

October 2016



BIRC

BIU Robotics Consortium

# ROS – Lecture 4

Gazebo simulator  
Reading Sensor Data  
Wander-Bot

Lecturer: Roi Yehoshua  
[roiyeho@gmail.com](mailto:roiyeho@gmail.com)

# Simulators

- In simulation, we can model as much or as little of reality as we desire
- Sensors and actuators can be modeled as ideal devices, or they can incorporate various levels of distortion, errors, and unexpected faults
- Automated testing of control algorithms typically requires simulated robots, since the algorithms under test need to be able to experience the consequences of their actions
- Due to the isolation provided by the messaging interfaces of ROS, a vast majority of the robot's software graph can be run identically whether it is controlling a real robot or a simulated robot

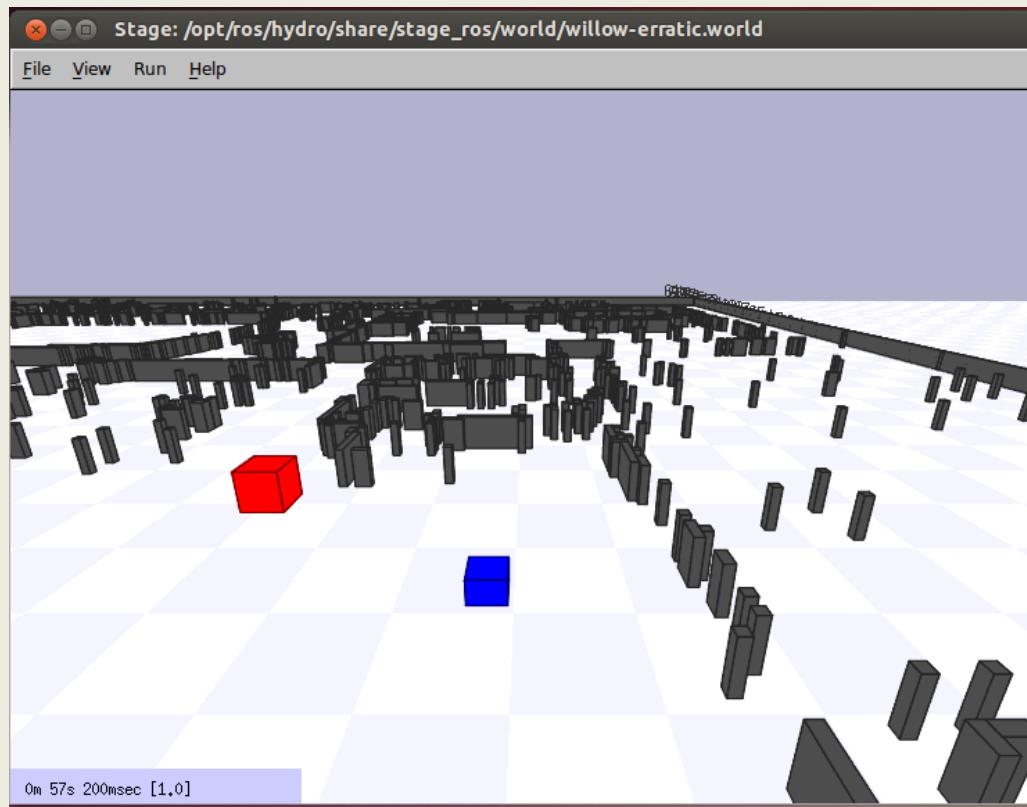
# ROS Stage Simulator

- [http://wiki.ros.org/simulator\\_stage](http://wiki.ros.org/simulator_stage)
- A 2D simulator that provides a virtual world populated by mobile robots, along with various objects for the robots to sense and manipulate



# ROS Stage Simulator

- In perspective view of the robot

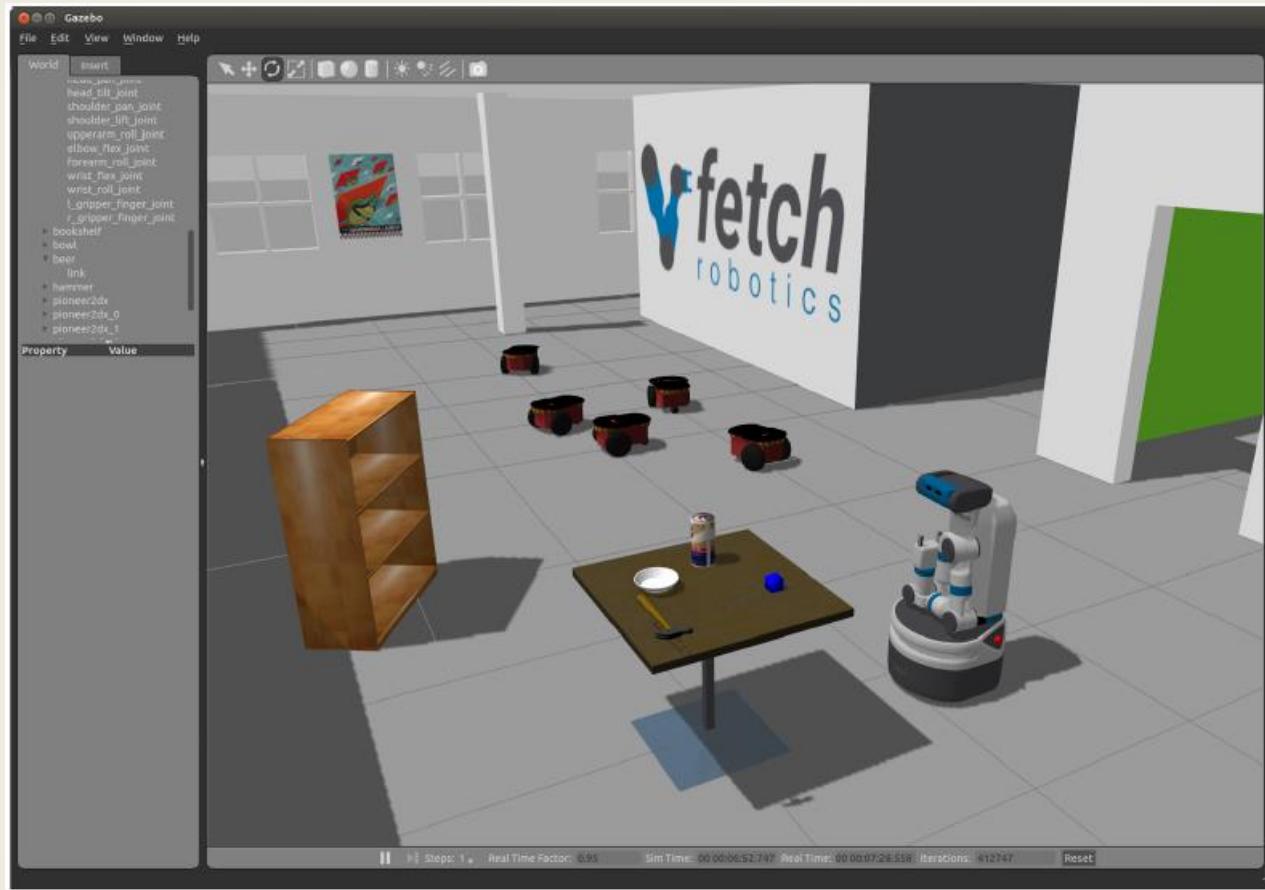


# Gazebo



- A multi-robot simulator
- Like Stage, it is capable of simulating a population of robots, sensors and objects, but does so in 3D
- Includes an accurate simulation of rigid-body physics and generates realistic sensor feedback
- Allows code designed to operate a physical robot to be executed in an artificial environment
- Gazebo is under active development at the OSRF (Open Source Robotics Foundation)

