Robotics & Data Mining Summer School

Lesson 07. Robotics Operating System

Kirill Svyatov, Alexander Miheev

Ulyanovsk State Technical University,

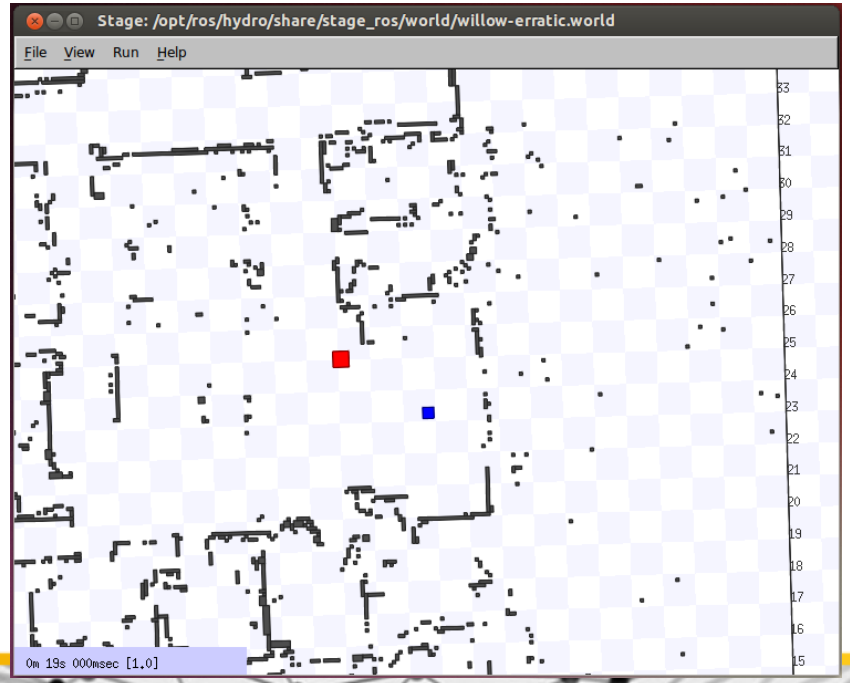
Faculty of Information Systems and Technologies

