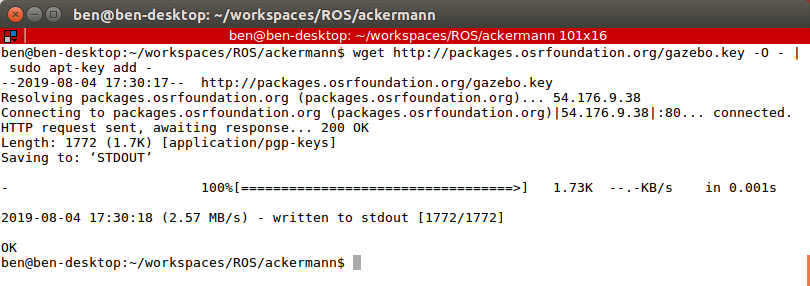
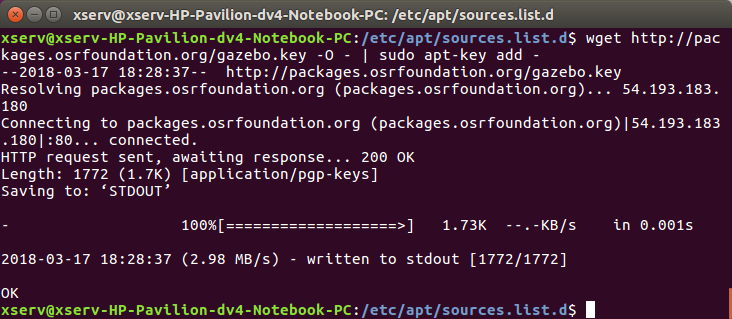
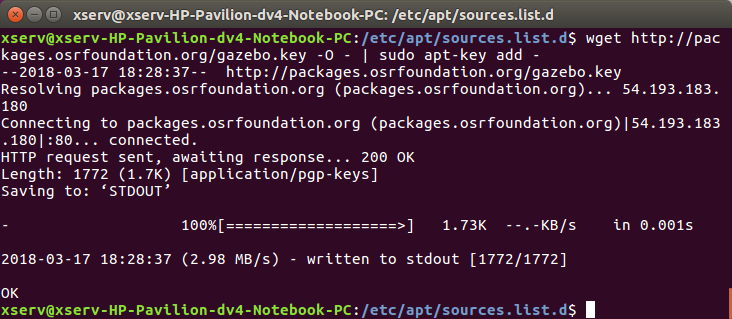
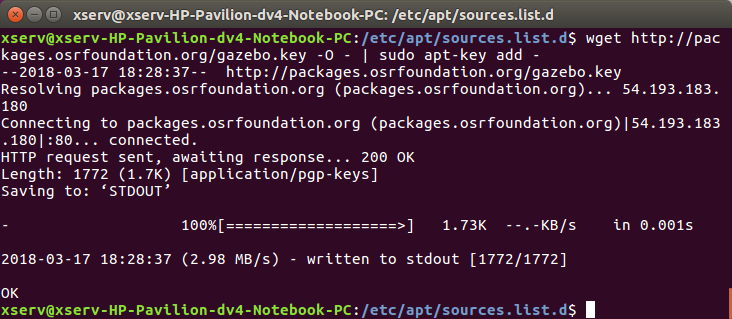
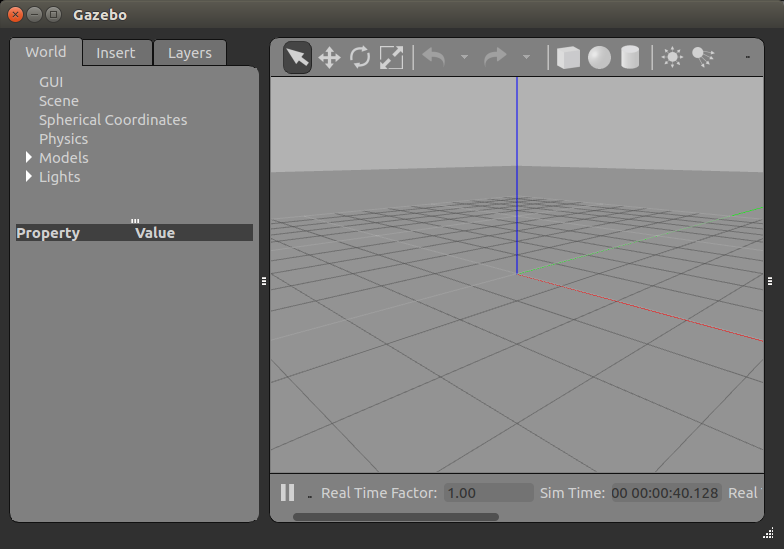
# Chapter 1: Install and Test

# Prerequisites

* Ubuntu 16.04 LTS (Xenial Xerus) Operating System
* [ROS (Kinetic Kame)](http://wiki.ros.org/kinetic/Installation/Ubuntu)

# Install Gazebo

There are a variety of ways to install Gazebo, however, to ensure that Gazebo fully integrates with ROS (Kinetic Kame) on Ubuntu 16.04, we make sure to install Gazebo 7 with the following steps:

1. Add Source to Repository List  
   The first step is to add packages.osrfoundation.org to your Deb sources list. This allows the Deb package manager to access the OSR Foundation package repository, and use apt-get to install Gazebo.   
     
   sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-stable `lsb\_release -cs` main" > /etc/apt/sources.list.d/gazebo-stable.list'  
     
   Verify  
   You can verify that the source has been added by inspecting the list files at /etc/apt/sources.list.d/gazebo-stable.list or by opening Ubuntu Software Center, Edit, Software Sources, and checking that it exists under the Other Software tab.
2. Add the apt-key  
   Next we use wget to download and add the security key to our list:   
     
   wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -  
     
     
     
   
3. Run The Install  
   First, update the Debian software database.  
     
   sudo apt-get update  
     
   Now we run the actual download and installation process.  
     
   sudo apt-get install gazebo7  
     
   Verify  
   To verify the installation is complete, run Gazebo. This should open the GUI as pictured.   
     
   $ gazebo   
     
   

NOTE: ERROR: cannot launch node of type [controller\_manager/spawner]: controller\_manager

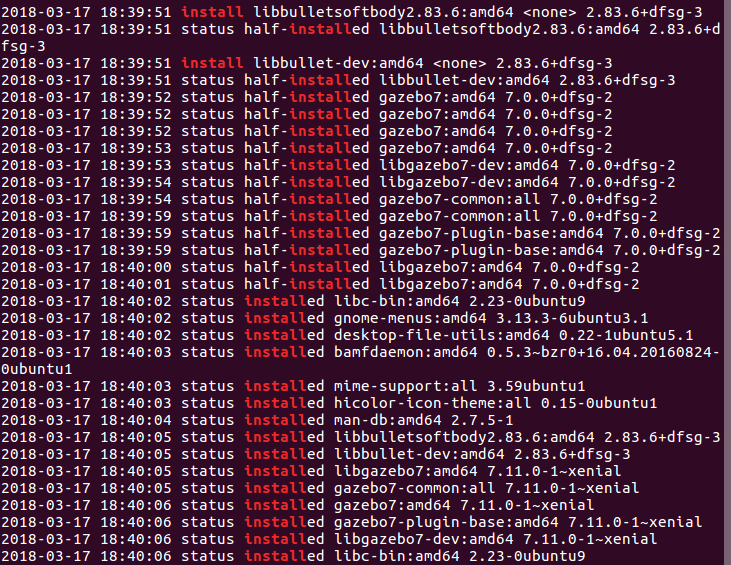
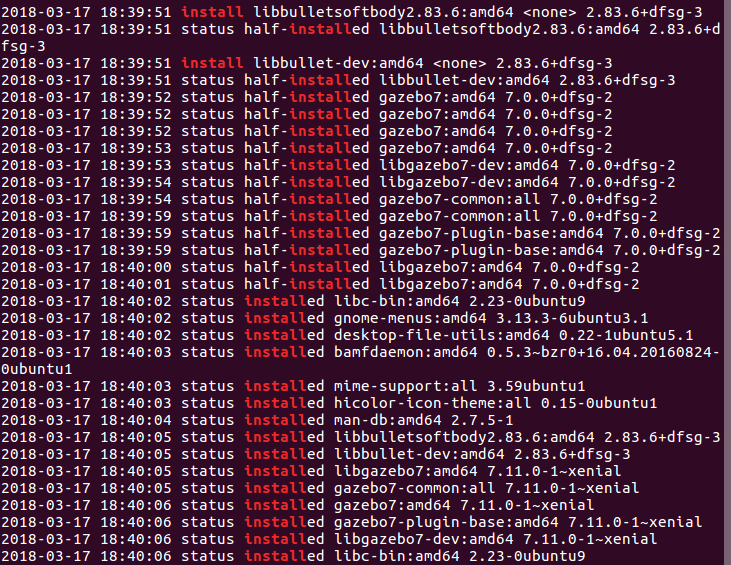
sudo apt install ros-kinetic-controller-manager