# Gazebo



- Gazebo Demo

# Gazebo

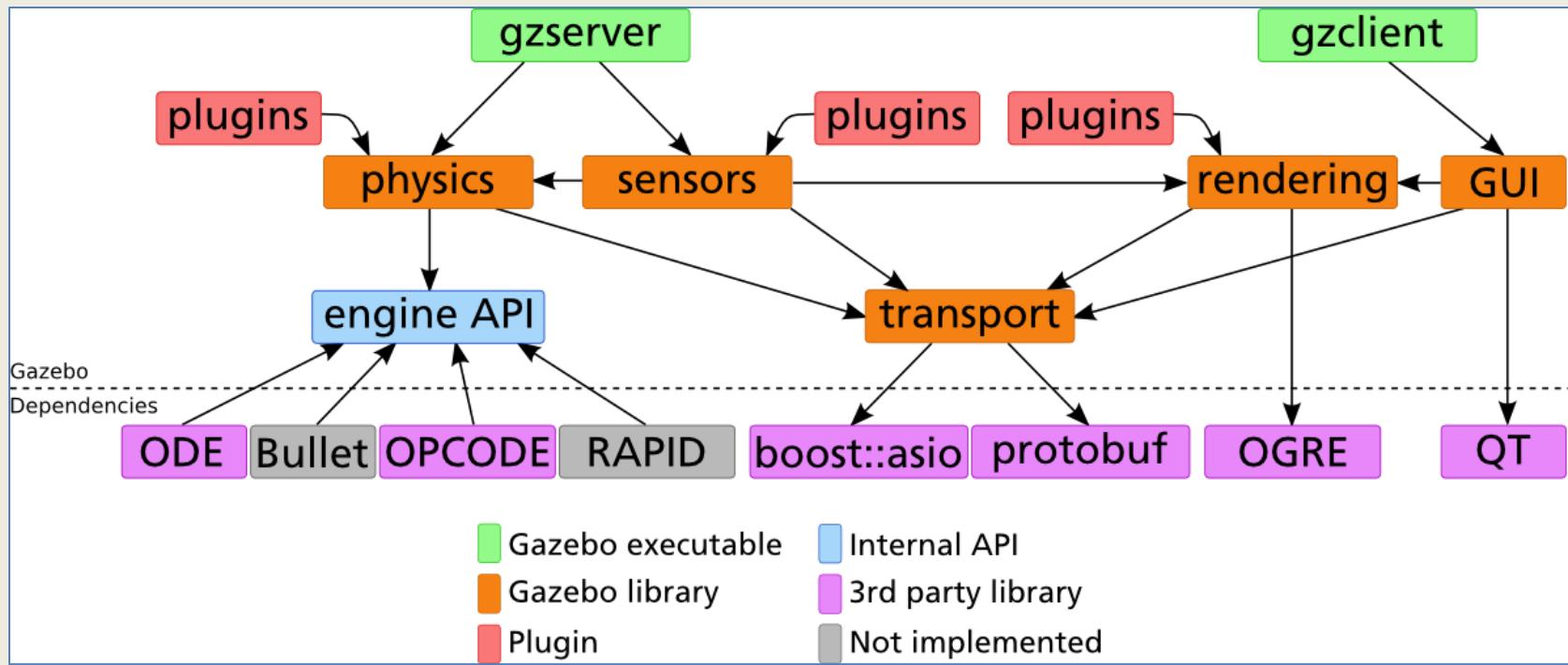
- ROS Indigo comes with Gazebo V2.2
- Gazebo home page - <http://gazebosim.org/>
- Gazebo tutorials - <http://gazebosim.org/tutorials>

# Gazebo Architecture

Gazebo consists of two processes:

- **Server:** Runs the physics loop and generates sensor data
  - *Executable:* gzserver
  - *Libraries:* Physics, Sensors, Rendering, Transport
- **Client:** Provides user interaction and visualization of a simulation.
  - *Executable:* gzclient
  - *Libraries:* Transport, Rendering, GUI

# Gazebo Architecture



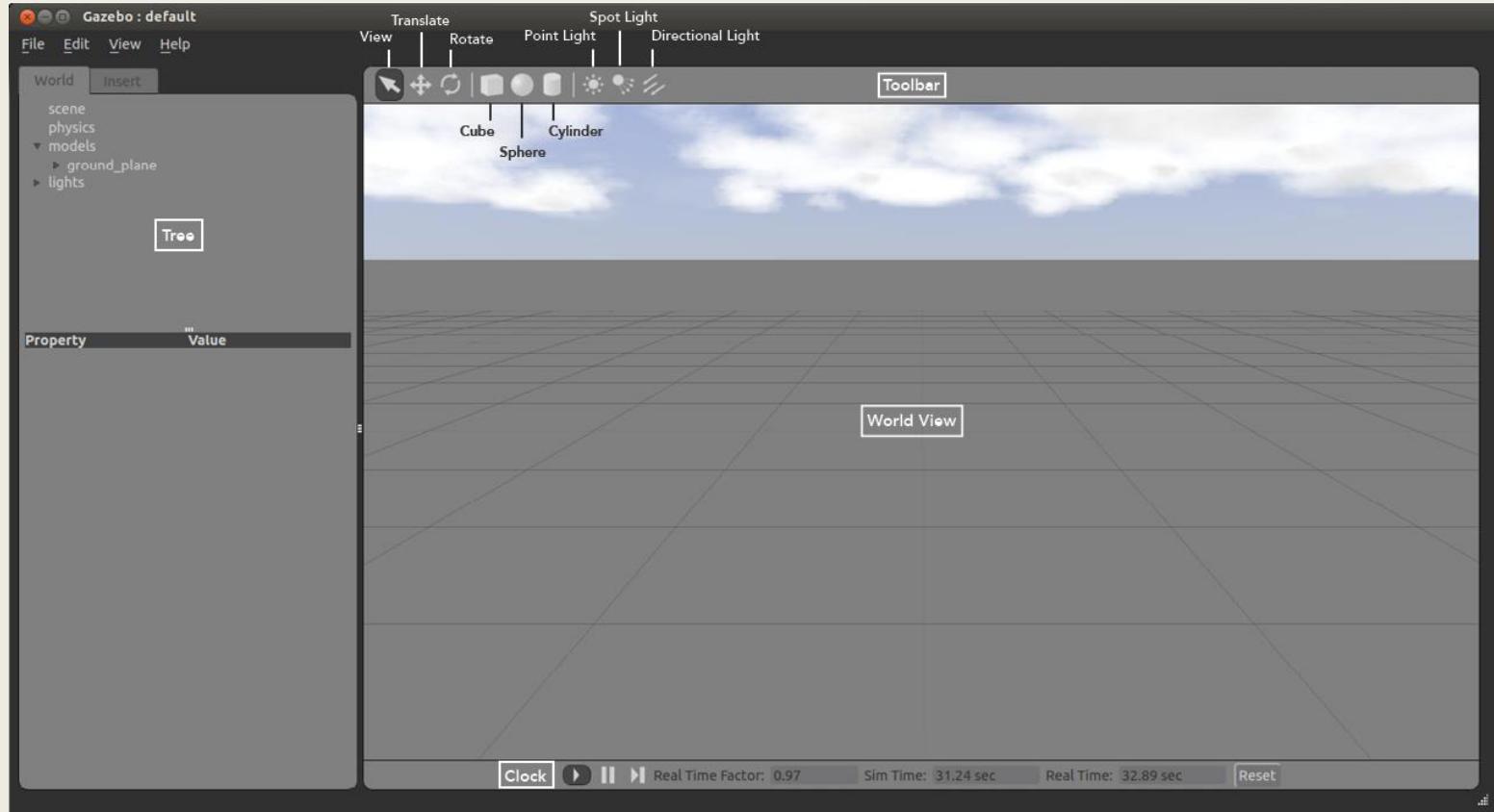
# Running Gazebo from ROS

- To launch Gazebo type:

```
$ rosrun gazebo_ros gazebo
```

- Note: When you first launch Gazebo it may take a few minutes to update its model database

