Gazebo Simulator: Introduction and Examples

Zhaopeng QIU

December 6, 2013

Abstract

This document presents the powerful physique simulator Gazebo including its installation, applications and some examples. You can access its official website for more detailed tutorials.

1 Introduction

Gazebo is an open-source physique simulator software for 3D kinematic and dynamic simulations which can be easily integrated in ROS (Robot Operating System). Based on its physics and rendering engines, users can model and simulate all kinds of robots, various sensors and environmental objects in a physically defined world. Gazebo can simulate both kinematics and dynamics of rigid bodies articulated with joints. It can also generate realistic sensor feedbacks and dynamic interactions between objects.

Several official versions of Gazebo have been released in recent years. The up-to-date version is Gazebo 2.0.0 which was released in October 8, 2013. This document is based on Gazebo 1.9 so you are highly recommended to install Gazebo 1.9 or higher.

2 Installation

Gazebo is an integrated stack in ROS, but it has also been made a standalone software for ROS recently. It will not be integrated in ROS Hydro or future distributions. So in this section, I introduce the standalone installation of Gazebo and the installation of packages that integrate Gazebo into ROS. This tutorial is based on ROS Groovy. You should have installed beforehand ROS Groovy on your machine as described here before you begin to install Gazebo.

2.1 Install Gazebo on Ubuntu

1. Configure your Ubuntu repositories to allow "restricted", "universe" and "multiverse". You can follow the Ubuntu guide for instructions on doing this.

2. Setup your computer to accept software from *packages.osrfoundation.org* (Attention, the following phrases may contain incorrect symbols when being copied to your terminal):

   Ubuntu Linux 12.04 (precise)

   ```
   #  sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu precise main" > /etc/apt/
    sources.list.d/gazebo-latest.list'
   ```

   Ubuntu Linux 12.10 (quantal)

```
#  sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu quantal main" > /etc/apt/
 sources.list.d/gazebo-latest.list'
```

Ubuntu Linux 13.04 (raring)

```
#  sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu raring main" > /etc/apt/
 sources.list.d/gazebo-latest.list'
```

3. Retrieve and install the keys for the Gazebo repositories.

```
#  wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -
```

4. Update *apt-get* and install Gazebo

```
#  sudo apt-get update
#  sudo apt-get install gazebo
```

2.2 Install gazebo_ros_pkgs

1. Install ROS Groovy if you have not yet installed ROS (see `http://wiki.ros.org/groovy/Installation/Ubuntu`);
2. Remove ROS's old version of Gazebo ROS integration:

```
#  sudo apt-get remove ros-fuerte-simulator-gazebo ros-groovy-simulator-gazebo
```

3. Setup a Catkin workspace if you have not yet made one:

```
#  mkdir -p ~/catkin_ws/src
#  cd ~/catkin_ws/src
#  catkin_init_workspace
#  cd ~/catkin_ws
#  catkin_make
```

4. Then add to your *.bashrc* file a source to the setup scripts:

```
#  echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
```

5. Install *git* if it has not yet been installed:

```
#  sudo apt-get install git
```

6. Download the source code from the *gazebo_ros_pkgs* Github repo:

```
#  cd ~/catkin_ws/src
#  git clone https://github.com/ros-simulation/gazebo_ros_pkgs.git
```

7. You will also need the *ros_control* and *ros_controllers* packages installed on your system from source:

```
#  git clone https://github.com/ros-controls/ros_control.git

#  git clone https://github.com/ros-controls/ros_controllers.git -b groovy-backported-hydro

#  git clone https://github.com/ros-controls/control_toolbox.git

#  git clone https://github.com/ros-controls/realtime_tools.git
```

8. Check for any missing dependencies using rosdep:

```
#  rosdep update
#  rosdep check --from-paths . --ignore-src --rosdistro groovy
```

9. You can automatically install the missing dependencies using rosdep via debian install:

```
#  rosdep install --from-paths . --ignore-src --rosdistro groovy -y
```

10. Build the *gazebo_ros_pkgs*

```
#  cd ~/catkin_ws/
#  catkin_make
```

Remark: though the packages that we have downloaded using *git* in this section are in branch *hydro-devel*, they run well in ROS Groovy. So you don't need to change to other branches.

2.3 Download the tutorial package *tutorial_gazebo*

1. Download the file "*tutorial_gazebo.zip*";
2. Extract files into catkin workspace (for example: $\sim /catkin\_ws$):

```
#  unzip tutorial_gazebo.zip -d ~/catkin_ws/src
```

3. Compile the tutorial package:

```
#  cd ~/catkin_ws
#  catkin_make
```

3 URDF and SDF

In this section I present two description formats used in Gazebo simulations: *URDF* and *SDF*. One can learn the two formats by following their on-line tutorials: URDF tutorial and SDF tutorials. In this section, I will also show with some simple examples how to define your own robot models using *URDF* and *Xacro*.