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

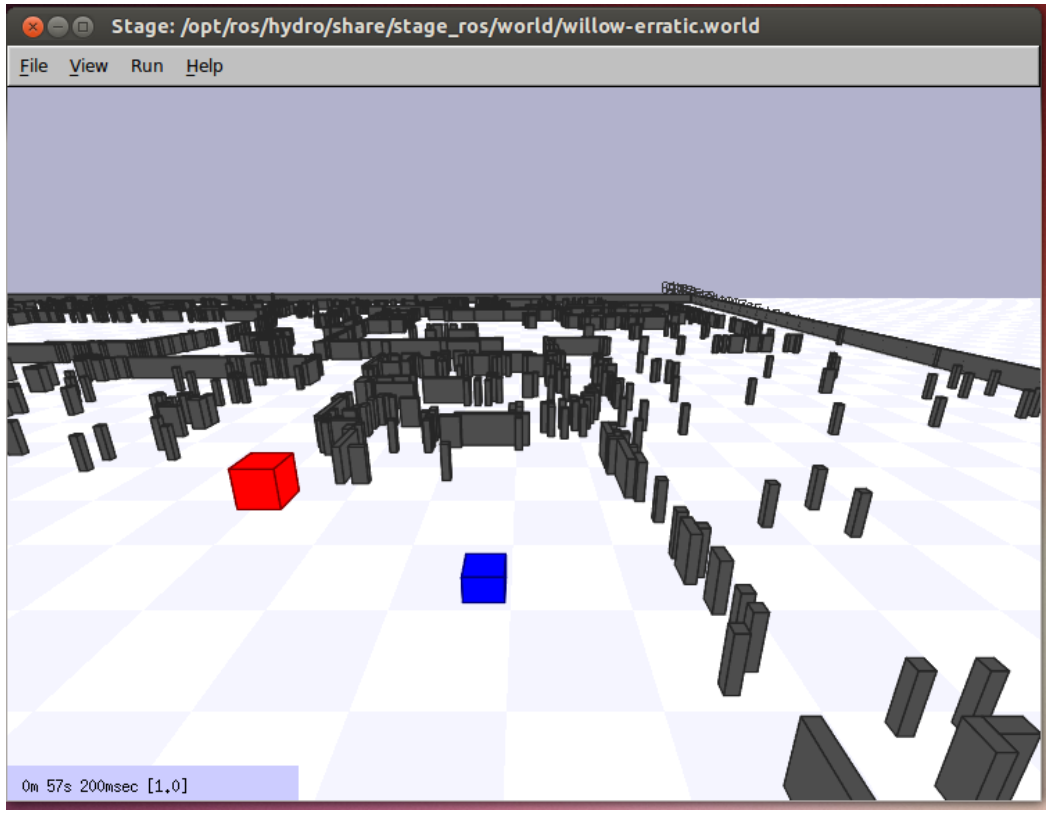
ROS Stage Simulator

- http://wiki.ros.org/simulator_stage
- 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