# Gazebo User Interface



# The World View

- The World View displays the world and all of the models therein
- Here you can add, manipulate, and remove models
- You can switch between View, Translate and Rotate modes of the view in the left side of the Toolbar

View Mode



Translate	Left-press + drag
Orbit	Middle-press + drag
Zoom	Scroll wheel
Accelerated Zoom	Alt + Scroll wheel
Jump to object	Double-click object
Select object	Left-click object

Translate Mode



Translate	Left-press + drag
Translate (x-axis)	Left-press + X + drag
Translate (y-axis)	Left-press + Y + drag
Translate (z-axis)	Left-press + Z + drag

(Orbit & Zoom work in this mode, as well)

Rotate Mode



Rotate (spin) object	Left-press + drag
Rotate (x-axis)	Left-press + X + drag
Rotate (y-axis)	Left-press + Y + drag
Rotate (z-axis)	Left-press + Z + drag

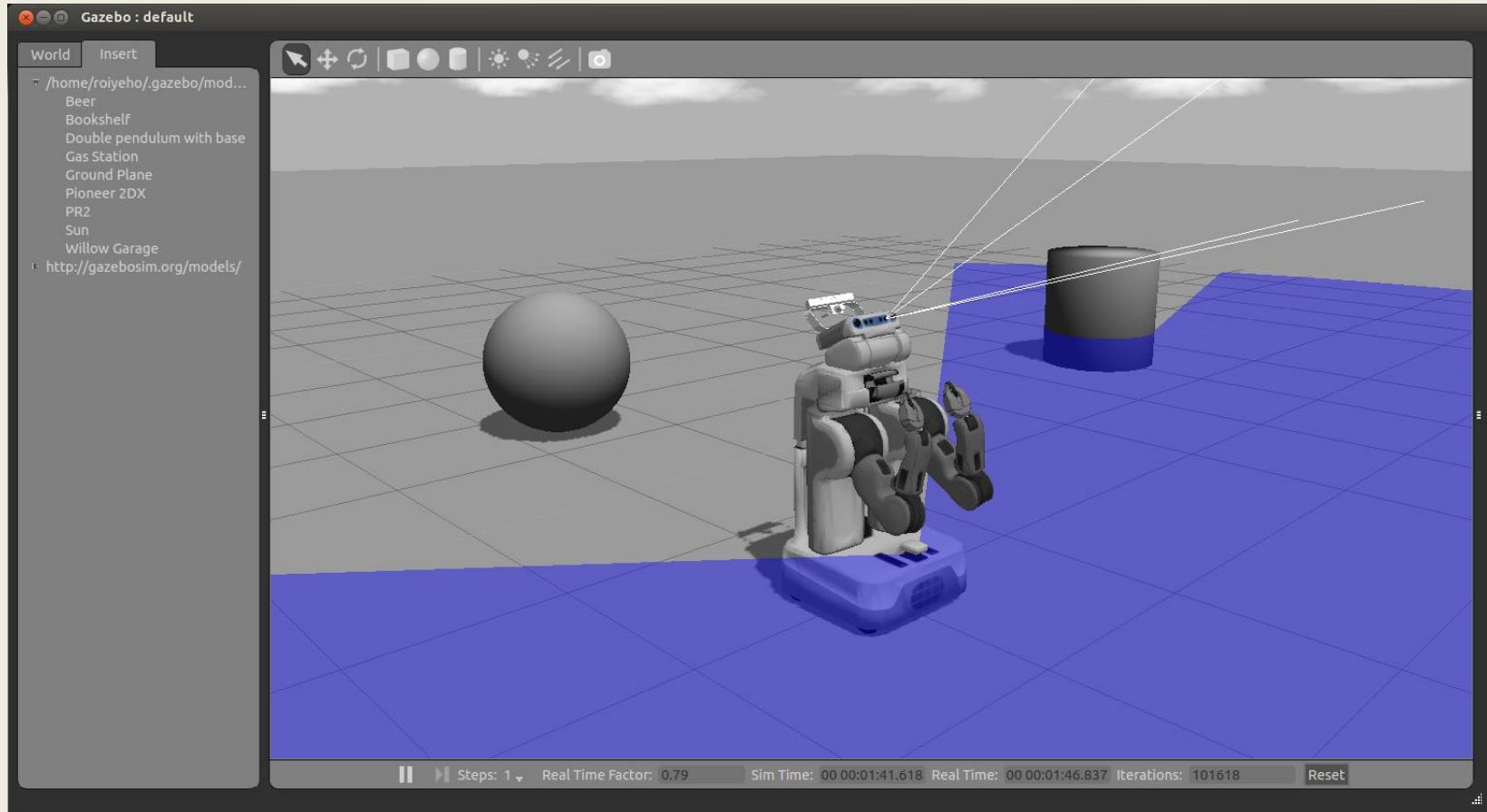
(Orbit & Zoom work in this mode, as well)

# Add a Model

To add a model to the world:

- left-click on the desired model in the Insert Tab on the left side
- move the cursor to the desired location in World View
- left-click again to release
- Use the Translate and Rotate modes to orient the model more precisely

