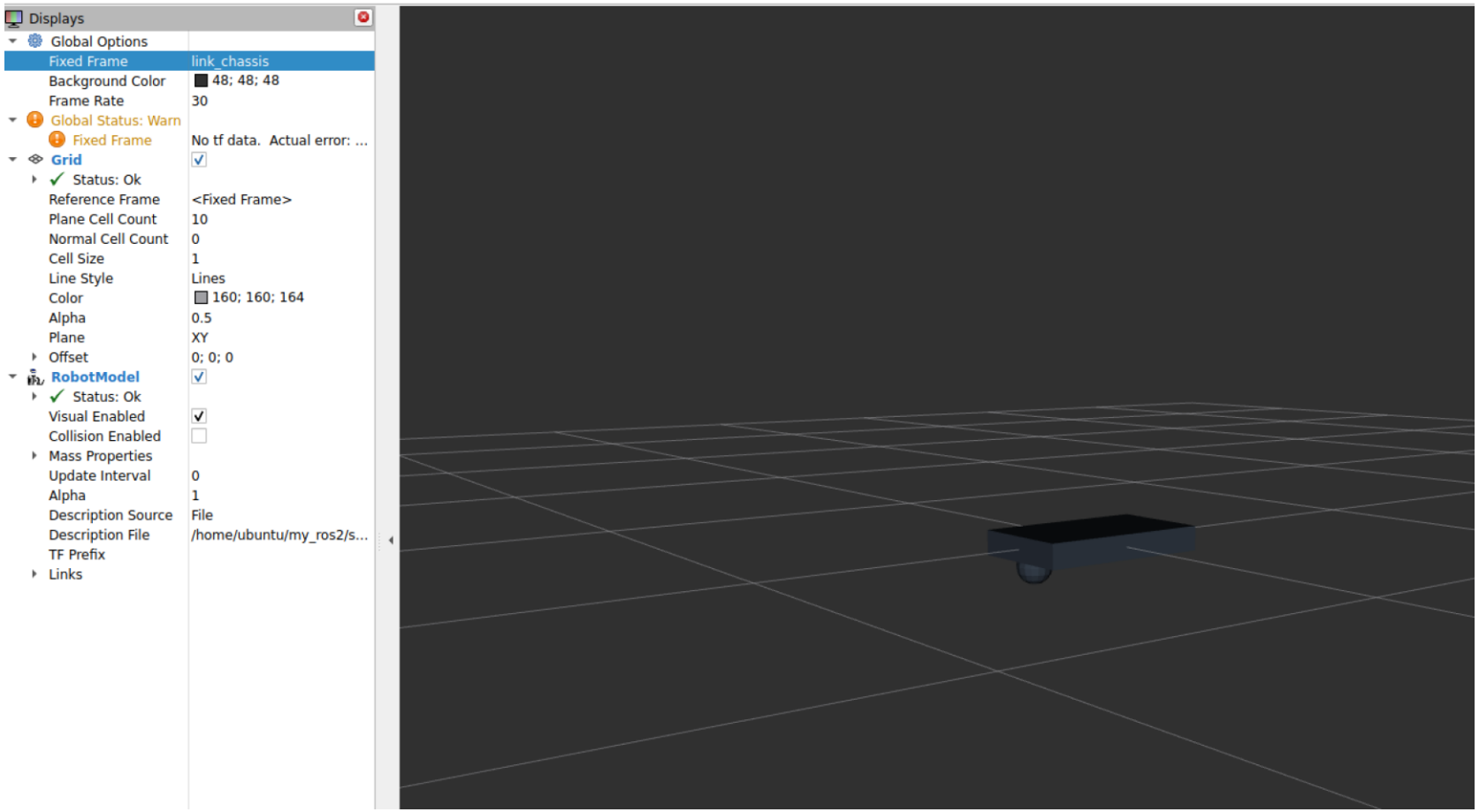
```
<xacro:macro name="default_inertial" params="mass">
        <inertial>
            <mass value="${mass}" />
            <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
            <inertia ixx="0.000526666666667" ixy="0" ixz="0"
iyy="0.000526666666667" iyz="0" izz="0.001"/>
        </inertial>
</xacro:macro>
```

- We can then recall the macro with:

```
<xacro:default_inertial mass="0.2" />
```

URDF file: LINKS

```
<material name="blue">
   <color rgba="0.203125 0.23828125 0.28515625 1.0"/>
 </material>


<link name="link_chassis">
   <!-- pose and inertial -->
   <pose>0 0 0.1 0 0 0</pose>
   <inertial>
     <mass value="5"/>
     <origin rpy="0 0 0" xyz="0 0 0.1"/>
     <inertia ixx="0.0395416666667" ixy="0" ixz="0"
iyy="0.106208333333" iyz="0" izz="0.106208333333"/>
   </inertial>
   <!-- body -->
   <collision name="collision_chassis">
     <geometry>
       <box size="0.5 0.3 0.07"/>
     </geometry>
   </collision>
   <visual>
     <origin rpy="0 0 0" xyz="0 0 0"/>
     <geometry>
       <box size="0.5 0.3 0.07"/>
     </geometry>
     <material name="blue"/>
   </visual>
   <!-- caster front -->
   <collision name="caster_front_collision">
     <origin rpy=" 0 0 0" xyz="0.35 0 -0.05"/>
     <geometry>
       <sphere radius="0.05"/>
     </geometry>
   </collision>
   <visual name="caster_front_visual">
     <origin rpy=" 0 0 0" xyz="0.2 0 -0.05"/>
     <geometry>
       <sphere radius="0.05"/>
     </geometry>
   </visual>
 </link>
```



Rviz2

Although is possible to use RViz like this, it's not the most convinient way. Later, when we add more links and joints we'll have the problem that the we don't have any program running that informs RViz about the right transformation among these links (if the joint isn't fixed).

Instead of this manual usage of RViz, let's move all these tasks into a ROS launch file.

```
ros2 launch urdf_tutorial display.launch.py model:=urdf/01-
myfirst.urdf
```

Based on this launch file, we could create our own launch file for our model!

First, let’s save a Rviz config file (urdf.rviz) in the rviz folder.