3.1 URDF and Xacro

Unified Robot Description Format (URDF) is an XML format for representing a robot model. In ROS, there exists a C++ parser for analysing and loading *URDF* files. Unlike *SDF*, *URDF* describes only robot models as well as their plugins.

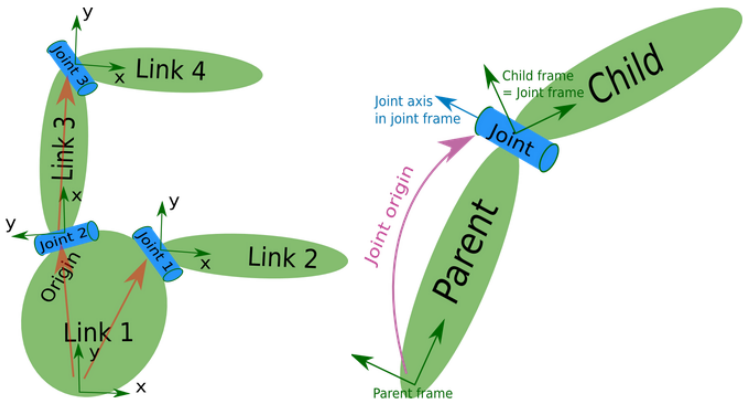

Figure 1: Left:urdf kinematic tree example of a robot model; right: definition of body frames and joint frames

Generally, a robot model consisting of multiple links (rigid bodies) and joints (see Fig. 1) is defined in a file *.urdf* in a hierarchic manner such like:

```
<robot name="robot_model_name">
  <link name="link1">
      <visual>     ......   </visual>
      <collision>   ......  </collision>
  </link>

  <link name="link2">
      <visual>     ......   </visual>
      <collision>   ......  </collision>
  </link>

  <joint name="joint_name" type="fixed">
     <origin ... />
     <parent link="link1"/>
     <child link="link2" />
   </joint>
</robot>
```

A simple example is given in the tutorial package:

```
#  roscd tutorial_gazebo/urdf
#  gedit demo1.urdf
```

You can visualize the robot in rviz:

```
#  roslaunch tutorial_gazebo visu_demo1_urdf.launch
```

Xacro (XML Macros) is an XML macro language. With *Xacro*, you can construct shorter and more readable XML files by using macros that expand to larger XML expressions. You are highly recommended to use *Xacro* in your *URDF* files, especially for complex robot models. You can get very detailed introduction on this web page: `http://wiki.ros.org/xacro`.

A simple example is given in our tutorial package:

```
#  roscd tutorial_gazebo/urdf
#  gedit demo1.xacro
```

You can visualize the robot in rviz:

```
#  roslaunch tutorial_gazebo visu_demo1_xacro.launch
```

Examples in this section are very simple and basic. In next section, I will show very complex applications with *URDF* and *Xacros*.

3.2 SDF

Simulation Description Format (SDF) is an XML format used exclusively in Gazebo for loading and saving information of simulations. *SDF* format encapsulates all the necessary information for a robotic simulator including:

- Scene: Ambient lighting, sky properties, shadows.
- Physics: Gravity, time step, physics engine.
- Models: Collection of links, collision objects, joints, and sensors.
- Lights: Point, spot, and directional light sources.
- Plugins: World, model, sensor, and system plugins.

3.3 SDF vs URDF

URDF can only specify the kinematic and dynamic properties of a single robot in isolation. It can not specify the pose of the robot itself within a world. It is also not a universal description format since it cannot specify joint loops, and it lacks friction and other properties. Additionally, it cannot specify things that are not robots, such as lights, heightmaps, etc.

On the implementation side, the *URDF* syntax breaks proper formatting with heavy use of XML attributes, which in turn makes *URDF* more inflexible. There is also no mechanism for backward compatibility.

SDF solves all of these problems. It is a complete description for everything from the world level down to the robot level. It is highly scalable, and makes it extremely easy to add and modify elements. The *SDF* format is itself described using XML, which facilitates a simple upgrade tool to migrate old versions to new versions. It is also self-descriptive.

4 Simulations with Gazebo

In this section, you will learn step by step how to realize simulations in Gazebo simulator. All the examples in this section are included in package *tutorial_gazebo*.