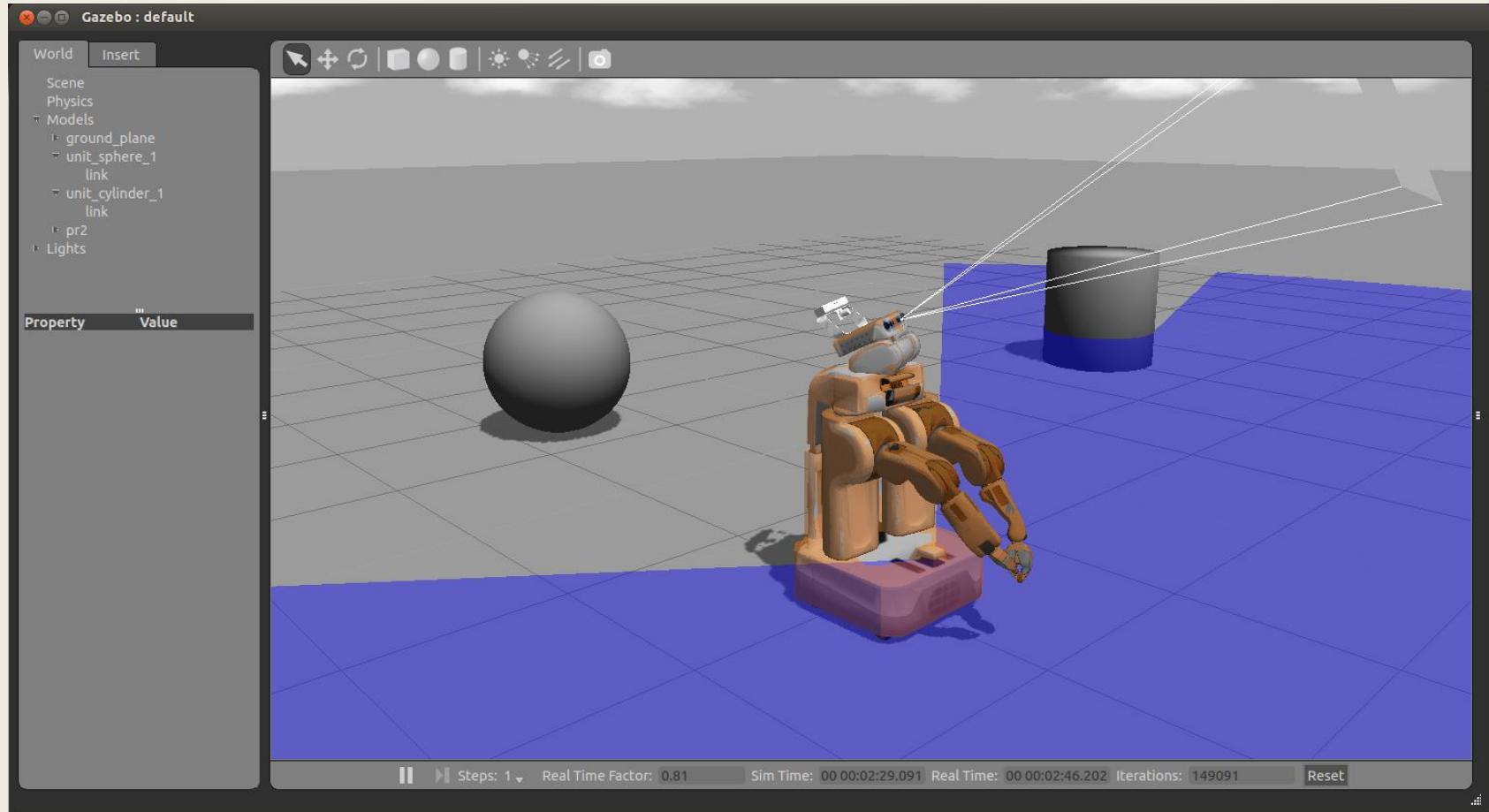
# Inserting PR2 Robot



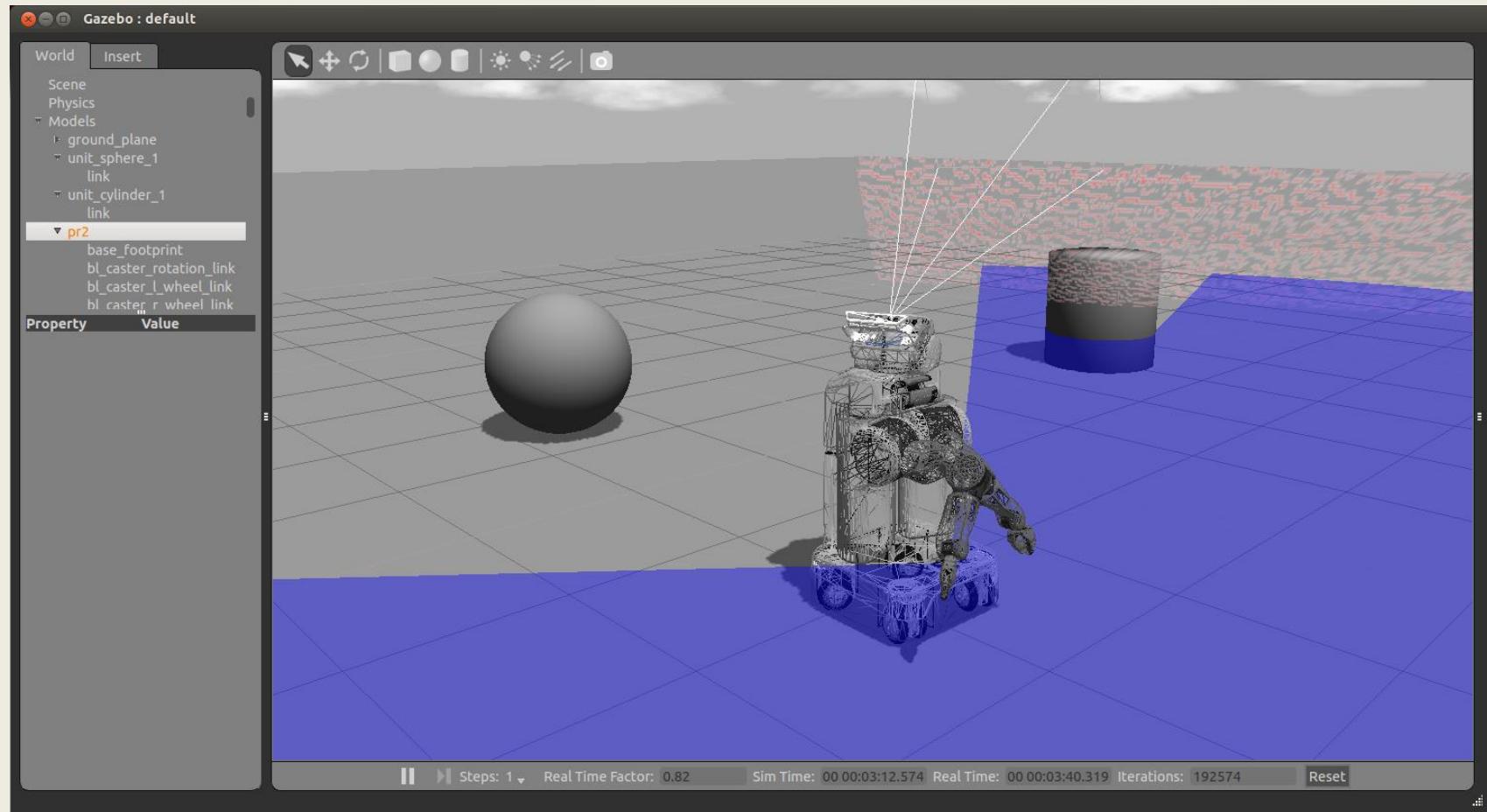
# Models Item

- The models item in the world tab contains a list of all models and their links
- Right-clicking on a model in the Models section gives you three options:
  - **Move to** – moves the view to be directly in front of that model
  - **Follow**
  - **View** – allows you to view different aspects of the model, such as Wireframe, Collisions, Joints
  - **Delete** – deletes the model

# Collisions View

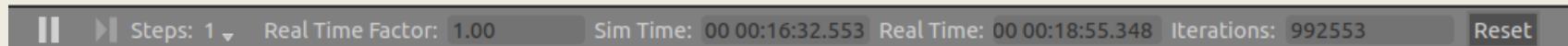


# Wireframe View



# Clock

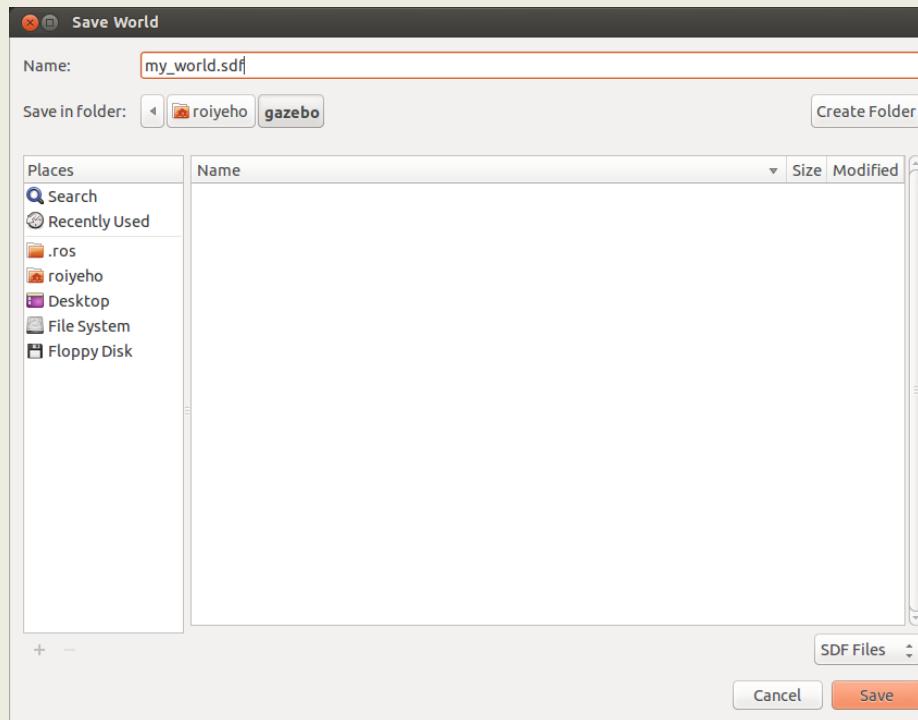
- You can start, pause and step through the simulation with the clock
- It is located at the bottom of the World View



- **Real Time Factor:** Displays how fast or slow the simulation is running in comparison to real time
  - A factor less than 1.0 indicates simulation is running slower than real time
  - Greater than 1.0 indicates faster than real time