Rviz2

As before, let's add the launch file to the launch directory of the package, and let's modify the CMakeLists accordingly

```
install(DIRECTORY
  launch
  worlds
  rviz
  urdf
  DESTINATION
share/${PROJECT_NAME}
)
```

```
import os
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, IncludeLaunchDescription
from launch.substitutions import LaunchConfiguration, PathJoinSubstitution
from launch_ros.substitutions import FindPackageShare


def generate_launch_description():

    pkg_erl1= FindPackageShare('erl1')
    default_rviz_config_path = PathJoinSubstitution([pkg_erl1, 'rviz', 'urdf.rviz'])

    # Show joint state publisher GUI for joints
    gui_arg = DeclareLaunchArgument(name='gui', default_value='true', choices=['true', 'false'],
                        description='Flag to enable joint_state_publisher_gui')

    # RViz config file path
    rviz_arg = DeclareLaunchArgument(name='rvizconfig', default_value=default_rviz_config_path,
                        description='Absolute path to rviz config file')


    # URDF model path within the bme_gazebo_basics package
    model_arg = DeclareLaunchArgument(
        'model', default_value='erl1.urdf',
        description='Name of the URDF description to load'
    )

    # Use built-in ROS2 URDF launch package with our own arguments
    urdf = IncludeLaunchDescription(
        PathJoinSubstitution([FindPackageShare('urdf_launch'), 'launch', 'display.launch.py']),
        launch_arguments={
            'urdf_package': 'erl1',
            'urdf_package_path': PathJoinSubstitution(['urdf', LaunchConfiguration('model')]),
            'rviz_config': LaunchConfiguration('rvizconfig'),
            'jsp_gui': LaunchConfiguration('gui')}.items()
    )

    launchDescriptionObject = LaunchDescription()

    launchDescriptionObject.add_action(gui_arg)
    launchDescriptionObject.add_action(rviz_arg)
    launchDescriptionObject.add_action(model_arg)
    launchDescriptionObject.add_action(urdf)

    return launchDescriptionObject
```

Building your own robot - 2

Let's keep building the differential drive robot by adding the 2 wheels:

```
<link name='left_wheel'>
   <inertial>
     <mass value="5.0"/>
     <origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
     <inertia
         ixx="0.014" ixy="0" ixz="0"
         iyy="0.014" iyz="0"
         izz="0.025"
     />
   </inertial>

   <collision>
     <origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
     <geometry>
       <cylinder radius=".1" length=".05"/>
     </geometry>
   </collision>

   <visual name='left_wheel_visual'>
     <origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
     <geometry>
       <cylinder radius=".1" length=".05"/>
     </geometry>
   </visual>
 </link>
```

```
<link name='right_wheel'>
    <inertial>
      <mass value="5.0"/>
      <origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
      <inertia
          ixx="0.014" ixy="0" ixz="0"
          iyy="0.014" iyz="0"
          izz="0.025"
      />
    </inertial>

    <collision>
      <origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
      <geometry>
        <cylinder radius=".1" length=".05"/>
      </geometry>
    </collision>

    <visual name='right_wheel_visual'>
      <origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
      <geometry>
        <cylinder radius=".1" length=".05"/>
      </geometry>
    </visual>
  </link>
```

Building your own robot - 2

If we want, we could also use some macros (xacro)!

```
        <xacro:property name="length_wheel" value="0.04" />
        <xacro:property name="radius_wheel" value="0.1" />

        <xacro:macro name="default_inertial" params="mass">
                <inertial>
                        <mass value="${mass}" />
                        <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
                        <inertia ixx="0.000526666666667" ixy="0" ixz="0" iyy="0.000526666666667"
iyz="0" izz="0.001"/>
                </inertial>
        </xacro:macro>

    <xacro:macro name="wheel_geometry">
                <geometry>
                        <cylinder length="${length_wheel}" radius="${radius_wheel}"/>
        </geometry>
        </xacro:macro>
```

URDF file: JOINTS

The joint element has two attributes:

name (required) Specifies a unique name of the joint

type (required) Specifies the type of joint, where type can be one of the following:

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. All degrees of freedom are locked. This type of joint does not require the axis, calibration, dynamics, limits or safety_controller.
floating - This joint allows motion for all 6 degrees of freedom.
planar - This joint allows motion in a plane perpendicular to the axis.

URDF file: JOINTS

The joint element has following elements:

<origin> This is the transform from the parent link (origin of the parent geometry) to the child link (origin of the child geometry). If origin is set to 0, the origin of the two links are coincident.

xyz (optional: defaults to zero vector) offset. All positions are specified in meters.

rpy (optional: defaults to zero vector) Represents the rotation around fixed axis: first roll around x, then pitch around y and finally yaw around z. All angles are specified in radians.

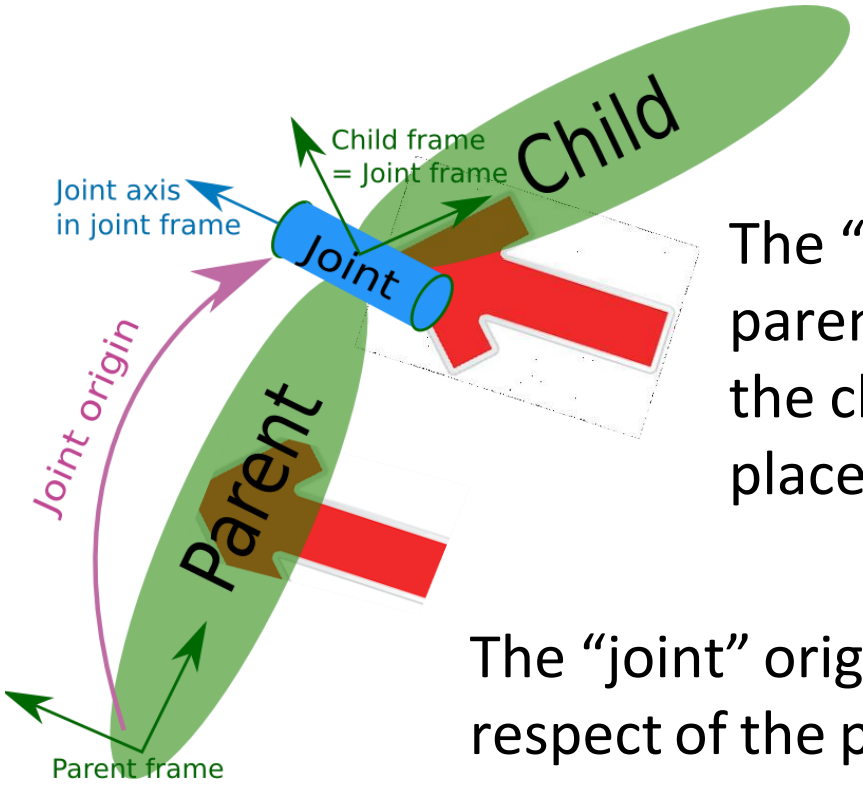
<parent> (required) Parent link name with mandatory attribute:

link The name of the link that is the parent of this link in the robot tree structure.