NOTE: Install gazebo\_ros\_pkgs,

sudo apt-get install ros-kinetic-gazebo-ros-pkgs ros-kinetic-gazebo-ros-control

# Build Workspace

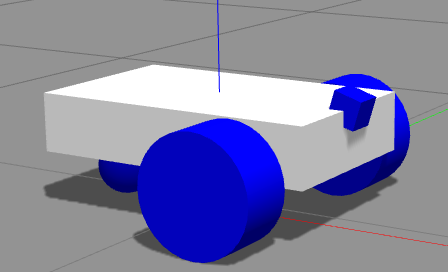
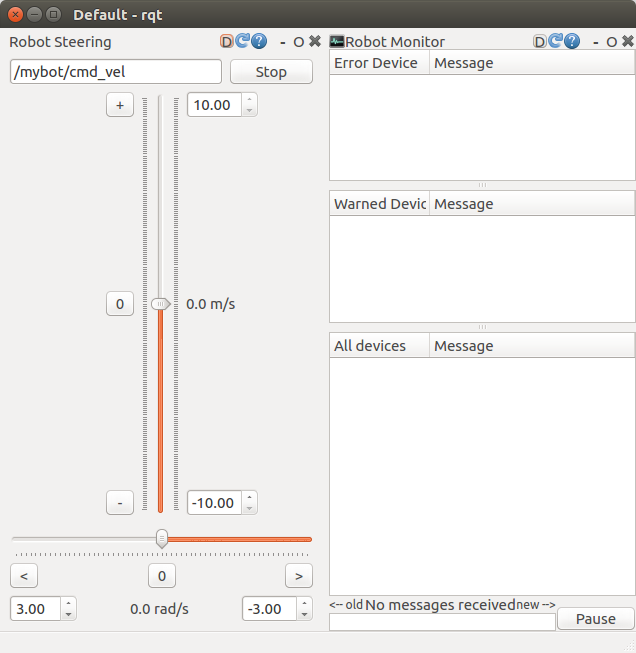
Here we create a standard ROS workspace. This is where we will keep our project code.

1. Install Package Building Tools  
     
   sudo apt-get install python-rosinstall python-rosinstall-generator python-wstool build-essential
2. Create Workspace Folder  
   ROS uses a specific workspace structure. We start by creating a parent directory which will contain all of our other files and folders, including an src/ directory. Code is organized into packages, which will be contained inside [workspace-name]/src/ directory.   
     
   So, assuming you have a main workspace folder, and a folder under it for ROS workspaces, and you name your new ROS workspace chapter1, your initial folder setup will look like this:   
     
   ~/workspaces/ROS/chapter1/src  
     
   This src folder will contain our packages.
3. Download Project Files  
   We use git clone to bring a complete copy of the example files into our workspace.   
     
   ~/workspaces/ROS/chapter1/src$ git clone <https://github.com/benwarrick/into-ROS-Gazebo.git>   
     
   Verify  
   You should now have a set of files and directories matching those on GitHub.
4. Install Dependencies  
   Installing dependencies can be a difficult job, if not for rosdep, which will find and install our needed packages. First we should make sure rosdep is initialized and up to date. Run these commands at the root of the workspace.   
     
   ~/workspaces/ROS/chapter1$ sudo rosdep init  
   ~/workspaces/ROS/chapter1$ rosdep update  
   ~/workspaces/ROS/chapter1$ rosdep install --from-paths src --ignore-src -r -y  
     
   sudo apt-get install ros-kinetic-joint-state-controller   
   sudo apt-get install ros-kinetic-effort-controllers   
   sudo apt-get install ros  
     
     
   VerifyThis command shows you   
     
   ~/workspaces/ROS/chapter1$ grep install /var/log/  
     
     
   
5. Build  
   Now we use catkin\_make to run the actual build. catkin\_make must run in the root of the workspace.   
     
   ~/workspaces/ROS/chapter1$ catkin\_make
6. Setup Terminal  
   Now we setup the terminal using the setup.sh script created for us by catkin\_make in the /devel folder. You'll need to run this command on any new terminal you open for running ROS nodes from this workspace. For regularly used workspaces, you can also add this to your ~/.bashrc file.  
     
   ~/workspaces/ROS/chapter1$ source devel/setup.sh
7. Note: ros-kinetic-velodyne-gazebo-plugins

# Example

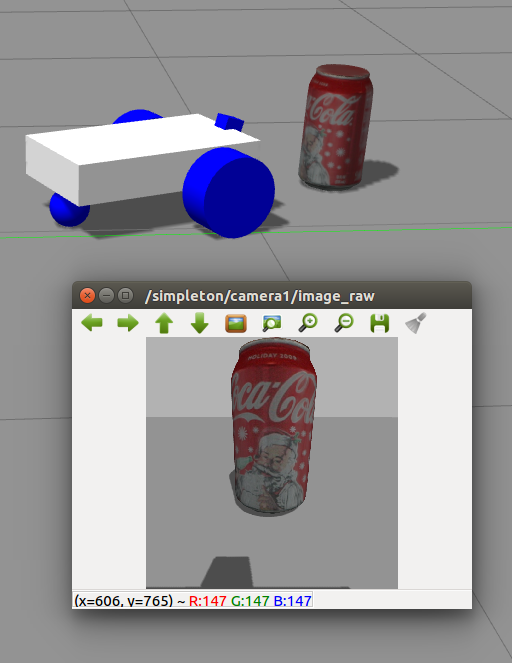
The example we cloned into the workspace is a simple differential drive robot with a forward facing camera. In chapter 3 we will work through the process of building this robot. However, if you're not interested in building robots, and instead only want a simple simulation environment for running computer vision based autonomous vehicle algorithms, this project is a complete solution, although a simple one. This is why I call it Simpleton.

**Differential Drive Vehicle**

1. Open a new terminal and run the setup script.   
     
   ~/workspaces/ROS/chapter1$ source devel/setup.bash  
   1. Launch Gazebo using the bot\_empty.launch file  
         
      ~/workspaces/ROS/chapter1$ roslaunch simpleton\_gazebo bot\_empty.launch  
         
      This will launch Gazebo with the differential drive vehicle, Simpleton, placed on a generic ground plane.  
      
2. To operate the robot, we need to publish messages to it. While Gazebo is still running, open a second terminal window and get the list of currently active topics.   
     
    ~/$ rostopic list  
     
   You should see among the list a topic called */simpleton/cmd\_vel*. We need to publish a Twist message to this queue. It will be received by the Gazebo plugin and consists of a linear and angular velocity component. We could publish this data directly to the topic, like so:   
     
   ~/$ rostopic pub -1 /simpleton/cmd\_vel geometry\_msgs/Tst -- '[0.5,0,0]' '[0,0,.5]'   
      
   You should see the robot make a slow circle. To change the direction, simply publish a new message. A less cumbersome way to control Simpleton is with RQT.   
     
   ~/$ rqt  
     
   This will open a GUI. In the main menu, go to Plugins > Robot Tools > Robot Steering. This will add the two tracker bars. Use these tracker bars to experiment with controlling the vehicle.   
     
   

**Camera**  
  
You may have noticed the small image viewer appearing on your desktop when you launched Simpleton. This is the output from the camera, which publishes the output to topic /simpleton/camera1/image\_raw. If you look at the bot\_empty.launch file, you can see at the bottom where we spawn a ROS node called image\_viewer which subscribes to the camera output topic.

The pictures coming from the camera are likely uninteresting at this point, because we're spawning our robot into an empty world. Let's place an object in the world for the camera to see. In the gazebo window, go to the insert tab and select an object. Here I chose the Coke can.



Camera height and angle is often an important variable when experimenting with computer vision algorithms. Here I have the camera at an even height with the top of the chassis, and angled down .3 radians. Adjusting this height and angle is easy. Take a look at the simpleton.xacro file in the simpleton\_description package. Toward the bottom you'll find the joint connecting the camera to the chassis.