# Saving a World

- Once you are happy with a world it can be saved through the File->Save As menu.
- Enter my\_world.sdf as the file name and click OK



# Loading a World

- A saved world may be loaded on the command line:

```
$ gazebo my_world.sdf
```

- The filename must be in the current working directory, or you must specify the complete path

# World Description File

- 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 has a .world extension
- The Gazebo server (gzserver) reads this file to generate and populate a world

# Example World Files

- Gazebo ships with a number of example worlds
- World files are found within the /worlds directory of your Gazebo resource path
  - A typical path might be /usr/share/gazebo-2.2
- In `gazebo_ros` package there are built-in launch files that load some of these world files
- For example, to launch `willowgarage_world` type:

```
$ roslaunch gazebo_ros willowgarage_world.launch
```

# willowgarage.world

```
<?xml version="1.0" ?>
<sdf version="1.4">
  <world name="default">
    <include>
      <uri>model://ground_plane</uri>
    </include>
    <include>
      <uri>model://sun</uri>
    </include>
    <include>
      <uri>model://willowgarage</uri>
    </include>
  </world>
</sdf>
```

- In this world file snippet you can see that three models are referenced
- The three models are searched for within your local Gazebo Model Database
- If not found there, they are automatically pulled from Gazebo's online database

# Launch World Files

- In `gazebo_ros` package there are built-in launch files that load some of these world files
- For example, to launch `willowgarage_world` type:

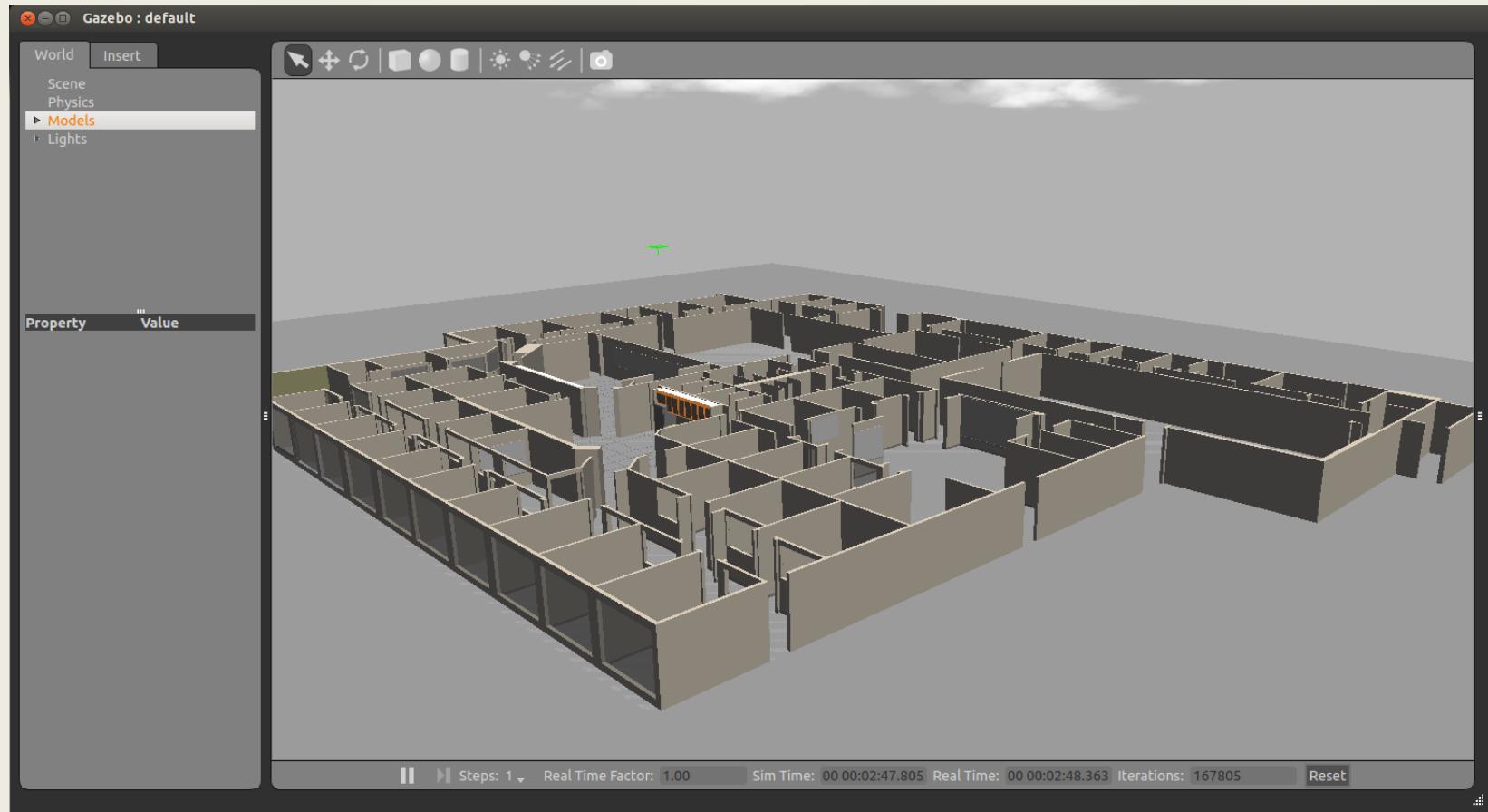
```
$ roslaunch gazebo_ros willowgarage_world.launch
```

# willowgarage\_world.launch

```
<launch>
  <!-- We reuse the logic in empty_world.launch, changing only the name of the world to
be launched -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="worlds/willowgarage.world"/> <!-- Note: the
world_name is with respect to GAZEBO_RESOURCE_PATH environmental variable -->
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="gui" value="true"/>
    <arg name="headless" value="false"/>
    <arg name="debug" value="false"/>
  </include>
</launch>
```

- This launch file inherits most of the necessary functionality from empty\_world.launch
- The only parameter we need to change is the world\_name parameter, substituting the empty.world world file with willowgarage.world
- The other arguments are simply set to their default values

# Willow Garage World



# Model Files

- A model file uses the same SDF format as world files, but contains only a single <model> tag
- Once a model file is created, it can be included in a world file using the following SDF syntax:

```
<include filename="model_file_name"/>
```

- You can also include any model from the online database and the necessary content will be downloaded at runtime

# willowgarage Model SDF File

```
<?xml version="1.0" ?>
<sdf version="1.4">
  <model name="willowgarage">
    <static>true</static>
    <pose>-20 -20 0 0 0 0</pose>
    <link name="walls">
      <collision name="collision">
        <geometry>
          <mesh>
            <uri>model://willowgarage/meshes/willowgarage_collision.dae</uri>
          </mesh>
        </geometry>
      </collision>
      <visual name="visual">
        <geometry>
          <mesh>
            <uri>model://willowgarage/meshes/willowgarage_visual.dae</uri>
          </mesh>
        </geometry>
        <cast_shadows>false</cast_shadows>
      </visual>
    </link>
  </model>
</sdf>
```

# Components of Models

- **Links:** A link contains the physical properties of one body of the model. This can be a wheel, or a link in a joint chain.
  - Each link may contain many collision, visual and sensor elements
- **Collision:** A collision element encapsulates a geometry that is used to collision checking.
  - This can be a simple shape (which is preferred), or a triangle mesh (which consumes greater resources).
- **Visual:** A visual element is used to visualize parts of a link.
- **Inertial:** The inertial element describes the dynamic properties of the link, such as mass and rotational inertia matrix.
- **Sensor:** A sensor collects data from the world for use in plugins.
- **Joints:** A joint connects two links.
  - A parent and child relationship is established along with other parameters such as axis of rotation, and joint limits.
- **Plugins:** A shared library created by a 3D party to control a model.