4.1 Build and modify worlds

The world description file contains all the elements in a simulation, including robots, lights, sensors, and static objects. This file is formatted using *SDF*, and typically has a *.world* extension. During simulations, the Gazebo server (gzserver) reads this file to generate and populate a world. To get a complete tutorial on building Gazebo worlds, you can following the online tutorial.

Firstly, let's open an empty world:

```
#  roslaunch tutorial_gazebo empty_world.launch
```

One can see an empty world shown in Gazebo simulator as in Fig. 2.

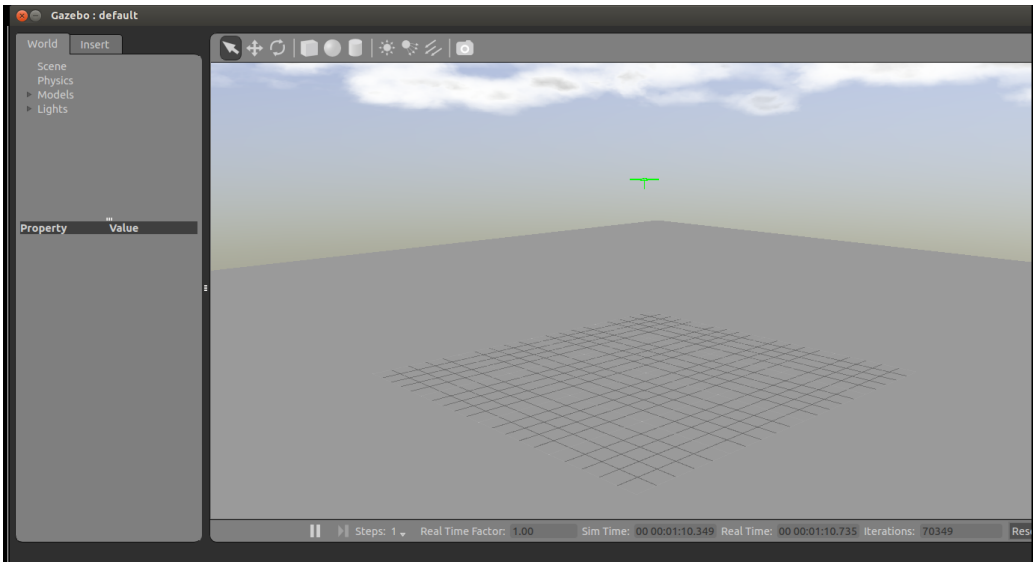

Figure 2: An empty world shown in Gazebo Simulator

Now click on the box button on top of the simulator interface and add a box in the world. You can also use other buttons for inserting objects into the world. Moreover, you can go to **insert** page on the left side of the window and add a series of realistic models into the scene. Then click **File -> Save World as** and save the modified world file in directory *tutorial_ gazebo/worlds* as *box.world*.

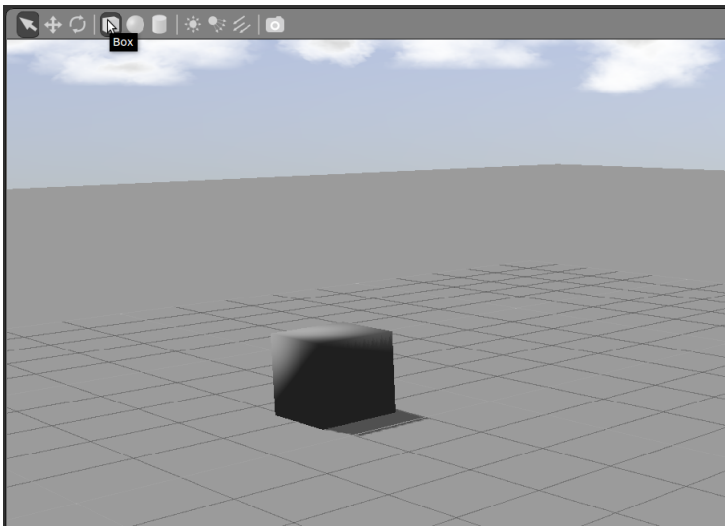

Figure 3: Insert a box into Gazebo world

Let's open the saved world file and see how a world is defined in *SDF*:

```
#  roscd tutorial_gazebo/worlds
#  gedit box.world
```

A world file defined in *SDF* has a structure shown in Fig. 4. In fact, we hardly need to change the physics and rendering descriptions when we build our own simulation world. Usually we just need to add models into the world in order to build the environment for robot simulations. Several examples are given in the *worlds* directory of this tutorial. You are encouraged to model a world file by yourself in order to better understand its structure.

```
▼<sdf version="1.4">
 ▼<world name="default">
   ▶<light name="sun" type="directional">...</light>
   ▶<model name="ground_plane">...</model>
   ▶<physics type="ode">...</physics>
   ▶<scene>...</scene>
   ▶<model name="unit_box_1">...</model>
   ▶<state world_name="default">...</state>
   ▶<gui fullscreen="0">...</gui>
  </world>
 </sdf>
```

Figure 4: World file structure

4.2 Spawn a robot model into simulation

In this tutorial package, you can spawn a Wifibot model into the world.

```
#  roslaunch tutorial_gazebo wifibot_demo.launch
```

Then you can see a Wifobot model shown in Gazebo simulator window as shown in Fig. 5.