<joint name="camera\_joint" type="fixed">

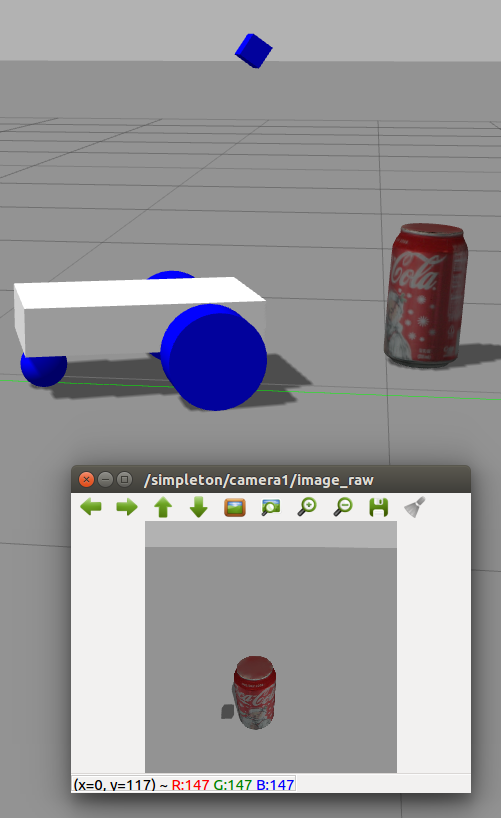
<origin xyz="${chassisLength/2} 0 ${wheelRadius+chassisHeight**+0.5**}" rpy="0 **0.6** 0"/>

<parent link="chassis"/>

<child link="camera"/>

</joint>

Here I've increased the height of the camera by 0.5 meters, and increased the angle to 0.6 radians.



# Chapter 2: ROS Overview

*A Gentle Introduction to ROS – Jason M. O'Kane*  
*Mastering ROS for Robotics Programming – Lentin Joseph*  
*A Systematic Approach to Learning Robot Programming with ROS – Wyatt S. Newman*

1. Nodes
2. Workspace <http://wiki.ros.org/catkin/workspaces>
3. Messaging
4. Example: Subscribe and Publish, Stop on White

Nodes

ROS is a collection of tools for developing robot software. It's not an operating system, but rather a framework that includes such components as hardware abstraction, device control, function libraries, a package management system, and a messaging system. The books listed above offer more details on ROS, so here I will only cover the basic concept necessary for getting started with Gazebo.

An ROS program generally consists of a network of nodes. These nodes communicate though the messaging system or through service calls. The main advantages to this architecture is modularity and re-use. A node that processes camera input, for example, can receive that data through the messaging system via a specific topic. Real cameras or simulated cameras can publish their raw data to that topic. Likewise, a navigation node could publish commands to a vehicle drive system, a small model of the vehicle, or a Gazebo model.

The primary node of any ROS system is roscore, which manages much of the infrastructure, including the messaging system, and so must be started before any other nodes.

Workspace

A standard ROS workspace with one package named "simpleton" will have the following structure:

workspace\_folder

src/

CMakeLists.txt

simpleton/

CMakeLists.txt

Package.xml

build/

devel/

For our purposes we simply need to understand that our packages go into the src folder and the devel folder contains the setup.bash/setup.sh file we need to source.

We can use catkin\_make to automatically build and re-build the workspace. To use catkin\_make, create the workspace folder and the src folder, and run catkin\_make at the root of the workspace – the workspace folder, next to src. The first time you run catkin\_make it will create the build and devel directories, the setup file, and build your executables.

catkin\_create\_pkg (<http://wiki.ros.org/ROS/Tutorials/CreatingPackage>) is a handy script for creating a workspace. It runs in the src directory and will create the package directory along with the package.xml and CMakeLists.txt files partially filled out, including any dependencies you specify. For example, this will create "simpleton" package with the std\_msgs and rospy ROS package dependencies.

~/workspace\_folder/src/$ catkin\_create\_pkg simpleton std\_msgs rospy

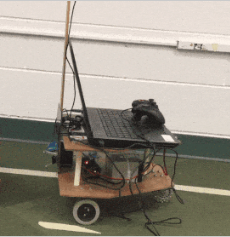
Messaging

One of the main aspects of ROS is the messaging system. It allows nodes to communicate by publishing data to topics and receive data by subscribing to topics. This capability works well with the individual node concept, where separate functionality is contained within nodes.

From our Simpleton example, we can see how the robot receives commands on the topic simpleton/cmd\_vel, and how the camera publishes raw images to simpleton/camera1/image\_raw. In order to control the robot based on camera input, all we need is a clever algorithm that subscribes to the camera's topic and publishes to the robot's topic.

Example: Subscribe and Publish, Stop on White

Let's put it all together now with an example. We can start by cloning another repository into our workspace, l2bot\_examples, which is a set of examples made to run on a robot that is something like a real-life Simpleton – a simple differential drive robot with a platform for a laptop and web camera, intended for the same purposes as Simpleton, testing autonomous vehicle algorithms.



~/workspace\_folder/src/$ git clone https://github.com/LTU-AutoEV/l2bot\_examples.git

We also need to make sure Open CV and the ROS Open CV bridge is installed.

~/$ sudo apt-get install ros-kinetic-cv-bridge

~/$ sudo apt-get install ros-kinetic-vision-opencv

One of the examples included with l2bot\_examples is stop\_on\_white.cpp, which sends the vehicle moving forward until a certain percentage of pixels over a certain whiteness threshold are detected in the input image – at which point the vehicle is stopped.

Using this example for Simpleton is straight forward. We only need it to listen to our camera topic and publish to our cmd\_vel topic. Open the stop\_on\_white.cpp file and do just that. At the top of the file you should see a couple lines similar to the following. Update them to our topics.

// Change this value if you are subscribing to a different camera

#define CAM\_TOPIC "/camera\_1/cam\_pub/image\_raw"

#define TWIST\_PUB "/rb\_drive/rb\_drive/twist\_cmd"

Rebuild the workspace:

~/workspace\_folder/$ catkin\_make

~/workspace\_folder/$ source devel/setup.bash

Let's test the example program to make sure it's working. In one terminal, start roscore.

~/$ roscore

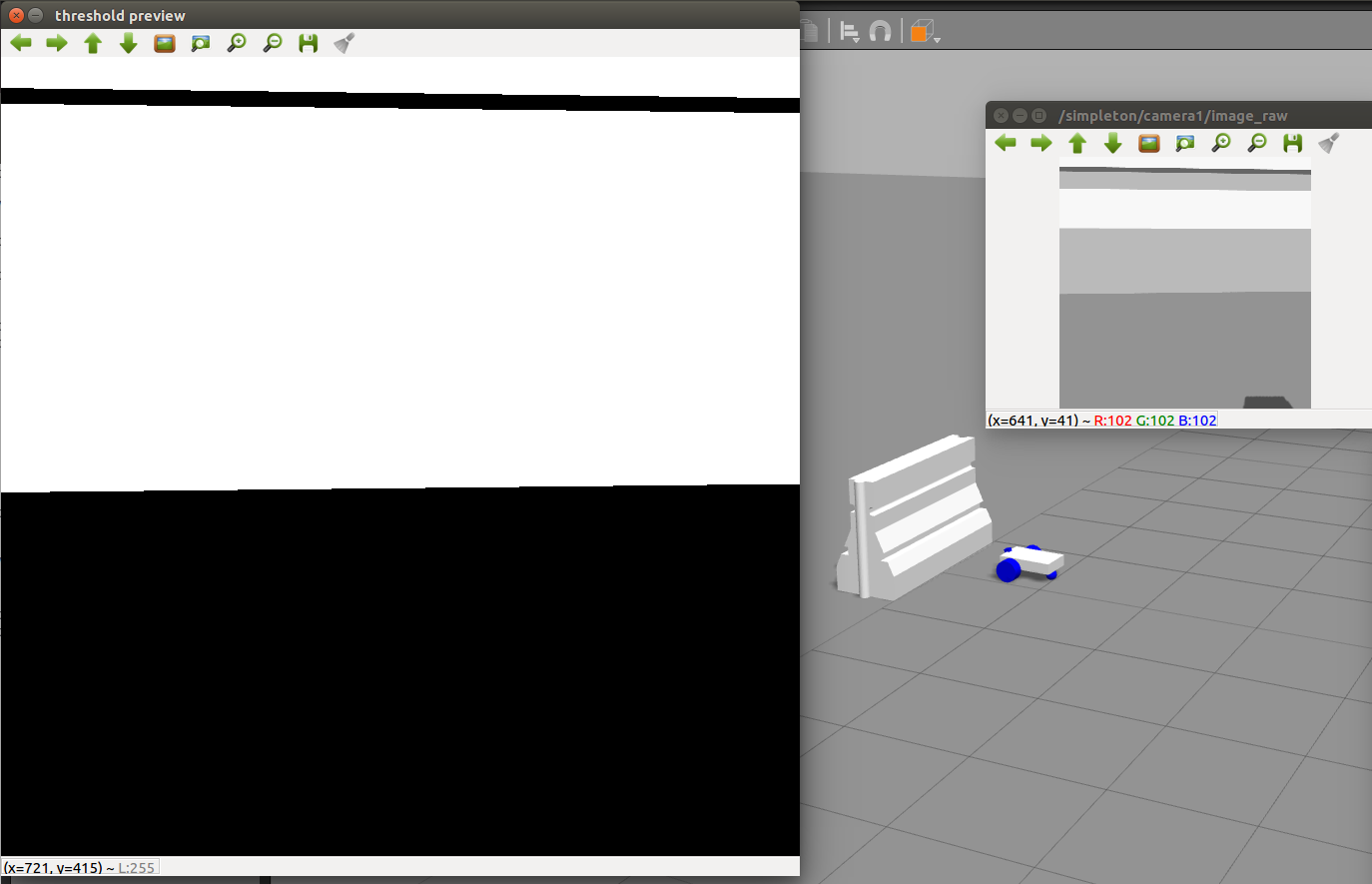
In a second terminal, sourced with our workspace setup, run stop\_on\_white.cpp

~/workspace\_folder/$ rosrun l2bot\_examples op\_on\_white

Now, if you check the list of active topics (rostopic list) you should see our topics, and you should see messages being published to our cmd\_vel topic. A new image viewer will appear, but won't show anything because images are not being published to the topic yet.