<child> (required) Child link name with mandatory attribute:

link The name of the link that is the child link.

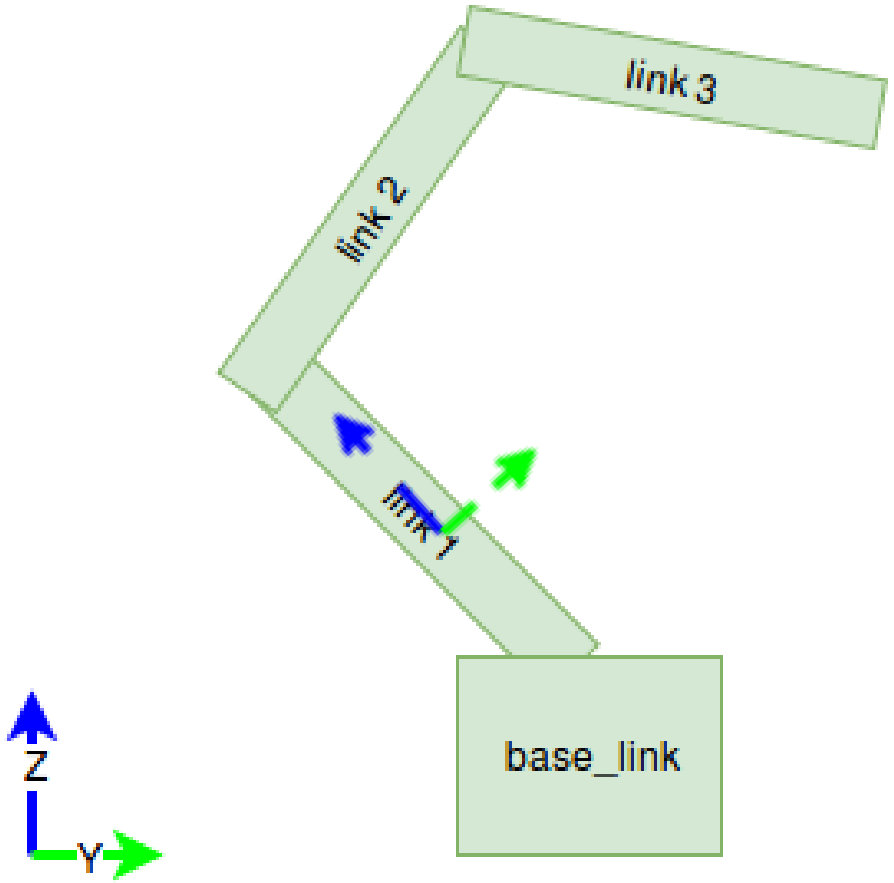
Origin (Links and Joints)



The “link” origin specifies where the joint connecting the parent link with the child link will be fixed with respect of the child link. If the xyz origin is <0 0 0>, the joint will be placed in the geometrical center of the link

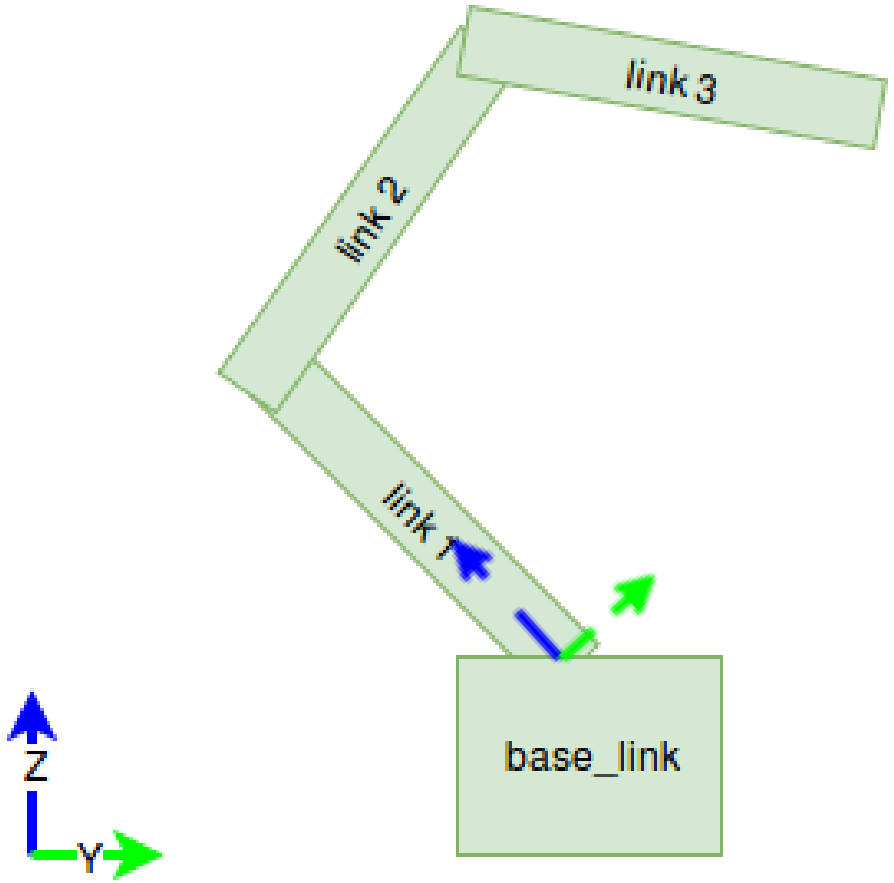
The “joint” origin specifies the relative position of the joint with respect of the parent frame (i.e., the joint associated to the parent link). If the xyz origin is <0 0 0>, the joint associated to the child link will coincide with the joint associated to the parent link

Origin (Links and Joints)