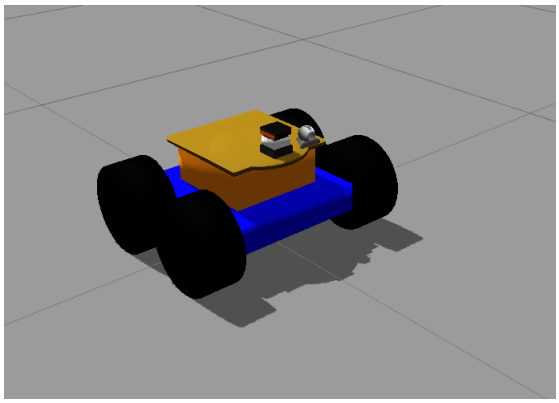

Figure 5: Wifibot in gazebo simulator

Now let's look into the launch file for this simulation example:

```
#  roscd tutorial_gazebo
#  cd launch
#  gedit wifibot_demo.launch
```

In the launch file, you can see the following commands that spawn the robot defined in file *wifibot_ demo.xacro*:

```
<param name="robot_description"
       command="$(find xacro)/xacro.py '$(find tutorial_gazebo)/urdf/wifibot_demo.xacro'" />
<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false" output="screen"
      args="-urdf -model wifibot -param robot_description"/>
```

4.3 Using sensors and controllers

In the previous example, the robot has been inserted into the world. But it cannot sense the world or move around. Thanks to its plugin mechanism, Gazebo makes it easy to simulate sensors and controllers with robot models. A Gazebo plugin is a shared library (.so) that can be integrated to robot model file (in *URDF* or *SDF*).

Gazebo simulator and its ROS packages provide a series of sensor and controller plugins which can be directly applied in your simulations. If you want to add your own controllers or sensors in the simulation, then you should write your own code in Gazebo plugin template and compile it as a shared library. As an example, I provide a sample plugin package *wifibot_ controller* with this tutorial. You can look into it to see how a plugin is coded.

Firstly let's see how to add sensors and controllers to the wifibot shown in Fig. 5.

```
#  roscd tutorial_gazebo
#  cd urdf
#  gedit gazebo_wifibot.urdf.xacro
```

In this file, the first part insert a laser sensor in the origin of the *camera_ pod_ link* of the wifibot model:

```
<gazebo reference="camera_pod_link">
    <sensor type="camera" name="camera1">
      <update_rate>30.0</update_rate>
      <camera name="head">
        ...
     </camera>
     <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
        ...
      </plugin>
    </sensor>
  </gazebo>
```

Generally speaking, in order to add a sensor, you need to specify your sensor type and choose the according shared library file. Moreover, you should specify sensor properties such as its parameters and topics.

The third part add a controller for wifibot model (plugin *libWifibotController.so* is compiled from package *wifibot_ controller*):

```
<gazebo>
  <plugin name="wifibot_controller" filename="libWifibotController.so">
      <alwaysOn>true</alwaysOn>
      <updateRate>100.0</updateRate>
      <topicName>cmd_vel</topicName>
  </plugin>
</gazebo>
```

Now let's see how the sensors and the controller work with wifibot in simulation:

1. In a terminal window, launch the simulation:

```
#  roslaunch tutorial_gazebo  wifibot.launch
```

2. In a second terminal window, open rviz:

```
#  roslaunch tutorial_gazebo visu_wifibot.launch
```

3. Enable keyboard tele-operation by typing the following command in a third terminal window:

```
#  rosrun tutorial_gazebo wifibot_telop
```

   Now you can control motions of Wifibot in the simulation using your keyboard. In *rviz* window, you can visualize data collected by camera and laser sensor plugins. Thus, a complete mobile robot has been simulated. You can use it as a platform for carrying out further tasks such as mapping and navigation. An example for autonomous navigation using *AMCL* and *move_base* is given in this tutorial. One can launch the simulation using the following commands:

```
#  roslaunch tutorial_gazebo wifibot_nav.launch
#  roslaunch tutorial_gazebo visu_wifibot.launch
```

   See official tutorial for move information about navigation using *move_base* and *rviz*.