~/$ rostopic echo /simpleton/cmd\_vel

If everything seems to work, we can try it out on Simpleton. One good way is to use the object called "drc\_practice\_white\_jersey\_barrier," which is available under the insert tab in Gazebo. Insert this object then drive Simpleton a good distance away. With the white barrier small but centered in Simpleton's camera window, start stop\_on\_white. Simpleton should proceed forward until the white barrier creates too many white pixels in the new image viewer. At this point Simpleton should stop, just in front of the barrier, and the terminal running stop\_on\_white will print "Stopping!" repeatedly.



Chapter 3: Creating a Robot in Gazebo

If you followed the examples in chapter 1, you should currently have our Simpleton differential control wheeled vehicle robot ready to operate in a current workspace. In this chapter, we will work through the process of building old Simpleton, and examine some of the interesting aspects of Gazebo and ROS:

1. Create a workspace
2. Create packages
3. Create a URDF model
4. Run the model in Gazebo
5. Experiment with model settings
6. Adding wheels
7. The Differential Control Plugin
8. XACRO

**1. Create a Workspace**

The first step is to create a new workspace. I'll call mine chapter3. Open a **new** terminal and run the following:

~/workspaces/ROS/$ mkdir chapter3

~/workspaces/ROS/$ cd chapter3

~/workspaces/ROS/$ mkdir src

~/workspaces/ROS/$ catkin\_make

~/workspaces/ROS/$ source devel/setup.bash

At this point it's a good idea to examine your ROS\_PACKAGE\_PATH. Sourcing the setup.bash file does a number of things, one of which is updating this environment variables. Sometimes ROS will add paths to packages outside of your workspace, which can cause a little confusion if you have packages in different work spaces with the same name. For example, if you downloaded our Simpleton robot and you are following the steps in this chapter.

If you started a new session, your path should include only the main ROS share folder and a path to the src in this workspace. If it includes a path to other workspaces, it's wise to remove them.

~/workspaces/ROS/$ echo $ROS\_PACKAGE\_PATH

/home/ben/workspaces/ROS/chapter3/src:/home/ben/workspaces/l2bot\_1/src:/home/ben/workspaces/mybot/src:/opt/ros/kinetic/share

~/workspaces/ROS/$ export ROS\_PACKAGE\_PATH=/home/ben/workspaces/ROS/chapter3/src:/opt/ros/kinetic/share

~/workspaces/ROS/$ echo $ROS\_PACKAGE\_PATH /home/ben/workspaces/ROS/chapter3/src:/opt/ros/kinetic/share

Note: Initialize workspace.

**2. Create Packages**

Now we can create our packages, simpleton\_description and simpleton\_gazebo. In simpleton\_description we will keep everything specific to the creation of our robot. simpleton\_description for contain everything specific to our Gazebo worlds and the launch file that spawn our robot into those worlds.

~/workspaces/ROS/chapter3/src$ catkin\_create\_pkg simpleton\_description

Created file simpleton\_description/CMakeLists.txt

Created file simpleton\_description/package.xml

Successfully created files in /home/ben/workspaces/ROS/chapter3/src/simpleton\_description. Please adjust the values in package.xml.

~/workspaces/ROS/chapter3/src$ catkin\_create\_pkg simpleton\_gazebo

Created file simpleton\_gazebo/package.xml

Created file simpleton\_gazebo/CMakeLists.txt

Successfully created files in /home/ben/workspaces/ROS/chapter3/src/simpleton\_gazebo. Please adjust the values in package.xml.

**3. Create URDF Model**

URDF XML Element Documentation: <http://wiki.ros.org/urdf/XML/>

Now we can begin! Let's start by going into the simpleton\_description package. Create a folder called urdf, and inside it create a test file called simpleton.urdf with the following XML content.

<?xml version="1.0"?>

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

<!-- Base link -->

<link name="footprint" />

<!-- Chassis -->

<joint name="base\_joint" type="fixed">

<parent link="footprint"/>

<child link="chassis"/>

</joint>

<link name="chassis">

<collision>

<origin xyz="0 0 0.3" rpy="0 0 0"/>

<geometry>

<box size="1 0.2 0.1"/>

</geometry>

</collision>

<visual>

<origin xyz="0 0 0.3" rpy="0 0 0"/>

<geometry>

<box size="1 0.2 0.1"/>

</geometry>

<material name="orange"/>

</visual>

<inertial>

<mass value="1"/>

<origin xyz="0 0 0.3" rpy="0 0 0"/>

<inertia ixx="0.4" ixy="0.1" ixz="0.1" iyy="0.4" iyz="0.1" izz="0.2"/>

</inertial>

</link>

<gazebo reference="chassis">

<!--Stiffness -->

<kp>1000000.0</kp>

<!--Dampening-->

<kd>0.1</kd>

<dampingFactor>0</dampingFactor>

<material>Gazebo/White</material>

<selfCollide>true</selfCollide>

<turnGravityOff>false</turnGravityOff>

<mu1 value="0.0"/>

<mu2 value="0.0"/>

<fdir1 value="0 0 0"/>

</gazebo>

</robot>

This XML defines two "links," within the context of a URDF model, a link being something like an object. The first link is the base link, necessary for some things we'll see later in this chapter. The second link, called the chassis, is a rectangle which will serve as such for our robot, and contains three child elements: Collision, Visual, and Inertia. Collision defines the Link's physical space and Visual it's visible material, so they're mostly the same with exception of the Material element of the Visual element in this example.

Lastly, we have the gazebo element, referencing our Chassis link. Naturally this one describes how the link should behave in Gazebo. We can launch this simple object into Gazebo in order to experiment with and understand these Gazebo elements, but first we need to create a world file and a launch file in our simpleton\_gazebo package.

simpleton\_gazebo/worlds/empty.world

<?xml version="1.0" ?>

<sdf version="1.4">

<world name="default">

<!-- A global light source -->

<include>

<uri>model://sun</uri>

</include>

<!-- A ground plane -->

<include>

<uri>model://ground\_plane</uri>

</include>

</world>

</sdf>

Simpleton\_gazebo/launch/bot\_empty.launch

<launch>

<!-- these are the arguments you can pass this launch file, for example paused:=true -->

<arg name="paused" default="true"/>

<arg name="use\_sim\_time" default="true"/>

<arg name="gui" default="true"/>

<arg name="headless" default="false"/>

<arg name="debug" default="false"/>

<!-- Load the URDF into the ROS Parameter Server -->

<param name="robot\_description"

command="$(find xacro)/xacro.py '$(find simpleton\_description)/urdf/simpleton.urdf'" />

<include file="$(find gazebo\_ros)/launch/empty\_world.launch">

<arg name="world\_name" value="$(find simpleton\_gazebo)/worlds/empty.world"/>

<arg name="debug" value="$(arg debug)" />

<arg name="gui" value="$(arg gui)" />

<arg name="paused" value="$(arg paused)"/>

<arg name="use\_sim\_time" value="$(arg use\_sim\_time)"/>

<arg name="headless" value="$(arg headless)"/>

</include>

<!-- Run a python script to the send a service call to gazebo\_ros to spawn a URDF robot -->

<node name="urdf\_spawner" pkg="gazebo\_ros" type="spawn\_model" respawn="false" output="screen"

args="-urdf -model simpleton -param robot\_description"/>

</launch>

We will learn more about creating worlds in a later chapter and will likewise get into more detail with the launch file later. Right now the important thing is to note where our .urdf and .world files are being specified and found using ROS's package find capabilities, which allows us to avoid having static paths.