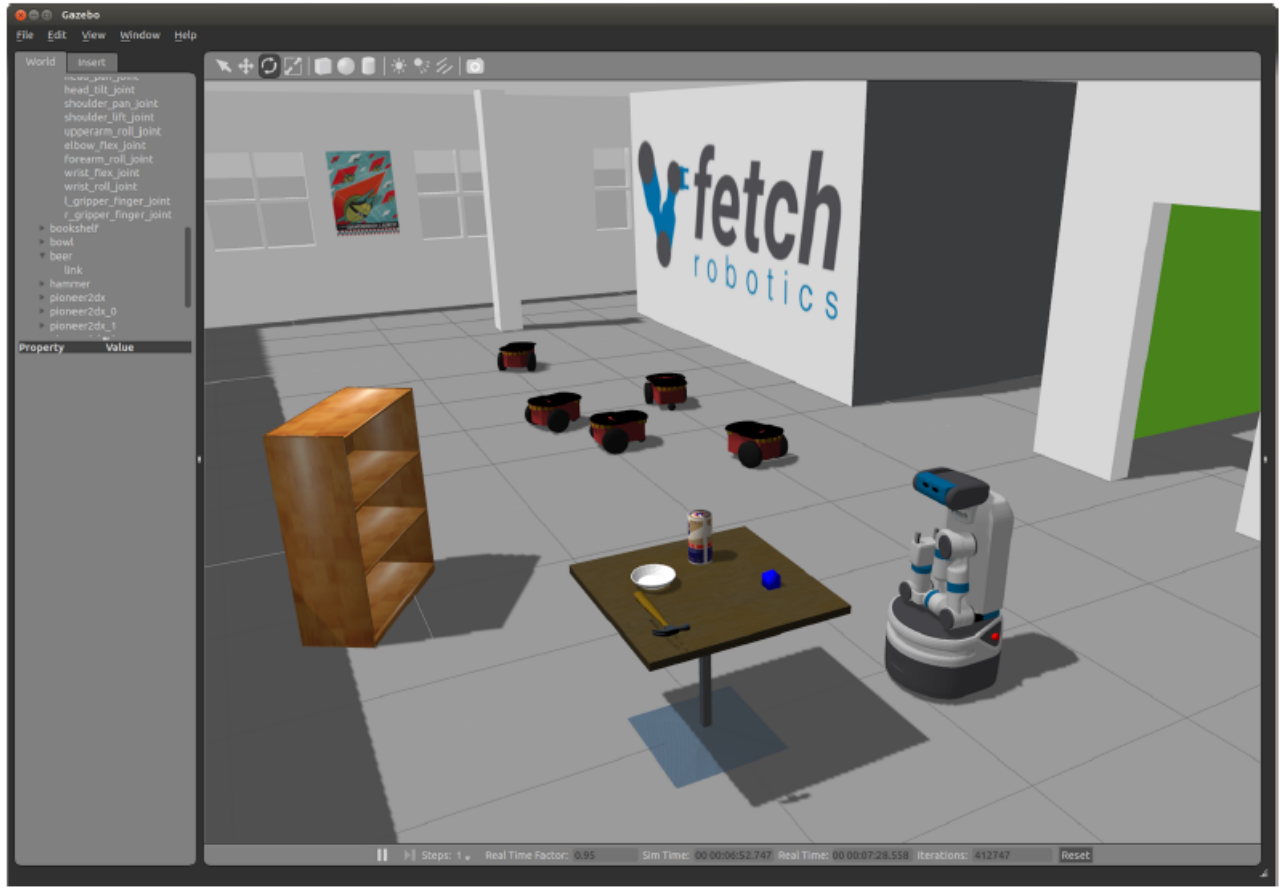
- In perspective view of the robot



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

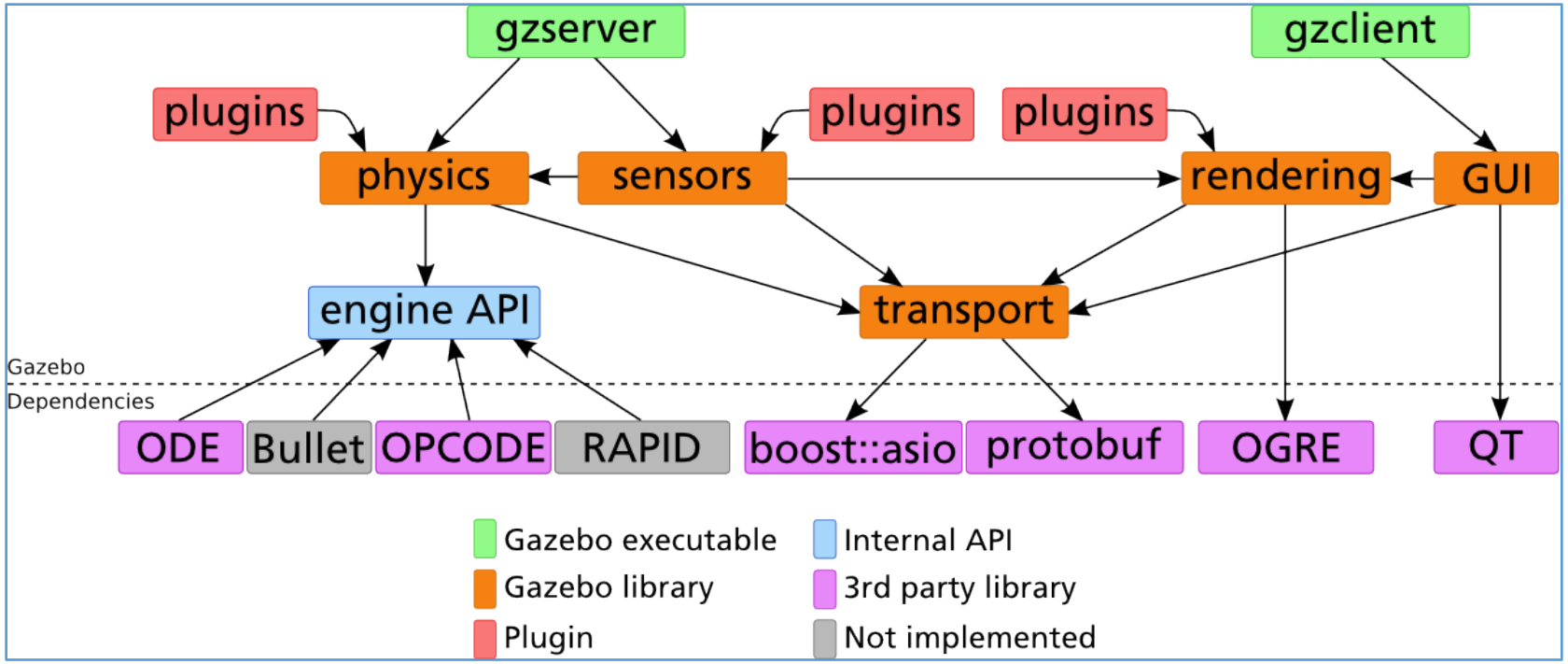
- Gazebo home page - http://gazebosim.org/
- Gazebo tutorials - http://gazebosim.org/tutorials