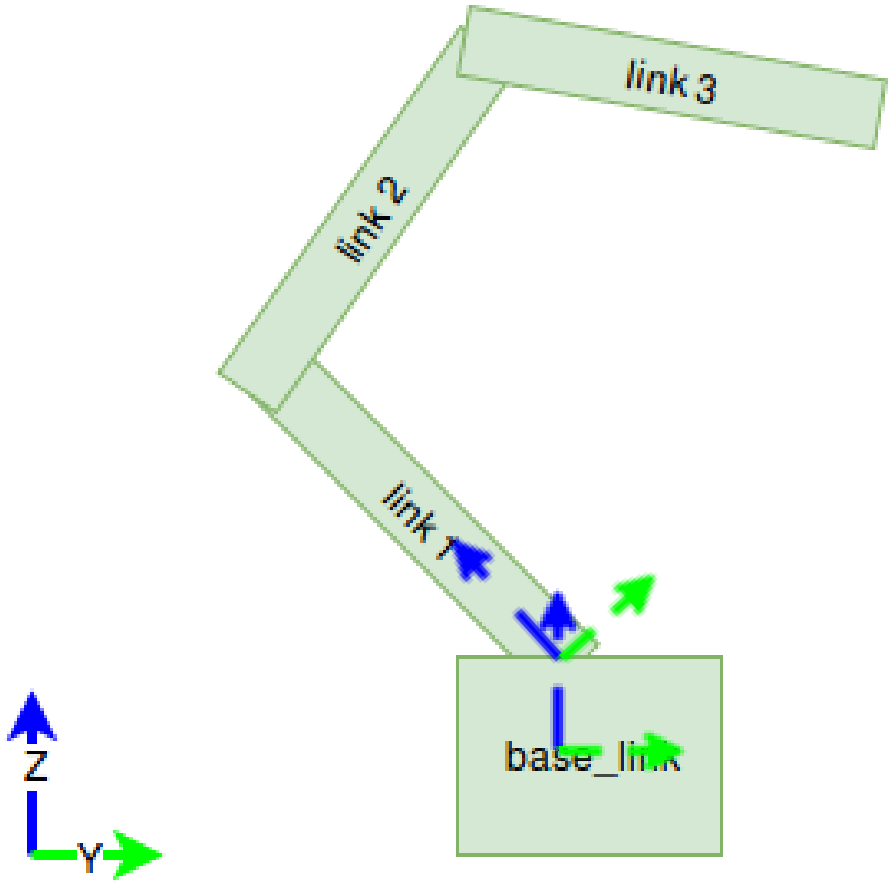
Es. For link 1, the “link” origin should be 0.5*z [link 1], otherwise link1 would rotate around its center.

Origin (Links and Joints)



Es. For link 1, the “link” origin should be 0.5*z [link 1], otherwise link1 would rotate around its center.

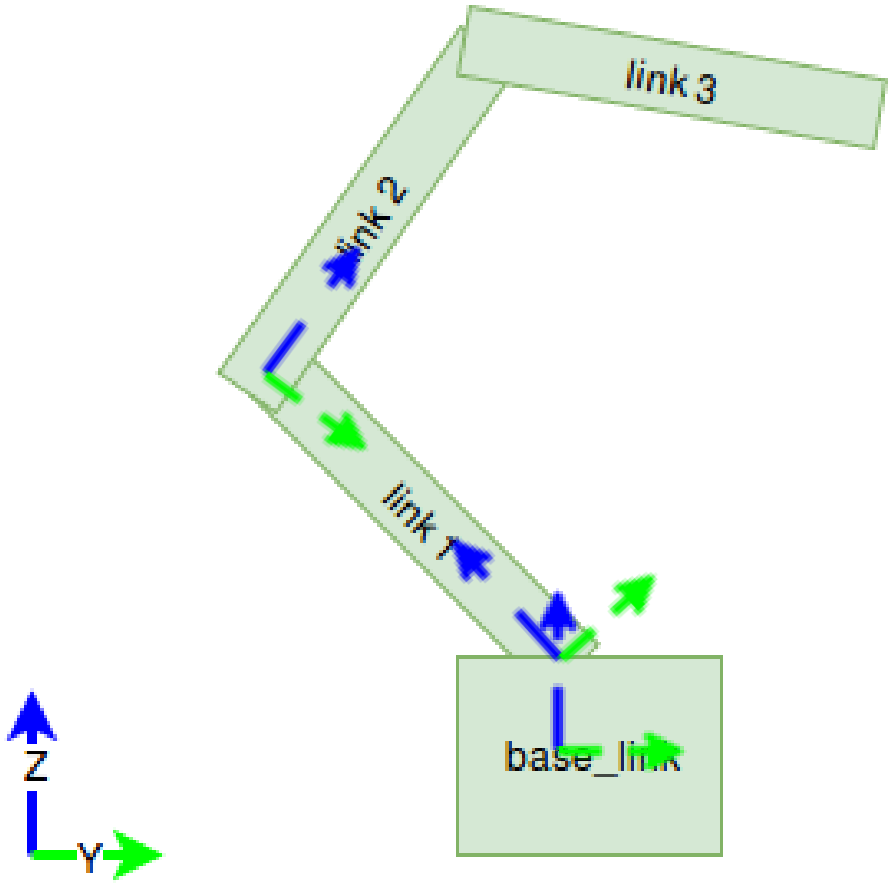
Origin (Links and Joints)



Es. For link 1, the “link” origin should be <0 0 0.5*z[link 1]>, otherwise link1 would rotate around its center.

The “joint” origin, if the “link” origin of base link is <0 0 0>, will be <0 0 0.5*z[base_link]>

Origin (Links and Joints)



Es. For link 1, the “link” origin should be <0 0 0.5*z[link 1]>, otherwise link1 would rotate around its center.

The “joint” origin, if the “link” origin of base link is <0 y 0>, will be <0 0 0.5*z[base_link]>

The same will apply for link2. We will use a “link” origin of <0 0 0.5*z[link 2]>, and a “joint” origin of <0 0 z[link 1]

URDF file: JOINTS

<axis> (optional: defaults to (1,0,0)) The joint axis specified in the joint frame. This is the axis of rotation for revolute joints, the axis of translation for prismatic joints, and the surface normal for planar joints. The axis is specified in the joint frame of reference. Fixed and floating joints do not use the axis field.

xyz (required) components of a vector. The vector should be normalized.