**4. Launching the Model into Gazebo**

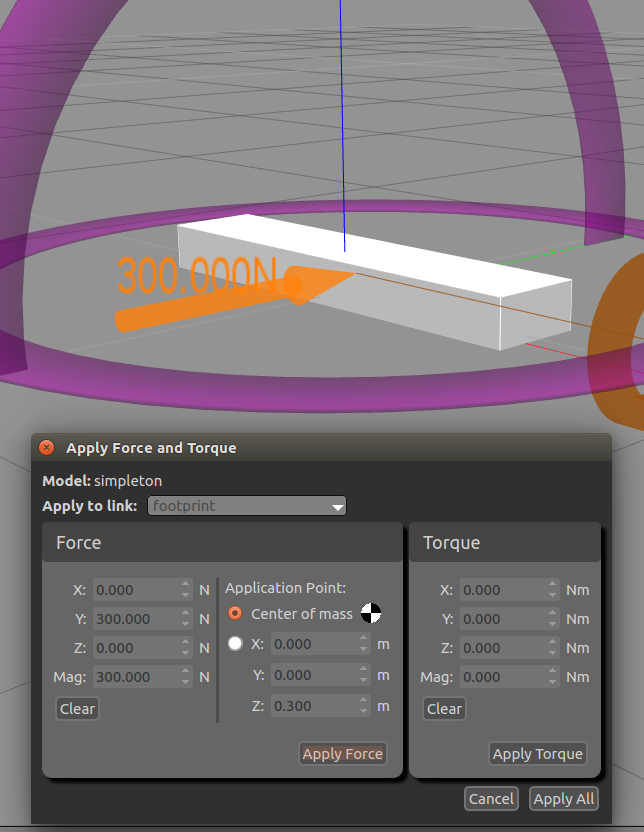
With our files created, enter your terminal and call the launch file. Remember to source the devel/setup.bash file if you're using a new terminal. Also, ROS does autocomplete with the tab key, which is very helpful in finding launch files and correctly spelling package names.

~/workspaces/ROS/chapter3/src$ ga simpleton\_gazebo bot\_empty.launch

If everything goes well, you should see a rectangle floating in midair. It's floating because we specified a positive z coordinate in the position element of the Link element, and it hasn't fallen yet because we specified the pause argument in our launch file as 'true'. We can un-pause the simulation by hitting the play button at the bottom of the Gazebo screen, causing the chassis to fall to the ground.

**5. Experiment with Model Settings**

Right click on the chassis and select 'Apply Force/torque'. This opens a GUI that allows us to set and execute as complicated of a force on the object as we have patience to design. It would pay to experiment with this feature in order to see how the object currently behaves. Try pushing it around or flipping it up on its end. Once you've had enough, go back to the terminal from which you launched our launch file and stop the program with ctrl+C.



Now let's make a change to one of the model settings and re-launch the model and perform the same force/torque experiments to see the difference. I would suggest trying each of the following settings one at a time to get a practical idea of what they do. You can achieve a more rigorous understanding of these setting from an old physics book or numerous online resources, so I won't try to do that here, but some of the more interesting settings to play with are as follows:

*link > inertial > inertia*  
This is the inertia matrix. I'll show you an easy was to automatically set these up a little later when we learn how to put math into our model description. For now, you can make small changes and see how the model bounces around very differently according to these settings.

*gazebo > mu1 & mu2*  
This is where you set the friction. This one is particularly fun. Setting it to zero allows the block to slide indefinitely when force is applied.

6. Adding Wheels

Now that we know everything about Gazebo physics, lets add some wheels to our chassis. The sizes I specify here are a little out of normal proportion, but I think it makes it a bit easier to see and understand how things work in Gazebo. Add the following left and right wheels to your .urdf file.

<!-- Left Wheel - Start -->

<link name="left\_wheel">

<collision>

<origin xyz="0 0 0" rpy="0 1.5708 1.5708" />

<geometry>

<cylinder length="0.1" radius="0.3"/>

</geometry>

</collision>

<visual>

<origin xyz="0 0 0" rpy="0 1.5708 1.5708" />

<geometry>

<cylinder length="0.1" radius="0.3"/>

</geometry>

<material name="black"/>

</visual>

<inertial>

<origin xyz="0 0 0" rpy="0 1.5708 1.5708" />

<mass value="0.2"/>

<cylinder\_inertia m="0.2" r="0.3" h="0.1"/>

<inertia ixx="0.4" ixy="0.1" ixz="0.1" iyy="0.4" iyz="0.1" izz="0.2"/>

</inertial>

</link>

<gazebo reference="left\_wheel">

<mu1 value="2.0"/>

<mu2 value="2.0"/>

<kp value="10000000.0" />

<kd value="1.0" />

<fdir1 value="0 1 0"/>

<material>Gazebo/Blue</material>

</gazebo>

<joint name="left\_wheel\_hinge" type="continuous">

<parent link="chassis"/>

<child link="left\_wheel"/>

<origin xyz="0.1 -0.15 0.3" rpy="0 0 0" />

<axis xyz="0 1 0" rpy="0 0 0" />

<limit effort="100" velocity="100"/>

<dynamics damping="0.0" friction="0.0"/>

</joint>

<!-- Left Wheel - End -->

<!-- Right Wheel - Start -->

<link name="right\_wheel">

<collision>

<origin xyz="0 0 0" rpy="0 1.5708 1.5708" />

<geometry>

<cylinder length="0.1" radius="0.3"/>

</geometry>

</collision>

<visual>

<origin xyz="0 0 0" rpy="0 1.5708 1.5708" />

<geometry>

<cylinder length="0.1" radius="0.3"/>

</geometry>

<material name="black"/>

</visual>

<inertial>

<origin xyz="0 0 0" rpy="0 1.5708 1.5708" />

<mass value="0.2"/>

<cylinder\_inertia m="0.2" r="0.3" h="0.1"/>

<inertia ixx="0.4" ixy="0.1" ixz="0.1" iyy="0.4" iyz="0.1" izz="0.2"/>

</inertial>

</link>

<gazebo reference="right\_wheel">

<mu1 value="2.0"/>

<mu2 value="2.0"/>

<kp value="10000000.0" />

<kd value="1.0" />

<fdir1 value="0 1 0"/>

<material>Gazebo/Blue</material>

</gazebo>

<joint name="right\_wheel\_hinge" type="continuous">

<parent link="chassis"/>

<child link="right\_wheel"/>

<origin xyz="0.1 0.15 0.3" rpy="0 0 0" />

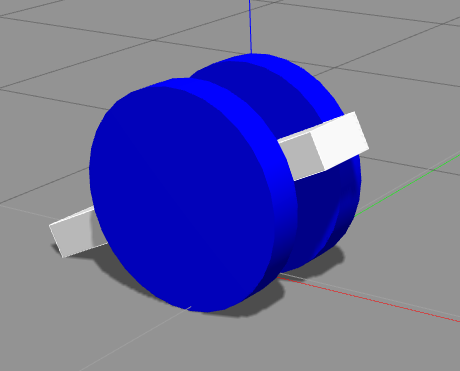
<axis xyz="0 1 0" rpy="0 0 0" />

<limit effort="100" velocity="100"/>

<dynamics damping="0.0" friction="0.0"/>

</joint>

<!-- Right Wheel - End -->



Most of the wheel description should look familiar by now, with exception to the joint element. The joint element is what creates a physical relationship between two link, allowing it to function as a joint. You specify the parent and child links, in the case the chassis and wheel, along with the location of the joint and its axis of movement. You can also set friction and dampening for the joint.

This configuration is pretty fun to play with. If I were you I would take a break and do some experiments with joint and link settings, and also with the variety of ways you can apply force. Also, realize you can have several force/torque GUI windows open at once. So you could, for example, have a GUI opened for each wheel...

**7. The Differential Control Plugin**

Now that we have a good understanding of how this robot works in the simulation, we'll go ahead and give it a drive system. One of the simplest drive system for robots is differential drive, where power is distributed to each or the two wheels and steering is accomplished by varying the speed between the two wheels.

For us, differential drive is made even simpler because ROS already has a standard differential drive Gazebo plugin. All we have to do is give each wheel a transmission and add the plugin. The plugin will specify a ROS topic which will accept a Twist message, consisting of a forward velocity and angular velocity component. The plugin, based on the wheel diameter and separation that we specify, will automatically calculate the force it will apply to each transmission element. So let's add these two components to our .urdf file.