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



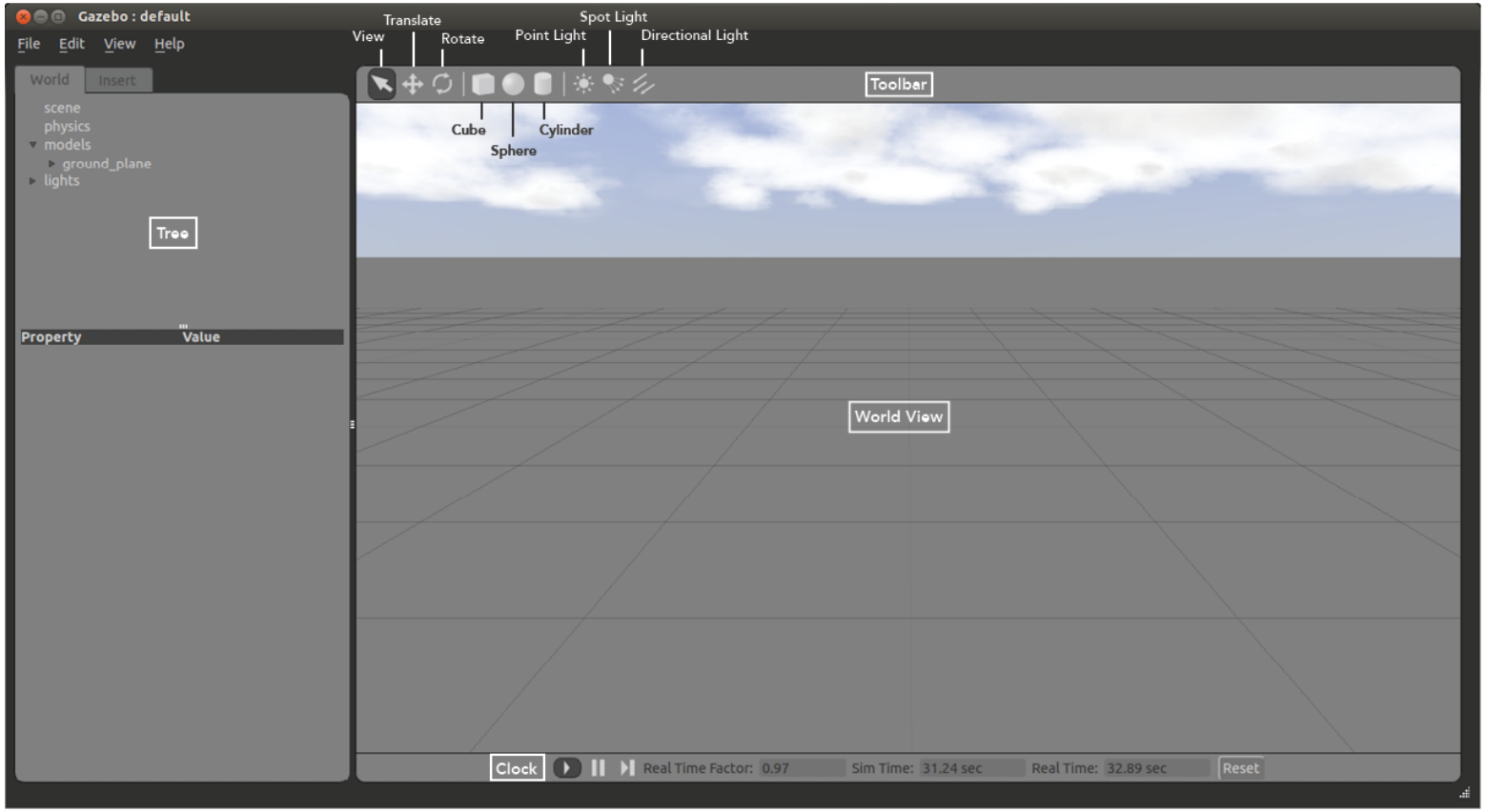
Running Gazebo from ROS