# Meet TurtleBot

- <http://wiki.ros.org/Robots/TurtleBot>
- A minimalist platform for ROS-based mobile robotics education and prototyping
- Has a small differential-drive mobile base
- Atop this base is a stack of laser-cut “shelves” that provide space to hold a netbook computer and depth camera and other devices
- Does not have a laser scanner
  - Despite this, mapping and navigation can work quite well for indoor spaces



# Turtlebot Simulation

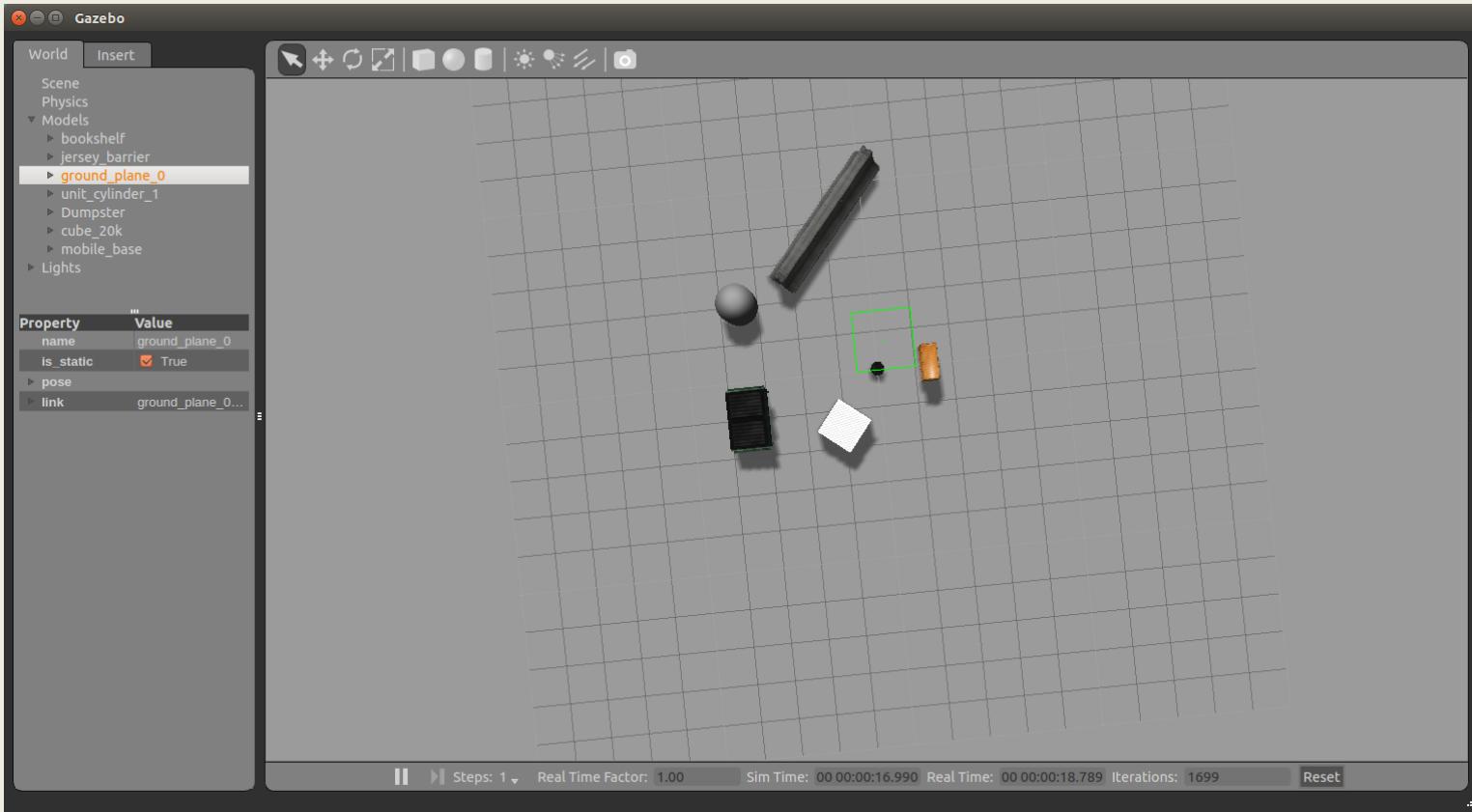
- To install Turtlebot simulation stack type:

```
$ sudo apt-get install ros-indigo-turtlebot-gazebo ros-indigo-turtlebot-apps ros-indigo-turtlebot-rviz-launchers
```

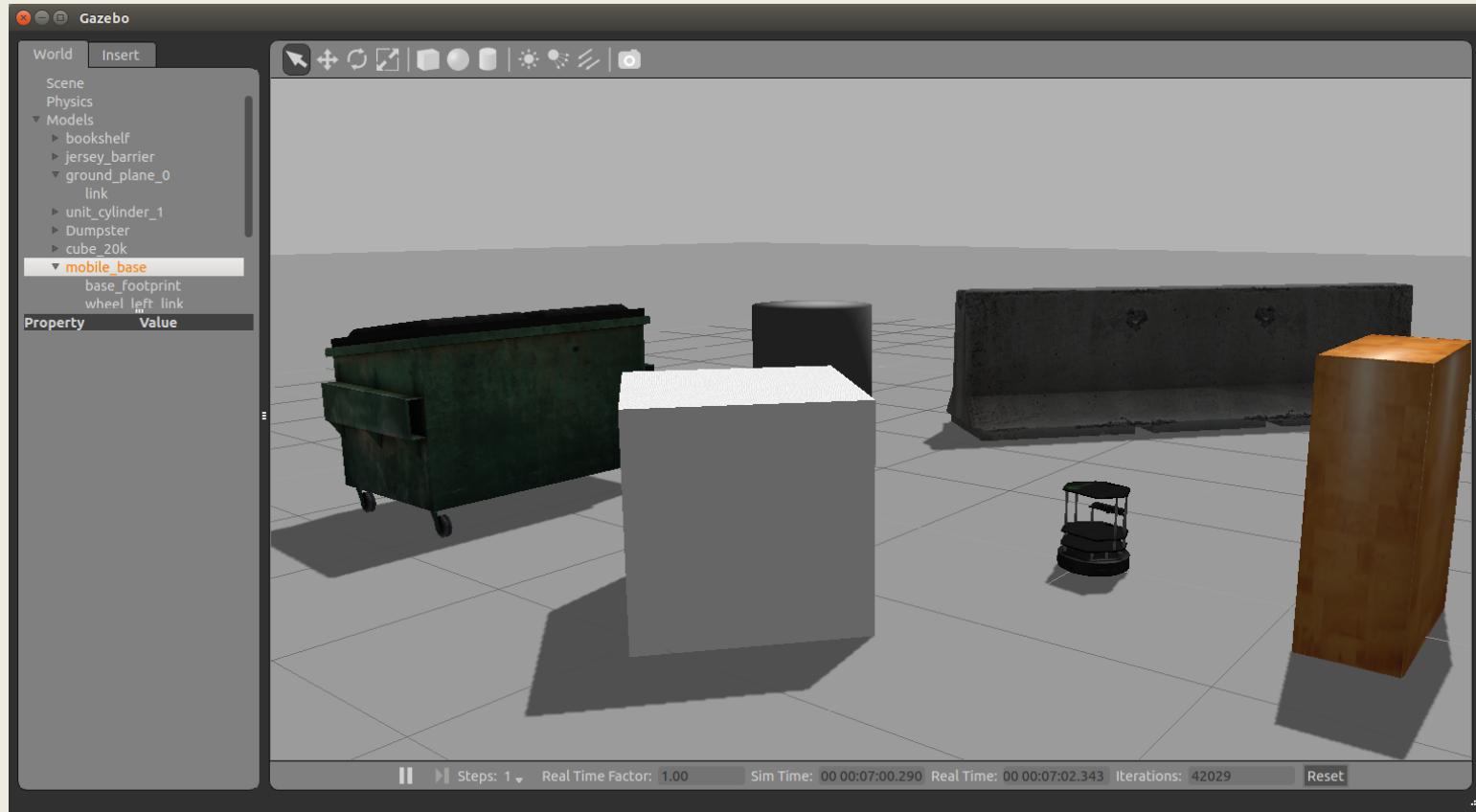
- To launch a simple world with a Turtlebot, type:

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

# Turtlebot Simulation



# Turtlebot Simulation

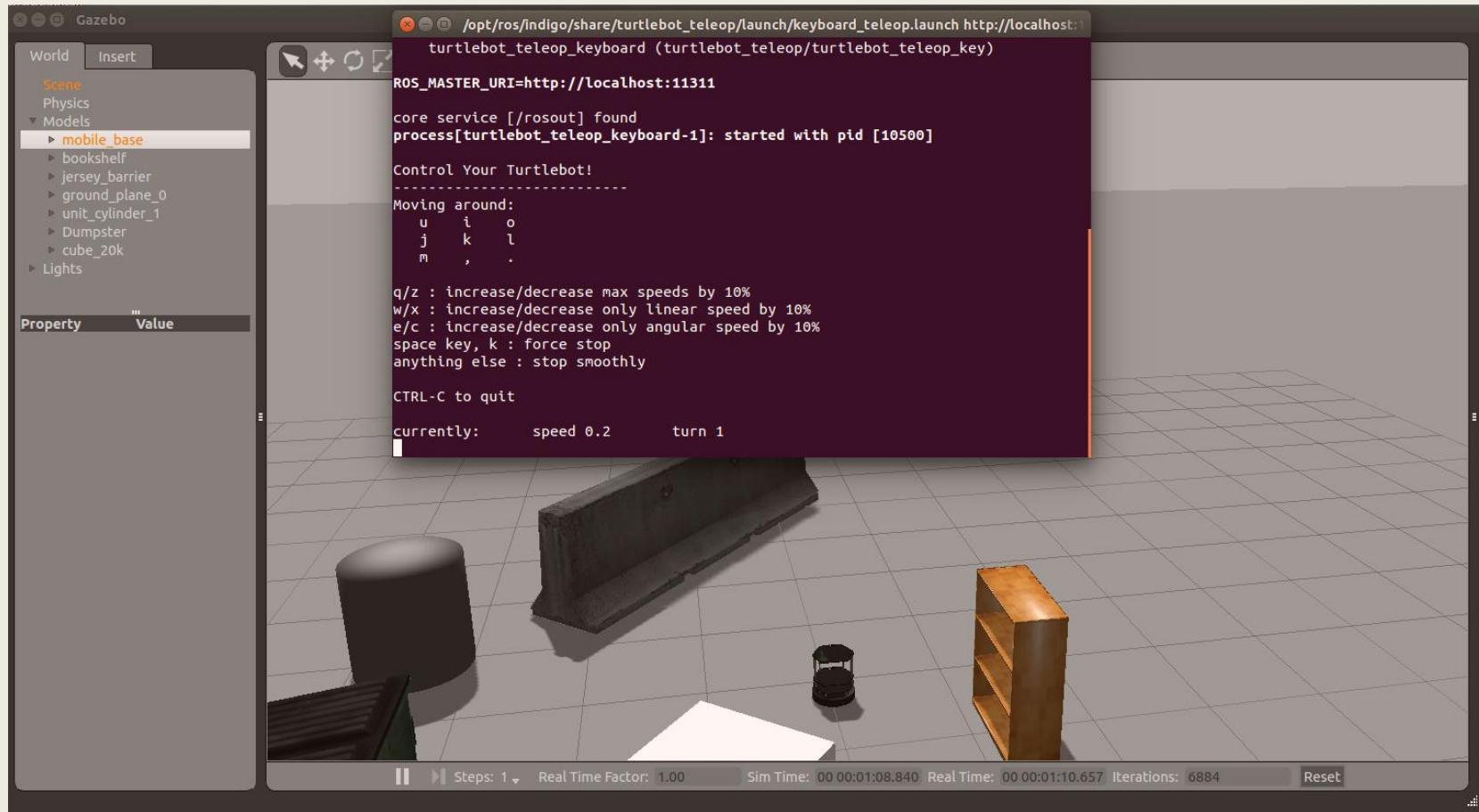


# Moving Turtlebot with Teleop

- Let's launch the teleop package so we can move it around the environment
- Run the following command:

```
$ roslaunch turtlebot_teleop keyboard_teleop.launch
```

# Moving Turtlebot with Teleop



# Laser Scan Data

- Laser data is published to the topic **/base\_scan**
- The message type that used to send information of the laser is `sensor_msgs/LaserScan`
- You can see the structure of the message using

```
$ rosmsg show sensor_msgs/LaserScan
```

# LaserScan Message

- [http://docs.ros.org/api/sensor\\_msgs/html/msg/LaserScan.html](http://docs.ros.org/api/sensor_msgs/html/msg/LaserScan.html)

```
# Single scan from a planar laser range-finder

Header header
# stamp: The acquisition time of the first ray in the scan.
# frame_id: The laser is assumed to spin around the positive Z axis
# (counterclockwise, if Z is up) with the zero angle forward along the x axis

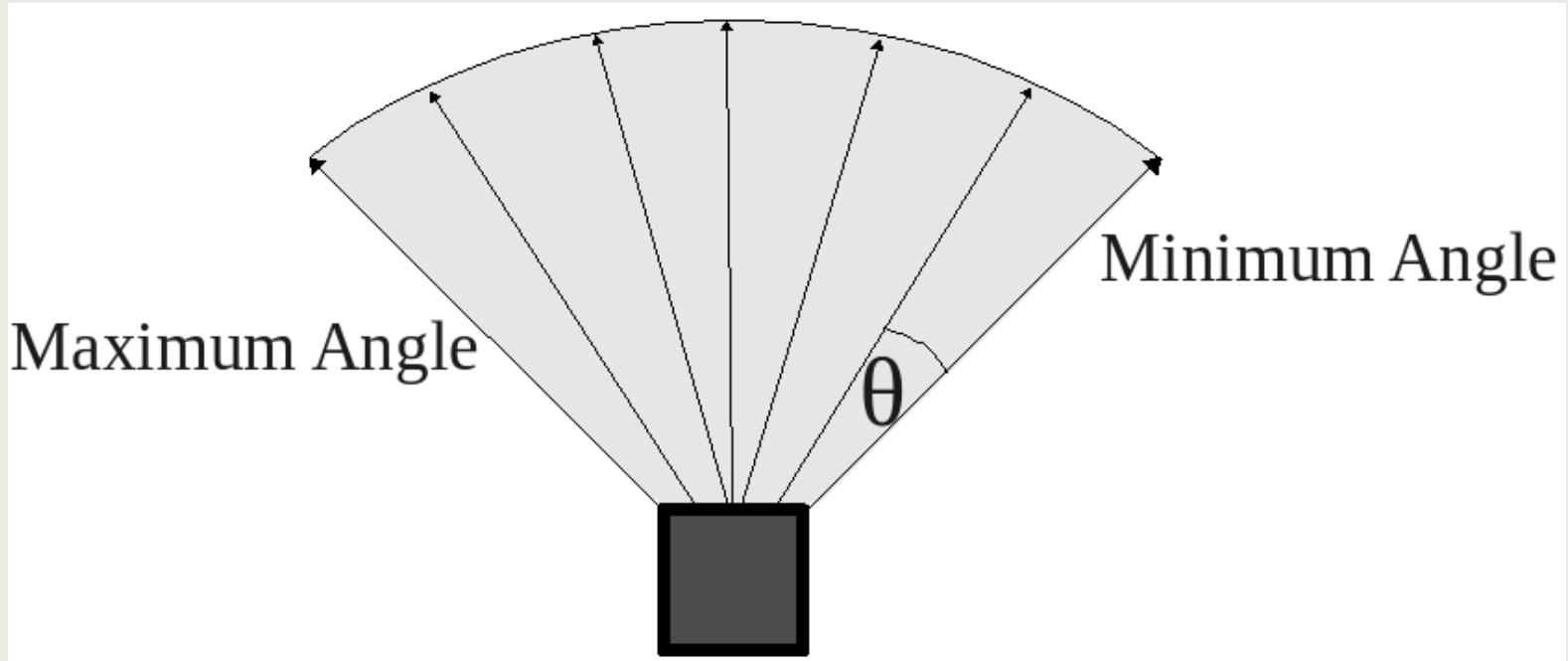
float32 angle_min # start angle of the scan [rad]
float32 angle_max # end angle of the scan [rad]
float32 angle_increment # angular distance between measurements [rad]

float32 time_increment # time between measurements [seconds] - if your scanner
# is moving, this will be used in interpolating position of 3d points
float32 scan_time # time between scans [seconds]

float32 range_min # minimum range value [m]
float32 range_max # maximum range value [m]

float32[] ranges # range data [m] (Note: values < range_min or > range_max should be
discarded)
float32[] intensities # intensity data [device-specific units]. If your
# device does not provide intensities, please leave the array empty.
```