<transmission name="left\_trans">

<type>transmission\_interface/SimpleTransmission</type>

<joint name="left\_wheel\_hinge">

<hardwareInterface>EffortJointInterface</hardwareInterface>

</joint>

<actuator name="leftMotor">

<hardwareInterface>hardwareInterface/EffortJointInterface</hardwareInterface>

<mechanicalReduction>10</mechanicalReduction>

</actuator>

</transmission>

<transmission name="right\_trans">

<type>transmission\_interface/SimpleTransmission</type>

<joint name="right\_wheel\_hinge">

<hardwareInterface>EffortJointInterface</hardwareInterface>

</joint>

<actuator name="rightMotor">

<hardwareInterface>hardwareInterface/EffortJointInterface</hardwareInterface>

<mechanicalReduction>10</mechanicalReduction>

</actuator>

</transmission>

<gazebo>

<plugin name="differential\_drive\_controller" filename="libgazebo\_ros\_diff\_drive.so">

<legacyMode>false</legacyMode>

<alwaysOn>true</alwaysOn>

<updateRate>100</updateRate>

<leftJoint>left\_wheel\_hinge</leftJoint>

<rightJoint>right\_wheel\_hinge</rightJoint>

<wheelSeparation>0.2</wheelSeparation>

<wheelDiameter>0.6</wheelDiameter>

<torque>2</torque>

<commandTopic>simpleton/cmd\_vel</commandTopic>

<odometryTopic>simpleton/odom\_diffdrive</odometryTopic>

<odometryFrame>odom</odometryFrame>

<robotBaseFrame>footprint</robotBaseFrame>

</plugin>

</gazebo>

Let's re-launch our robot and take a look at what's changed. You might not notice anything about the robot itself, so let's take a look and see if the topic has become available.

~$ rostopic list

/clock

/gazebo/link\_states

/gazebo/model\_states

/gazebo/parameter\_descriptions

/gazebo/parameter\_updates

/gazebo/set\_link\_state

/gazebo/set\_model\_state

/rosout

/rosout\_agg

/simpleton/cmd\_vel

/simpleton/odom\_diffdrive

/tf

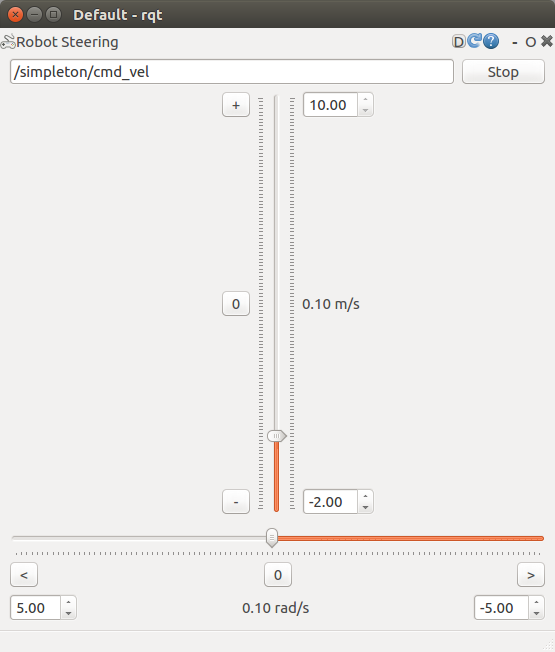
Hopefully you see a similar list that includes /simpleton/cmd\_vel. This is the topic we have specified for the command input. Through a second terminal we can publish a command to this topic like so...

~$ rostopic pub /simpleton/cmd\_vel geometry\_msgs/Twist '{linear: {x: 0.1, y: 0.0, z: 0.0}, angular: {x: 0.0,y: 0.0,z: 0.0}}'

… which will cause the robot to move forward slowly. However, this is pretty cumbersome. So let's launch RQT:

~$ rqt

This will open a GUI. Go to the menu and select Plugins > Robot Tools > Robot Steering. This will display a set of controls – forward velocity m/s and angular speed rad/s. Specify our topic in the text box and give it a try.



Hopefully our robot is running around. Depending on the inertia settings, this configuration can be pretty stable but also fun. With the wrong settings it can behave pretty crazy, which is also fun. But you're probably interested in having a practical robot that will serve as a sensor platform for testing autonomous vehicle algorithms, so let's move directly toward this.

**8. XACRO**

Using xacro, we can split our robot description file up into reusable component files and also create variables and use math. So let's split Simpleton into three different file: The primary simpleton.xacro, a macro.xacro for our reusable xml macros, and a simpleton.gazebo for all Gazebo specific components.

Step 1 – Create useful properties:

* Properties should go in the header of simpleton.xacro like so:
* <xacro:property name="propertyName" value="value"/>
* And can be referenced in the XML with "${propertyName}" , which can also include math: "${propertyName\*2}"

Step 2 – simpleton.gazebo:

* Create a copy of simpleton.urdf called simpleton.xacro.
* Create a new file called simpleton.gazebo containing the XML declaration and the robot tag specifying the xacro namespace.
* Move the <gazebo>…</gazebo> XML objects into simpleton.gazebo file inside the robot tag.
* Test the simpleton.xacro file and make sure that the robot still launches and behaves correctly in Gazebo

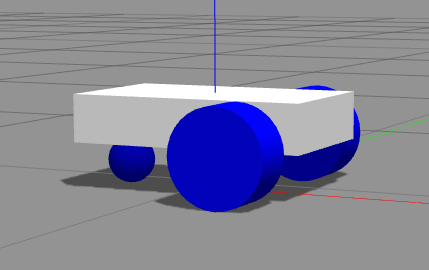
Step 3 – macros.xacro

* Create a file called macros.xacro containing a robot element: <?xml version="1.0"?><robot></robot>
* Move the Left Wheel XML objects (link, joint, transmission) into the new macros.xacro file, inside the robot tags.
* Wrap the Left Wheel objects with a macro tag, naming it "wheel" and creating two parameters: <macro name="wheel" params="lr y"> … </macro>
* For each "left" in the name specified for each XML object, replace it with ${lr}. We will pass this part of the name to the macro when we call it.
* Likewise, we need to pass to the macro which side of the robot we want the wheel. We can do this by passing a positive or negative one to the Y position for the wheel joint. If we've used our other properties well, we can do something like this: ${y\*(wheelWidth+chassisWidth)/2}
* Include the macros file in your main xacro document:   
  <xacro:include filename="$(find simpleton\_description)/urdf/macros.xacro" />
* Replace the Left Wheel XML objects with a call to the macro: <wheel lr="left" y="-1"/>
* Finally, test it out by launching the robot in Gazebo again. If everything looks correct, replace the Right Wheel XML objects with <wheel lr="right" y="1"/>. If not, ROS is pretty good about printing the issues to standard output. You can also review the final example of the files below.

Bonus Step

One tricky part about having a semi-realistic simulation is setting the correct inertial matrix values. You can calculate these yourself, measure them on an actual robot, generate them from a solid modeling program, or estimate these values based on their volume and mass. For the latter option, provided here are three macros for automatically calculating inertia.

The example below includes the inertia macros and some adjustments to Simpleton to make him a good robot for autonomous vehicle testing, including a caster wheel to keep the chasses flat, in the form of a frictionless sphere.



simpleton.xacro

<?xml version="1.0"?>

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

<xacro:include filename="$(find simpleton\_description)/urdf/simpleton.gazebo" />

<xacro:include filename="$(find simpleton\_description)/urdf/macros.xacro" />

<xacro:property name="wheelThickness" value="0.1"/>

<xacro:property name="wheelRadius" value="0.1"/>

<xacro:property name="chassisWidth" value="0.3"/>

<xacro:property name="chassisHeight" value="0.1"/>

<xacro:property name="chassisLength" value="0.5"/>

<!-- Base link -->

<link name="footprint" />

<!-- Chassis -->

<joint name="base\_joint" type="fixed">

<parent link="footprint"/>

<child link="chassis"/>

</joint>

<link name="chassis">

<collision>

<origin xyz="0 0 ${wheelRadius+(chassisHeight/2)}" rpy="0 0 0"/>

<geometry>

<box size="${chassisLength} ${chassisWidth} ${chassisHeight}"/>

</geometry>

</collision>

<visual>

<origin xyz="0 0 ${wheelRadius+(chassisHeight/2)}" rpy="0 0 0"/>

<geometry>

<box size="${chassisLength} ${chassisWidth} ${chassisHeight}"/>

</geometry>

<material name="white"/>

</visual>

<inertial>

<mass value="1"/>

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

<box\_inertia m="0.5" x="${chassisLength}" y="${chassisWidth}" z="${chassisHeight}"/>