<dynamics> *(optional)* An element specifying physical properties of the joint. These values are used to specify modeling properties of the joint, particularly useful for simulation (e.g., damping, friction)

<limit> (required only for revolute and prismatic joint) An element can contain the following attributes:

lower (optional, defaults to 0). An attribute specifying the lower joint limit (radians for revolute joints, meters for prismatic joints). Omit if joint is continuous.
upper (optional, defaults to 0) An attribute specifying the upper joint limit (radians for revolute joints, meters for prismatic joints). Omit if joint is continuous.
effort (required) An attribute for enforcing the maximum joint effort
velocity (required). An attribute for enforcing the maximum joint velocity. See safety limits.

Types of joints

```
<joint type="continuous"
name="right_wheel_joint">
    <origin xyz="-0.1 -0.15 0" rpy="0 0 0"/>
    <child link="right_wheel"/>
    <parent link="link_chassis"/>
    <axis xyz="0 1 0" rpy="0 0 0"/>
    <limit effort="100" velocity="10"/>
    <dynamics damping="1.0" friction="1.0"/>
  </joint>

<joint type="continuous"
name="left_wheel_joint">
    <origin xyz="-0.1 0.15 0" rpy="0 0 0"/>
    <child link="left_wheel"/>
    <parent link="link_chassis"/>
    <axis xyz="0 1 0" rpy="0 0 0"/>
    <limit effort="100" velocity="10"/>
    <dynamics damping="1.0" friction="1.0"/>
  </joint>
```

In the joint field, we define the name, which must be unique as well. Also, we define the type of joint (fixed, revolute, continuous, floating, or planar), the parent, and the child.

In case of revolute joints, the following limits should be set:
<limit effort =" " lower=" " upper=" " velocity=" "/>

Let's rebuild the workspace and see what we have now!

We can check the tree with:

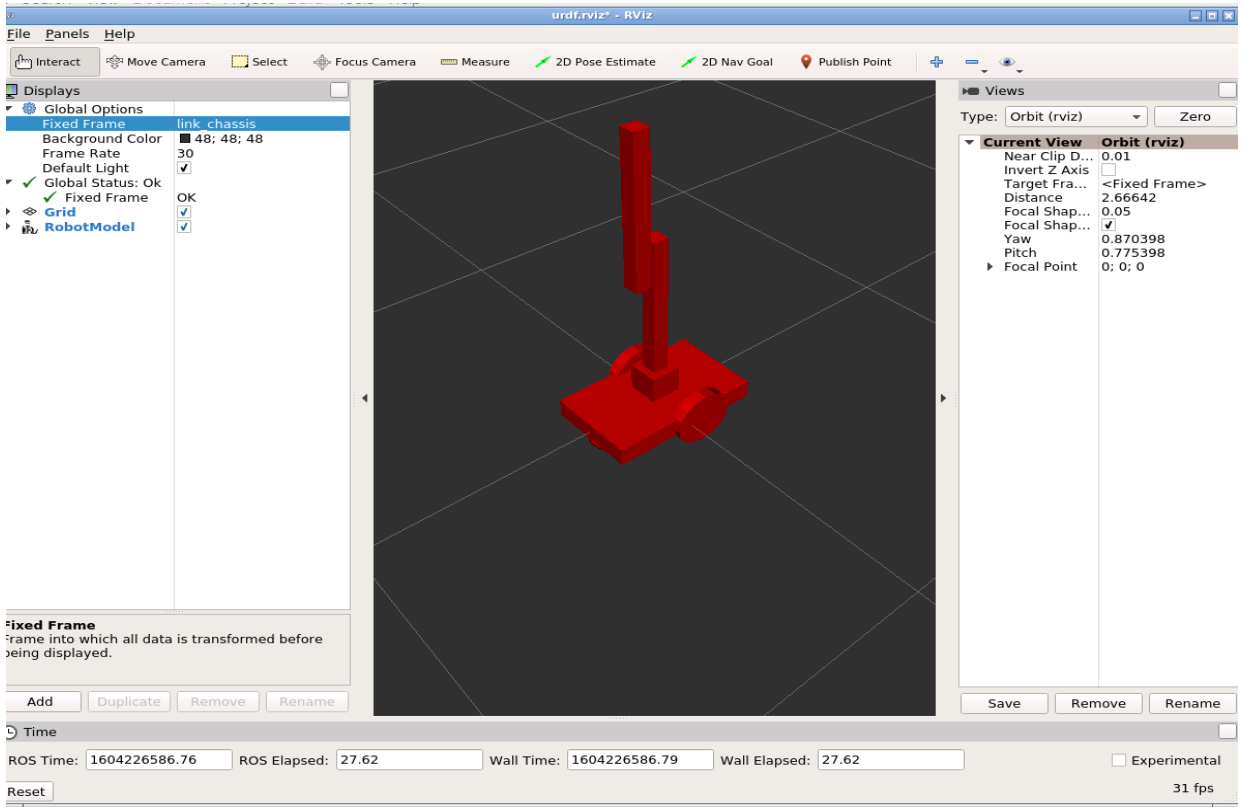
```
ros2 run rqt_tf_tree
rqt_tf_tree
```

Exercise 1

Starting from the robot build so far, add the following links/joints:

- a rotating base (i.e., a continuous joint rotating along the z axis)
- a link connected to the base with a revolute joint, rotating along the x axis (i.e., pitch)
- an additional link, connected to the previous one with a revolute joint (again along the x axis)

Visualize the robot with Rviz



Simulation in Gazebo

Now we have a robot model that we see in RViz, but RViz is just a visualization tool!!

Even if we can rotate some joints with the joint_state_publisher_gui it has nothing to do with the physical simulation. To insert our robot model into the Gazebo simulation environment we have to write a new launch file that spawns the robot through the right services of Gazebo.

Let's create spawn_robot.launch.py in our launch folder.

This launch file will:
- include the *my_launch.py* we have created before!
- start *rviz* with our configuration
- spawn our robot model starting from the urdf we have created
- Start two additional nodes: *robot_state_publisher*, which converts and loads the URDF/xacro into the robot_description topic, providing the transformations between links, and *joint_state_publisher*, which updates rviz accordingly to gz.