# Laser Scanner



# Hokuyo Laser

- A common laser sensor used in robotics

[http://www.hokuyo-aut.jp/02sensor/07scanner/urg\\_04lx.html](http://www.hokuyo-aut.jp/02sensor/07scanner/urg_04lx.html)



# Hokuyo Laser

Model No.	URG-04LX
Power source	5VDC±5% <sup>*1</sup>
Current consumption	500mA or less(800mA when start-up)
Measuring area	60 to 4,095mm(white paper with 70mm□) 240°
Accuracy	60 to 1,000mm : ±10mm, 1,000 to 4,095mm : 1% of measurement
Repeatability	60 to 1,000mm : ±10mm
Angular resolution	Step angle : approx. 0.36° (360° /1,024 steps)
Light source	Semiconductor laser diode( $\lambda=785\text{nm}$ ), Laser safety class 1(IEC60825-1, 21 CFR 1040.10 & 1040.11)
Scanning time	100ms/scan
Noise	25dB or less
Interface	USB, RS-232C(19.2k, 57.6k, 115.2k, 250k, 500k, 750kbps), NPN open-collector(synchronous output of optical scanner : 1 pce)
Communication specifications	Exclusive command(SCIP Ver.1.1 or Ver.2.0) <sup>*2</sup>
Ambient temperature/humidity	-10 to +50 degrees C, 85% or less(Not condensing, not icing)
Vibration resistance	10 to 55Hz, double amplitude 1.5mm Each 2 hour in X, Y and Z directions
Impact resistance	196m/s <sup>2</sup> , Each 10 time in X, Y and Z directions
Weight	Approx. 160g
Accessory	Cable for power+communication/input+output(1.5m) 1 pce, D-sub connector with 9 pins 1 pce <sup>*3</sup>

# LaserScan Message

- Example of a laser scan message from Stage:  
(rostopic echo /scan -n1)

```
roiyeho@ubuntu: ~
...
header:
  seq: 1594
  stamp:
    secs: 159
    nsecs: 500000000
  frame_id: base_laser_link
angle_min: -2.35837626457
angle_max: 2.35837626457
angle_increment: 0.00436736317351
time_increment: 0.0
scan_time: 0.0
range_min: 0.0
range_max: 30.0
ranges: [2.427844524383545, 2.42826247215271, 2.4287266731262207, 2.4292376041412354, 2.429795026779175, 2.430398941040039, 2.4310495853424072, 2.4317471981048584, 2.4324913024902344, 2.4332826137542725, 2.4341206550598145, 2.4350056648254395, 2.4359381198883057, 2.436917543411255, 2.437944173812866, 2.439018487930298, 2.4401402473449707, 2.4413094520568848, 2.4425265789031982, 2.443791389465332, 2.4451043605804443, 2.446465253829956, 2.4478745460510254, 2.4493319988250732, 2.450838088989258, 2.452392816543579, 2.453996419906616, 2.455648899078369, 2.457350492477417, 2.459101438522339, 2.460901975631714, 2.462752103805542, 2.4646518230438232, 2.466601848602295, 2.468601942062378, 2.4706523418426514, 2.472753247802734, 2.474905490875244, 2.4771084785461426, 2.479362726211548, 2.481668472290039, 2.4840259552001953, 2.4864354133605957, 2.4888970851898193, 2.4914112091064453, 2.4939777851104736, 2.4965975284576416, 2.4992706775665283, 2.5019969940185547, 2.504777193069458, 2.5076115131378174, 2.510500192642212, 2.5134434700012207, 2.516441822052002, 2.5194954872131348, 2.5226047039031982, 2.5257697105407715, 2.5289909839630127, 2.53226900100708, 2.5356037616729736, 2.5389959812164307, 2.542445659637451, 2.5459535121917725, 2.5495197772979736, 2.553144693374634, 2.5568289756774902, 2.560572624206543, 2.56437611579895, 2.568240165710449, 2.572165012359619, 2.576151132583618, 2.5801987648010254, 2.584308624267578, 2.5884809494018555, 2.592716454595947, 2.597015380859375, 2.6013782024383545, 2.6058056354522705, 2.610297918319702, 2.6148557662963867, 2.6194796562194824, 2.6241698265075684, 2.628927230834961, 2.6337523460388184, 2.63478422164917, 2.6436073780059814, 2.6486384868621826, 2.6537396907806396, 3.4479820728302, 3.4547808170318604, 3.461672306060791, 3.4686577320098877, 3.4757378101348877, 3.4829134941101074, 3.490185499191284, 3.4975550174713135, 3.5050225257873535, 3.5125889778137207, 3.5202558040618896, 3.5280232429504395, 3.535892963409424, 3.543865442276001, 3.5519418716430664, 3.5601232051849365, 3.568410634994507, 3.5768051147460938, 3.5853075
```

# Depth Image to Laser Scan

- TurtleBot doesn't have a LIDAR by default
- However, the image from its depth camera can be used as a laser scan
- The node [depthimage\\_to\\_laserscan](#) takes a depth image and generates a 2D laser scan based on the provided parameters
- This node is run automatically when you run the turtlebot simulation in Gazebo
- The output range arrays contain NaNs and +-Infs (when the range is less than range\_min or larger than range\_max)
  - Comparisons with NaNs always return false

# Depth Image to Laser Scan

- tutrlebot\_world.launch

```
<launch>
...
<!-- Fake laser -->
<node pkg="nodelet" type="nodelet" name="laserscan_nodelet_manager"
args="manager"/>
<node pkg="nodelet" type="nodelet" name="depthimage_to_laserscan"
      args="load depthimage_to_laserscan/DepthImageToLaserScanNodelet
laserscan_nodelet_manager">
    <param name="scan_height" value="10"/>
    <param name="output_frame_id" value="/camera_depth_frame"/>
    <param name="range_min" value="0.45"/>
    <remap from="image" to="/camera/depth/image_raw"/>
    <remap from="scan" to="/scan"/>
</node>
</launch>
```

# LaserScan Message

- Example of a laser scan message from TurtleBot:  
(`rostopic echo /scan -n1`)

# Wander-bot

- We'll now put all the concepts we've learned so far together to create a robot that can wander around its environment
- This might not sound terribly earth-shattering, but such a robot is actually capable of doing meaningful work: there is an entire class of tasks that are accomplished by driving across the environment
- For example, many vacuuming or other floor-cleaning tasks can be accomplished by cleverly algorithms where the robot, carrying its cleaning tool, traverses its environment somewhat randomly
- The robot will eventually drive over all parts of the environment, completing its task

# Wander-bot

- **Refer:** Morgan Quigley O' Reiley: Programming Robots with ROS – **Chapter 7: Wander-bot**

## Ex. 4

- Implement a simple walker algorithm much like a Roomba robot vacuum cleaner
- The robot should move forward until it reaches an obstacle, then rotate in place until the way ahead is clear, then move forward again and repeat
- Bonus: rotate the robot in the direction that is more free from obstacles