- 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

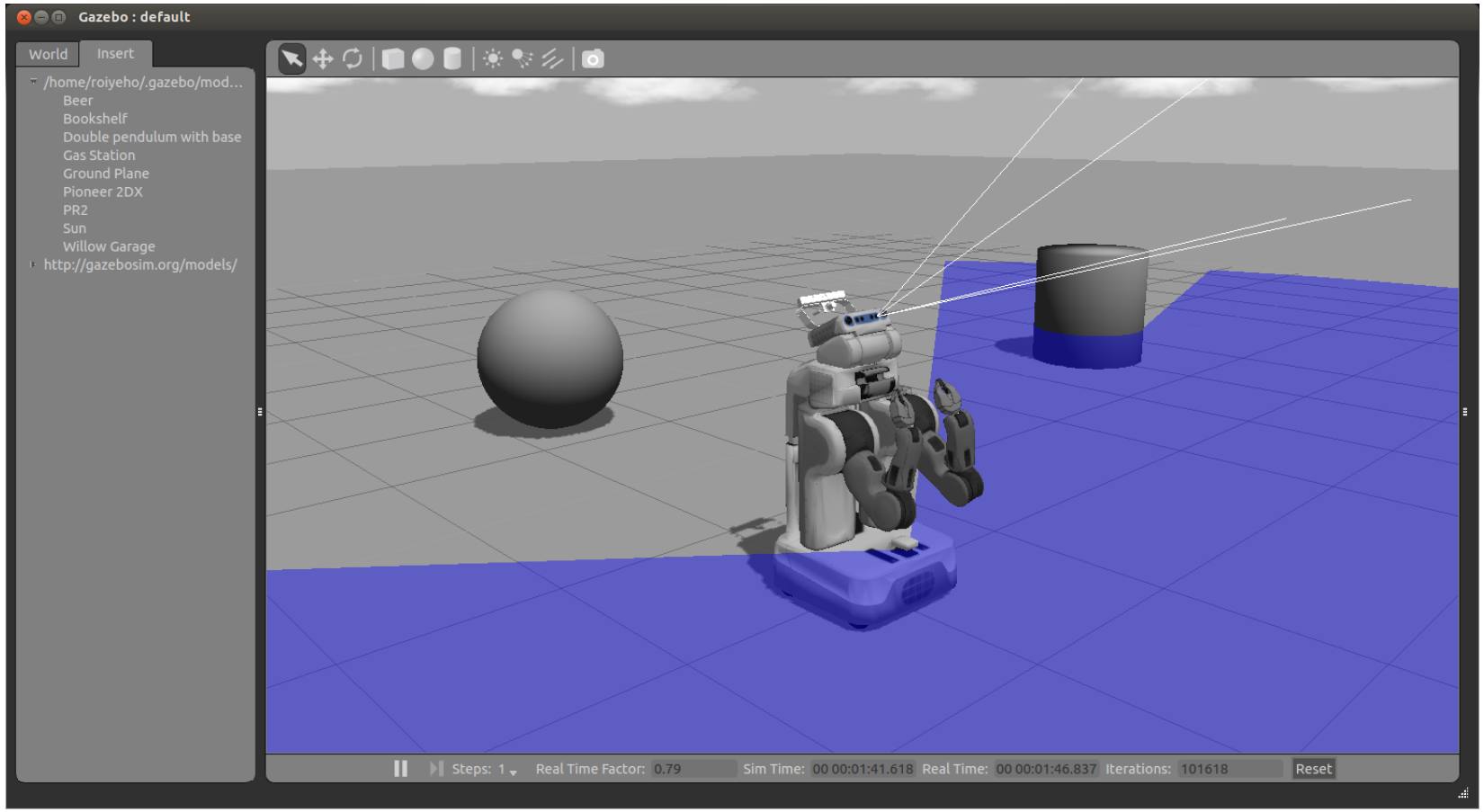
(Orbit & Zoom work in this mode, as well)