</inertial>

</link>

<wheel lr="left" y="-1"/>

<wheel lr="right" y="1"/>

</robot>

macros.xacro

<?xml version="1.0"?>

<robot>

<!-- Wheel - Start -->

<macro name="wheel" params="lr y">

<link name="${lr}\_wheel">

<collision>

<origin xyz="0 0 0" rpy="0 1.5708 1.5708" />

<geometry>

<cylinder length="${wheelThickness}" radius="${wheelRadius}"/>

</geometry>

</collision>

<visual>

<origin xyz="0 0 0" rpy="0 1.5708 1.5708" />

<geometry>

<cylinder length="${wheelThickness}" radius="${wheelRadius}"/>

</geometry>

<material name="black"/>

</visual>

<inertial>

<origin xyz="0 0 0" rpy="0 1.5708 1.5708" />

<mass value="0.2"/>

<cylinder\_inertia m="0.2" r="${wheelRadius}" h="${wheelThickness}"/>

<inertia ixx="0.4" ixy="0.1" ixz="0.1" iyy="0.4" iyz="0.1" izz="0.2"/>

</inertial>

</link>

<joint name="${lr}\_wheel\_hinge" type="continuous">

<parent link="chassis"/>

<child link="${lr}\_wheel"/>

<origin xyz="${chassisLength/4} ${y\*(((wheelThickness+chassisWidth)/2)+.01)} ${wheelRadius}" rpy="0 0 0" />

<axis xyz="0 1 0" rpy="0 0 0" />

<!--<limit effort="100" velocity="100"/>-->

<!--<joint\_properties damping="0.0" friction="0"/>-->

<dynamics damping="0.0" />

</joint>

<gazebo reference="${lr}\_wheel">

<mu1 value="1.0"/>

<mu2 value="1.0"/>

<kp value="10000000.0" />

<kd value="1.0" />

<fdir1 value="0 1 0"/>

<material>Gazebo/Blue</material>

</gazebo>

<gazebo reference="${lr}\_wheel\_hinge">

<implicitSpringDamper value='true' />

</gazebo>

<transmission name="${lr}\_trans">

<type>transmission\_interface/SimpleTransmission</type>

<joint name="${lr}\_wheel\_hinge">

<hardwareInterface>EffortJointInterface</hardwareInterface>

</joint>

<actuator name="${lr}Motor">

<hardwareInterface>hardwareInterface/EffortJointInterface</hardwareInterface>

<mechanicalReduction>10</mechanicalReduction>

</actuator>

</transmission>

</macro>

<!-- Wheel - End -->

<macro name="cylinder\_inertia" params="m r h">

<inertia ixx="${m\*(3\*r\*r+h\*h)/12}" ixy = "0" ixz = "0"

iyy="${m\*(3\*r\*r+h\*h)/12}" iyz = "0"

izz="${m\*r\*r/2}" />

</macro>

<macro name="box\_inertia" params="m x y z">

<inertia ixx="${m\*(y\*y+z\*z)/12}" ixy = "0" ixz = "0"

iyy="${m\*(x\*x+z\*z)/12}" iyz = "0"

izz="${m\*(x\*x+z\*z)/12}" />

</macro>

<macro name="sphere\_inertia" params="m r">

<inertia ixx="${2\*m\*r\*r/5}" ixy = "0" ixz = "0"

iyy="${2\*m\*r\*r/5}" iyz = "0"

izz="${2\*m\*r\*r/5}" />

</macro>

</robot>

Simpleton.gazebo

<?xml version="1.0"?>

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

<gazebo reference="chassis">

<!--Stiffness -->

<kp>1000000.0</kp>

<!--Dampening-->

<kd>0.1</kd>

<dampingFactor>0</dampingFactor>

<material>Gazebo/White</material>

<selfCollide>true</selfCollide>

<turnGravityOff>false</turnGravityOff>

<mu1 value="0.0"/>

<mu2 value="0.0"/>

<fdir1 value="0 0 0"/>

</gazebo>

<gazebo>

<plugin name="differential\_drive\_controller" filename="libgazebo\_ros\_diff\_drive.so">

<legacyMode>false</legacyMode>

<alwaysOn>true</alwaysOn>

<updateRate>100</updateRate>

<leftJoint>left\_wheel\_hinge</leftJoint>

<rightJoint>right\_wheel\_hinge</rightJoint>

<wheelSeparation>${chassisWidth+0.02}</wheelSeparation>

<wheelDiameter>${2\*wheelRadius}</wheelDiameter>

<torque>2</torque>

<commandTopic>simpleton/cmd\_vel</commandTopic>

<odometryTopic>simpleton/odom\_diffdrive</odometryTopic>

<odometryFrame>odom</odometryFrame>

<robotBaseFrame>footprint</robotBaseFrame>

</plugin>

</gazebo>

</robot>

Chapter 4: Sensors

1. Add a single-lens camera
   1. Access image data
   2. Adjust camera angle and height

**1. Camera**

Similar to the differential drive control plugin, ROS also has a standard camera controller, so adding a camera to our Simpleton robot is a simple exercise. All we have to do is add a link that will represent the camera, add a joint to mount the camera to Simpleton, and add another gazebo element referencing the new link. This new Gazebo element will include a sensor element which will specify the camera\_controller plugin and also the name of the topics it will publish too. So let us add the following:

simpleton.xacro (additions)

<xacro:property name="cameraSize" value="0.05"/>

<!-- Camera -->

<!-- You can adjust the camera angle with the p in rpy, measured in radians. -->

<!-- You can also increate the height of the camera by adding a number, in meters, -->

<!-- to the z in xyz. For example, to simulate a camera mounted on a 1/2 meter post, -->

<!-- we can replace the z value with ${wheelRadius+chassisHeight+0.5} -->

<joint name="camera\_joint" type="fixed">

<origin xyz="${chassisLength/2} 0 ${wheelRadius+chassisHeight}" rpy="0 .2 0"/>

<parent link="chassis"/>

<child link="camera"/>

</joint>

<link name="camera">

<collision>

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

<geometry>

<box size="${cameraSize} ${cameraSize} ${cameraSize}"/>

</geometry>

</collision>

<visual>

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

<geometry>

<box size="${cameraSize} ${cameraSize} ${cameraSize}"/>

</geometry>

<material name="blue"/>

</visual>

<inertial>

<mass value="0.05" />

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

<box\_inertia m="0.05" x="${cameraSize}" y="${cameraSize}" z="${cameraSize}" />

</inertial>

</link>

simpleton.gazebo (additions)

<gazebo reference="camera">

<material>Gazebo/Blue</material>

<sensor type="camera" name="camera1">

<update\_rate>30.0</update\_rate>

<camera name="head">

<horizontal\_fov>1.3962634</horizontal\_fov>

<image>

<width>800</width>

<height>800</height>

<format>R8G8B8</format>

</image>

<clip>

<near>0.02</near>

<far>300</far>

</clip>

</camera>

<plugin name="camera\_controller" filename="libgazebo\_ros\_camera.so">

<alwaysOn>true</alwaysOn>

<updateRate>0.0</updateRate>

<cameraName>simpleton/camera1</cameraName>

<imageTopicName>image\_raw</imageTopicName>

<cameraInfoTopicName>camera\_info</cameraInfoTopicName>

<frameName>camera\_link</frameName>

<hackBaseline>0.07</hackBaseline>

<distortionK1>0.0</distortionK1>

<distortionK2>0.0</distortionK2>

<distortionK3>0.0</distortionK3>

<distortionT1>0.0</distortionT1>

<distortionT2>0.0</distortionT2>

</plugin>

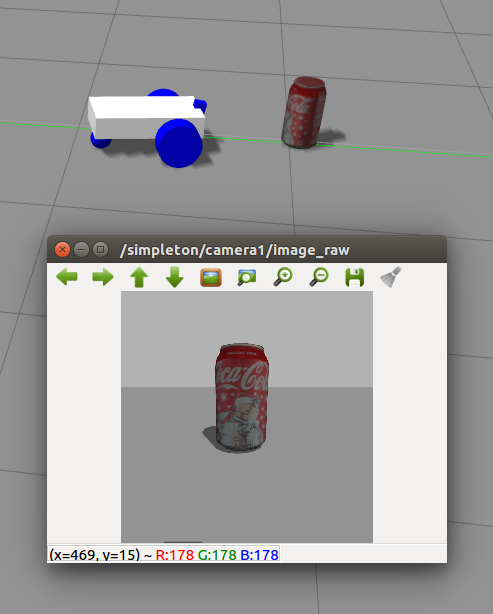
</sensor>

</gazebo>

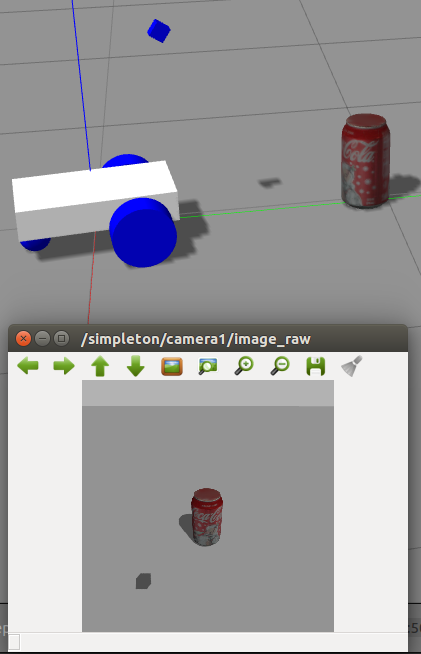
At this point we can launch Simpleton, and in a separate terminal run the image viewer to test our camera.