Simulation in Gazebo

```
import os
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, IncludeLaunchDescription
from launch.conditions import IfCondition
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.substitutions import LaunchConfiguration, PathJoinSubstitution, Command
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory

def generate_launch_description():

    pkg_erl1= get_package_share_directory('erl1')

    gazebo_models_path, ignore_last_dir = os.path.split(pkg_erl1)
    os.environ["GZ_SIM_RESOURCE_PATH"] += os.pathsep + gazebo_models_path

    rviz_launch_arg = DeclareLaunchArgument(
        'rviz', default_value='true',
        description='Open RViz.'
    )

    world_arg = DeclareLaunchArgument(
        'world', default_value='my_world.sdf',
        description='Name of the Gazebo world file to load'
    )

    model_arg = DeclareLaunchArgument(
        'model', default_value='erl1.urdf',
        description='Name of the URDF description to load'
    )

    # Define the path to your URDF or Xacro file
    urdf_file_path = PathJoinSubstitution([
        pkg_erl1,  # Replace with your package name
        "urdf",
        LaunchConfiguration('model')  # Replace with your URDF or Xacro file
    ])

    world_launch = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(pkg_erl1, 'launch', 'my_launch.py'),
        ),
        launch_arguments={
        'world': LaunchConfiguration('world'),
        }.items()
    )
```

```
# Launch rviz
    rviz_node = Node(
        package='rviz2',
        executable='rviz2',
        arguments=['-d', os.path.join(pkg_erl1, 'rviz', 'urdf.rviz')],
        condition=IfCondition(LaunchConfiguration('rviz')),
        parameters=[
            {'use_sim_time': True},
        ]
    )

    # Spawn the URDF model using the `/world/<world_name>/create` service
    spawn_urdf_node = Node(
        package="ros_gz_sim",
        executable="create",
        arguments=[
            "-name", "my_robot",
            "-topic", "robot_description",
            "-x", "0.0", "-y", "0.0", "-z", "0.5", "-Y", "0.0"  # Initial spawn position
        ],
        output="screen",
        parameters=[
            {'use_sim_time': True},
        ]
    )

    robot_state_publisher_node = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='screen',
        parameters=[
            {'robot_description': Command(['xacro', ' ', urdf_file_path]),
             'use_sim_time': True},
        ],
        remappings=[
            ('/tf', 'tf'),
            ('/tf_static', 'tf_static')
        ]
    )

    joint_state_publisher_gui_node = Node(
        package='joint_state_publisher_gui',
        executable='joint_state_publisher_gui',
    )

    launchDescriptionObject = LaunchDescription()

    launchDescriptionObject.add_action(rviz_launch_arg)
    launchDescriptionObject.add_action(world_arg)
    launchDescriptionObject.add_action(model_arg)
    launchDescriptionObject.add_action(world_launch)
    launchDescriptionObject.add_action(rviz_node)
    launchDescriptionObject.add_action(spawn_urdf_node)
    launchDescriptionObject.add_action(robot_state_publisher_node)
    launchDescriptionObject.add_action(joint_state_publisher_gui_node)

    return launchDescriptionObject
```

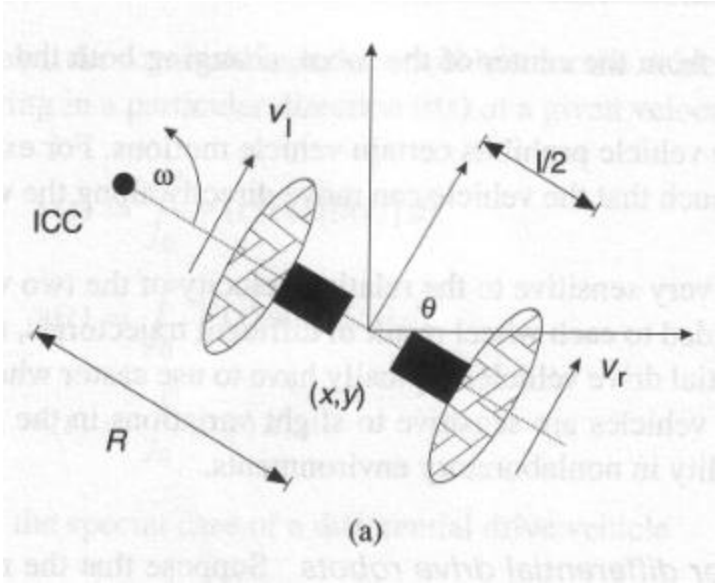
Gazebo Integration

We have now our robot in the physical simulation. We still have to do 2 things!

- adding a Gazebo plugin to move the robot, or sensors to interact with the environment+
- bridging the messages between ROS and Gazebo.

Let's start with the first point.

Example: Differential Drive Plugin.



Gazebo Plugins

The best would be creating a new file,
erl1.gazebo, in the urdf folder.
You need than to include it in the main .urdf file:

```
<xacro:include filename="$(find erl1)/urdf/erl1.gazebo" />
```

```
<?xml version="1.0"?>
<robot>
  <gazebo>
    <plugin
        filename="gz-sim-diff-drive-system"
        name="gz::sim::systems::DiffDrive">
        <!-- Topic for the command input -->
        <topic>/cmd_vel</topic>

        <!-- Wheel joints -->
        <left_joint>left_wheel_joint</left_joint>
        <right_joint>right_wheel_joint</right_joint>

        <!-- Wheel parameters -->
        <wheel_separation>0.3</wheel_separation>
        <wheel_radius>0.1</wheel_radius>

        <!-- Control gains and limits (optional) -->
        <max_velocity>3.0</max_velocity>
        <max_linear_acceleration>1</max_linear_acceleration>
        <min_linear_acceleration>-1</min_linear_acceleration>
        <max_angular_acceleration>2</max_angular_acceleration>
        <min_angular_acceleration>-2</min_angular_acceleration>
        <max_linear_velocity>0.5</max_linear_velocity>
        <min_linear_velocity>-0.5</min_linear_velocity>
        <max_angular_velocity>1</max_angular_velocity>
        <min_angular_velocity>-1</min_angular_velocity>

        <!-- Other parameters (optional) -->
        <odom_topic>odom</odom_topic>
        <tf_topic>tf</tf_topic>
        <frame_id>odom</frame_id>
        <child_frame_id>link_chassis</child_frame_id>
        <odom_publish_frequency>30</odom_publish_frequency>
    </plugin>

    <plugin
        filename="gz-sim-joint-state-publisher-system"
        name="gz::sim::systems::JointStatePublisher">
        <topic>joint_states</topic>
        <joint_name>left_wheel_joint</joint_name>
        <joint_name>right_wheel_joint</joint_name>
    </plugin>
  </gazebo>
</robot>
```

Gazebo Plugins

The *gz-sim-diff-drive-system plugin* is handling the differential drive kinematics, and we will use another plugin *gz-sim-joint-state-publisher-system* to publish joint states from Gazebo to ROS2.