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

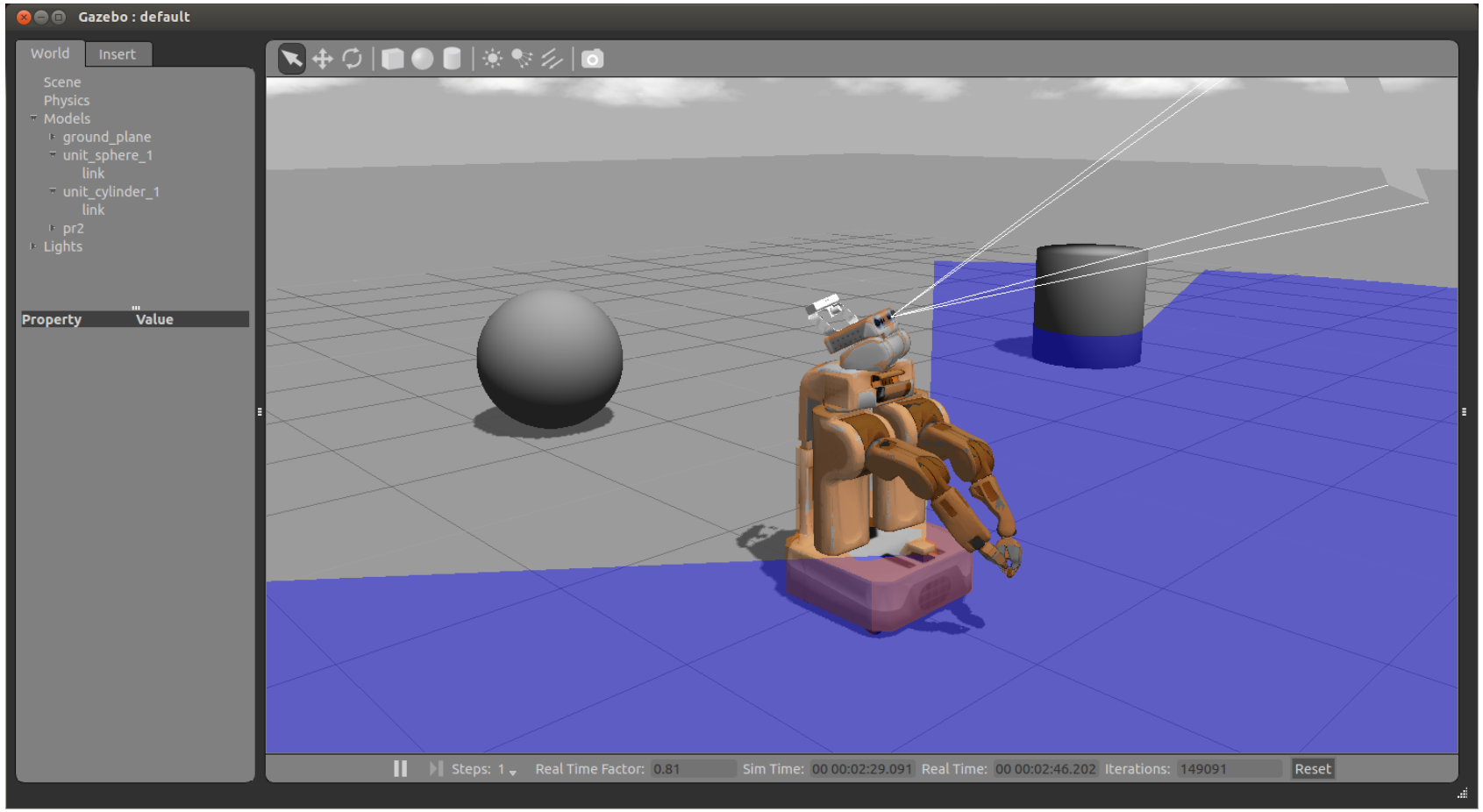
Inserting PR2 Robot



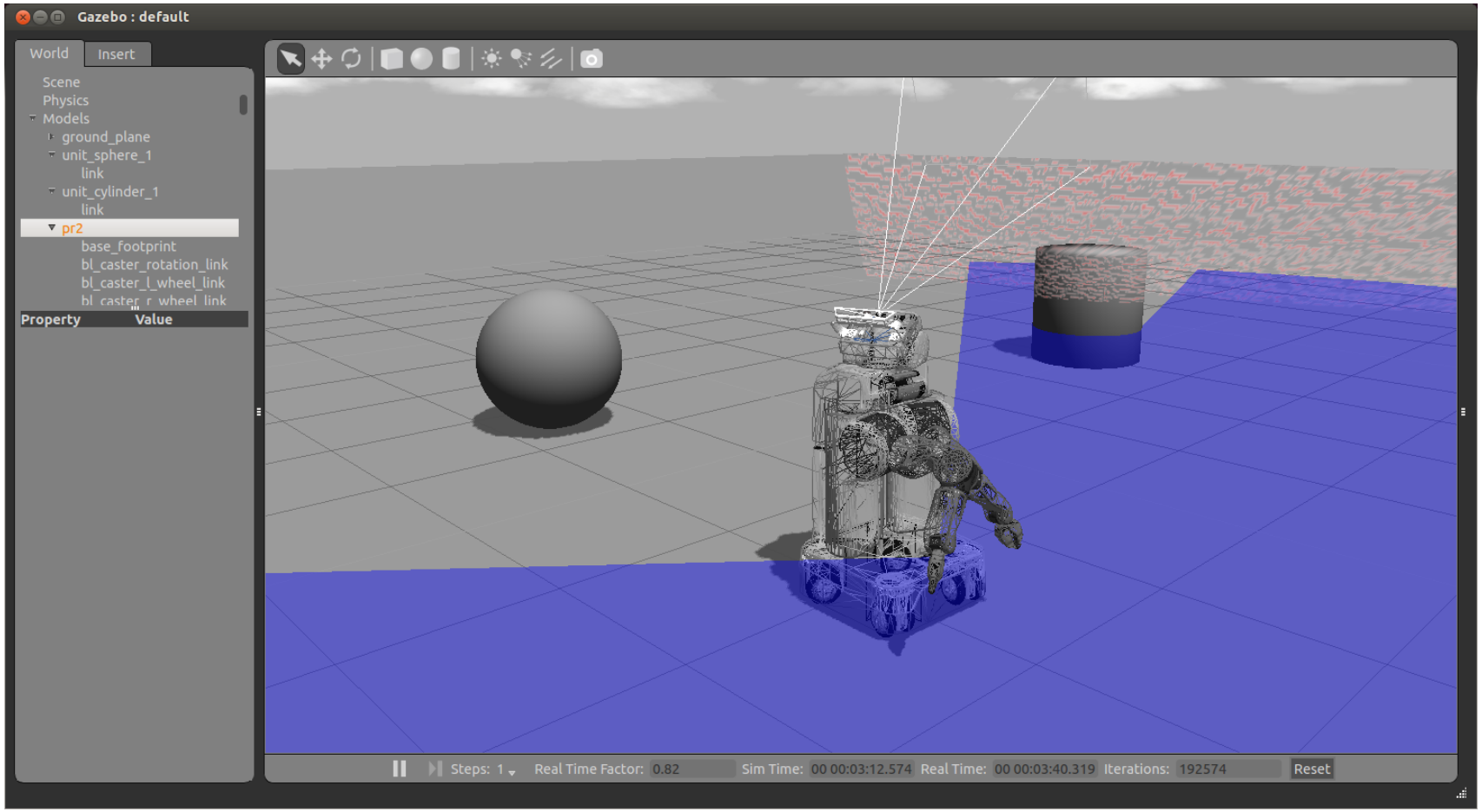
Models Item

- 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

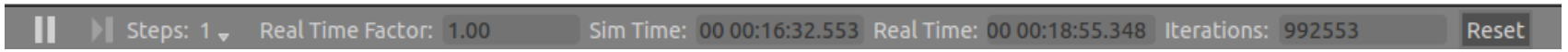


Wireframe View