~$ rosrun image\_view image\_view image:=/simpleton/camera1/image\_raw

The scene can be a little uninteresting, so it's helpful to add an object, such as the Coke can that can be found under the Insert tab on the left of the Gazebo GUI. To avoid too much terminal juggling, let's add a new node to our launch file in simpleton\_gazebo package in order to launch the image viewer automatically for now one. Add the following to the bottom of the launch file



For line and lane following algorithms, one of the big experiment variables is camera angle and height. We can adjust both of these quite easily with the joint > origin element, changing z and p.

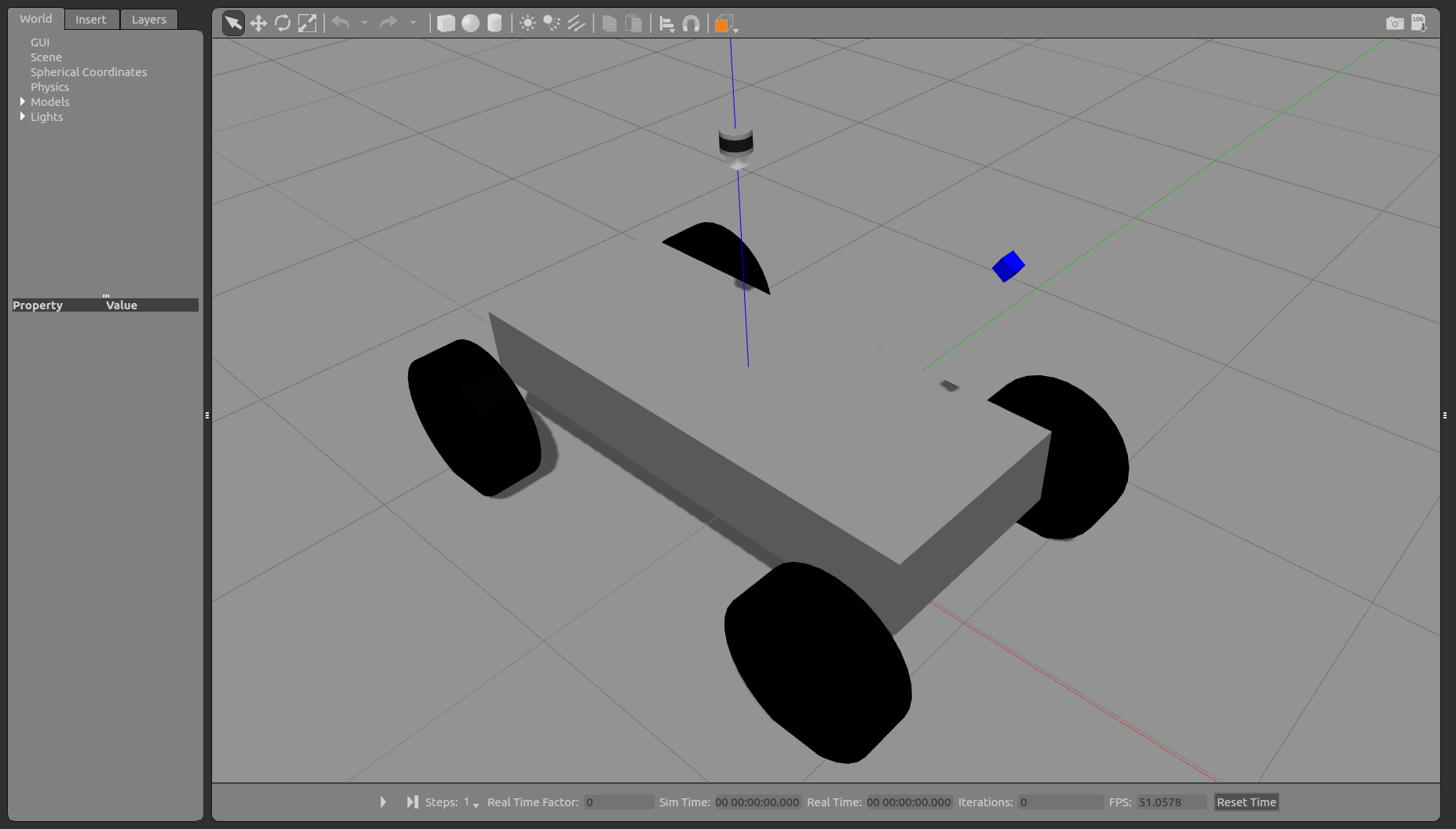


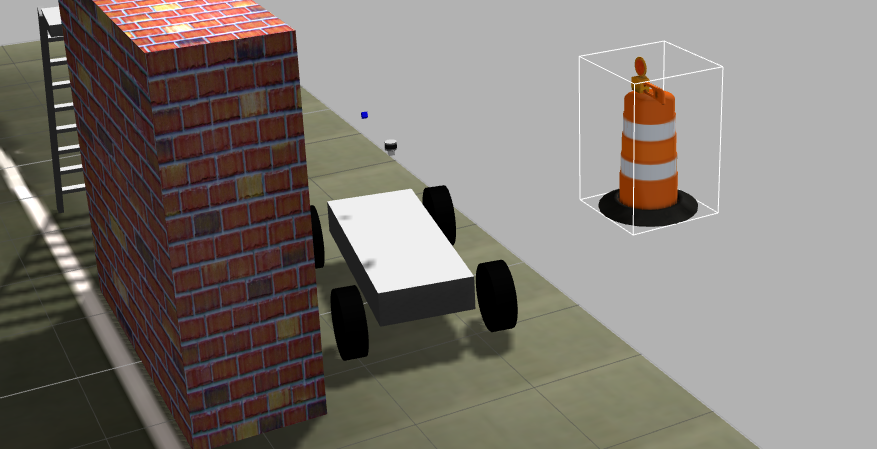
Chapter 5: ACTor Vehicle Model (Polaris GEM e2)

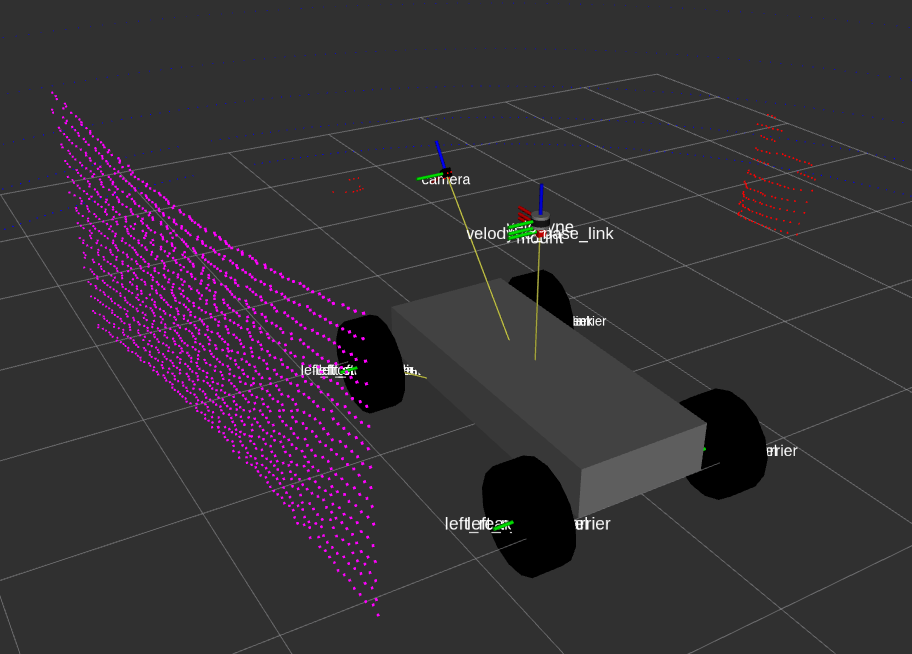
Repository location for the model: <https://github.com/LTU-AutoEV/actor_gazebo_model.git>

This project models the physical properties and sensors of the ACTor research platform vehicle, Polaris GEM e2. It receives Twist messages on topic /twist\_to\_ackermann. Camera output goes to /camera1/image\_raw and LiDAR point cloud data goes to /velodyne\_points.

To launch the vehicle with the LiDAR sensor, launch ackermann\_vehicle\_lidar.launch from the ackermann\_vehicle\_gazebo package. To launch without the LiDAR use ackermann\_vehicle.launch







Notes:

URDF can only have one root link, so all links need to be joined to other links or the root link.

sudo apt-get install ros-kinetic-velodyne-gazebo-plugins