Lists of Gz plugins: https://github.com/gazebosim/gz-sim/tree/gz-sim10/src/systems

Documentation of each plugin is written in the header files, e.g. here.

You can try teleoperating the robot with the teleop plugin in Gazebo!

But still we could not control our robot from ROS2…

Gazebo Plugins

More details here:
https://github.com/gazebosim/ros_gz/tree/ros2/ros_gz_bridge

We have to further update the launch file, by also starting the gz_bridge node!

```
# Node to bridge messages like /cmd_vel and /odom
    gz_bridge_node = Node(
        package="ros_gz_bridge",
        executable="parameter_bridge",
        arguments=[

"/clock@rosgraph_msgs/msg/Clock[gz.msgs.Clock",

"/cmd_vel@geometry_msgs/msg/Twist@gz.msgs.Twist",

"/odom@nav_msgs/msg/Odometry@gz.msgs.Odometry",

"/joint_states@sensor_msgs/msg/JointState@gz.msgs.M
odel",

"/tf@tf2_msgs/msg/TFMessage@gz.msgs.Pose_V"
        ],
        output="screen",
        parameters=[
            {'use_sim_time': True},
        ]
    )
```

We forward the following topics:

•/clock: The topic used for tracking simulation time or any custom time source.

•/cmd_vel: We'll control the simulated robot from this ROS topic.

•/odom: Gazebo's diff drive plugin provides this odometry topic for ROS consumers.

•/joint_states: Gazebo's other plugin provides the dynamic transformation of the wheel joints.

•/tf: Gazebo provides the real-time computation of the robot's pose and the positions of its links, sensors, etc.

remove the joint_state_publisher from the spawn_robot.launch.py, and add gz_bridge_node!

Gazebo Plugins

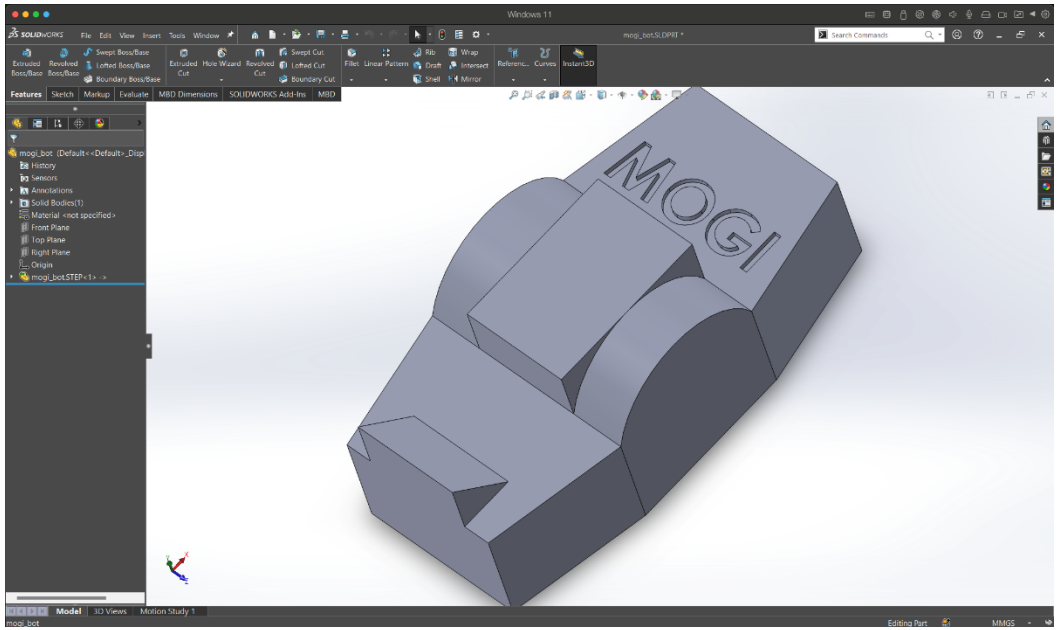
The friction between the wheels and the ground plane can be unrealistic so we can adjust it inside our URDF, let's add the following physical simulation parameters to the end of our .gazebo file before the </robot> tag is closed:

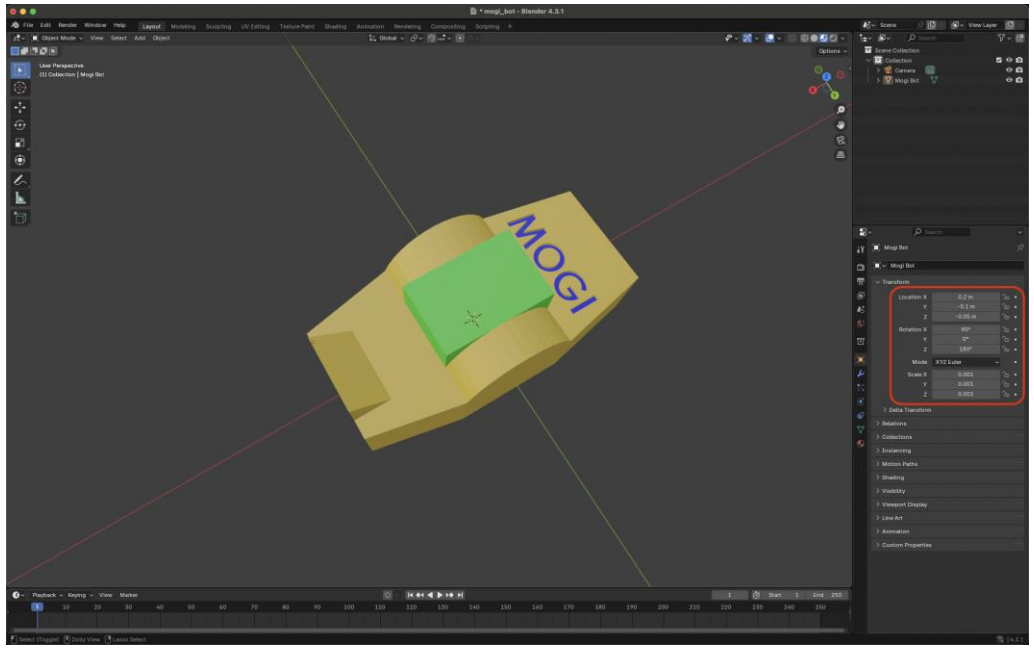
```
<gazebo reference="left_wheel">
    <mu1>0.5</mu1>
    <mu2>0.5</mu2>
    <kp>1000000.0</kp>
    <kd>100.0</kd>
    <minDepth>0.0001</minDepth>
    <maxVel>1.0</maxVel>
  </gazebo>

  <gazebo reference="right_wheel">
    <mu1>0.5</mu1>
    <mu2>0.5</mu2>
    <kp>1000000.0</kp>
    <kd>100.0</kd>
    <minDepth>0.0001</minDepth>
    <maxVel>1.0</maxVel>
  </gazebo>

  <gazebo reference="link_chassis">
    <mu1>0.000002</mu1>
    <mu2>0.000002</mu2>
  </gazebo>
```

3D Models and Textures

The robot can be made visually more appealing with some 3D models.

We can either use .stl files or .dae collada meshes. With collada meshes you can individually color certain areas of meshes (e.g. the tyre, the hub and spokes in case of the wheel). With .stl files we can only assign a single color for the model.





taken from: https://github.com/MOGI-ROS/Week-3-4-Gazebo-basics

3D Models and Textures

You can clone meshes from: https://github.com/CarmineD8/meshes

Put the folder in the ROS2 package. Of course we need to modify the urdf, and the CMakeLists.txt

```
 <collision name="collision_chassis">
      <geometry>
        <mesh filename =
"package://erl1/meshes/mogi_bot.dae"/>
      </geometry>
    </collision>
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename =
"package://erl1/meshes/mogi_bot.dae"/>
      </geometry>
      <material name="blue"/>
    </visual>
```

```
install(DIRECTORY
  launch
  worlds
  rviz
  urdf
  meshes
  DESTINATION share/${PROJECT_NAME}
)
```

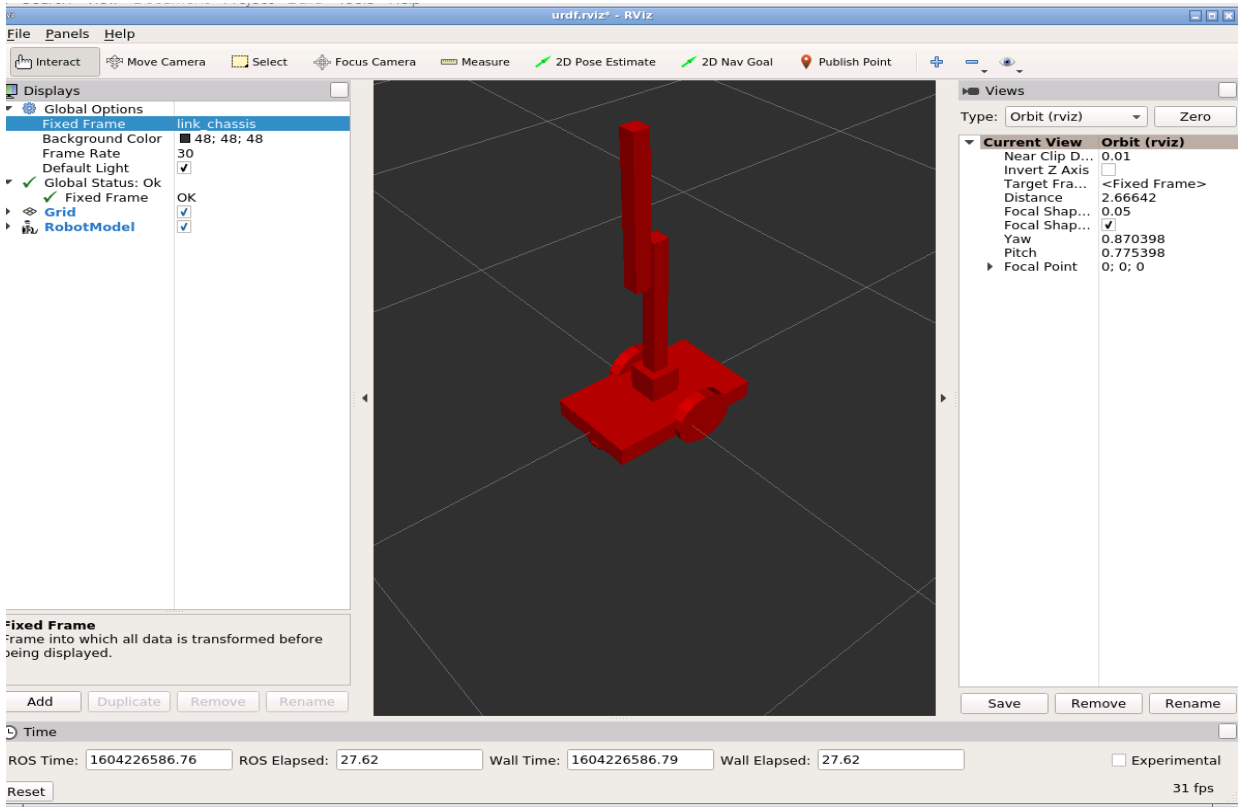
taken from: https://github.com/MOGI-ROS/Week-3-4-Gazebo-basics

Exercise 1

Starting from the robot build so far, add the following links/joints:

- a rotating base (i.e., a continuous joint rotating along the z axis)
- a link connected to the base with a revolute joint, rotating along the x axis (i.e., pitch)
- an additional link, connected to the previous one with a revolute joint (again along the x axis)

Visualize the robot with Rviz



Exercise 2

- Try to create a robot with 4 wheels, and a skid steer drive controller!

  Documentation:
https://gazebosim.org/api/sim/8/classgz_1_1sim_1_1systems_1_1DiffDrive.html

- Add a prismatic joint on top of the link_chassis