Clock

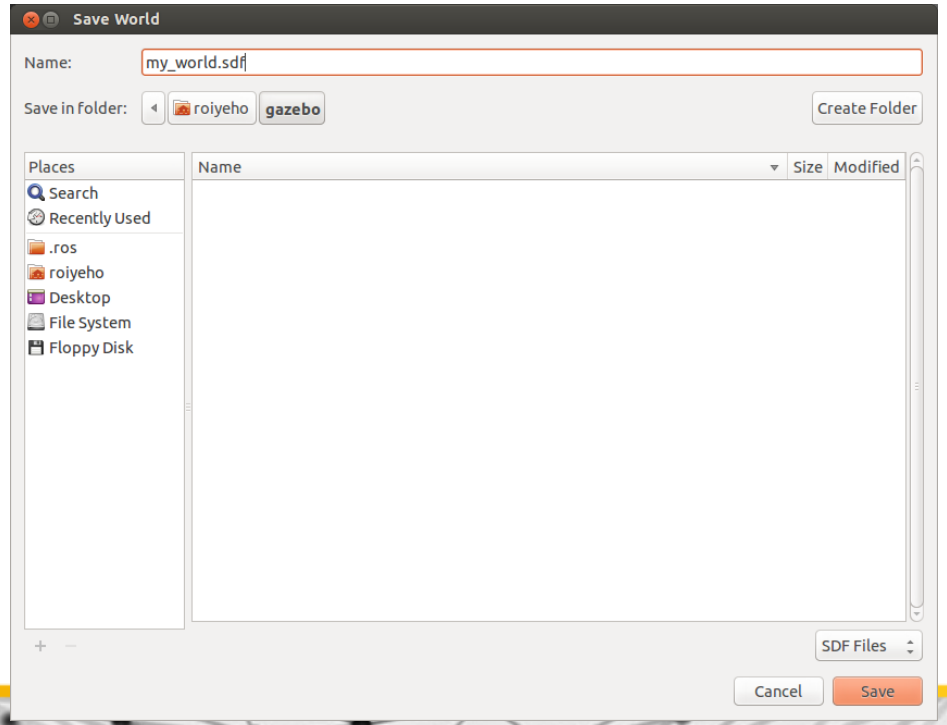
- You can start, pause and step through the simulation with the clock
- It is located at the bottom of the World View



- **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 then 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:

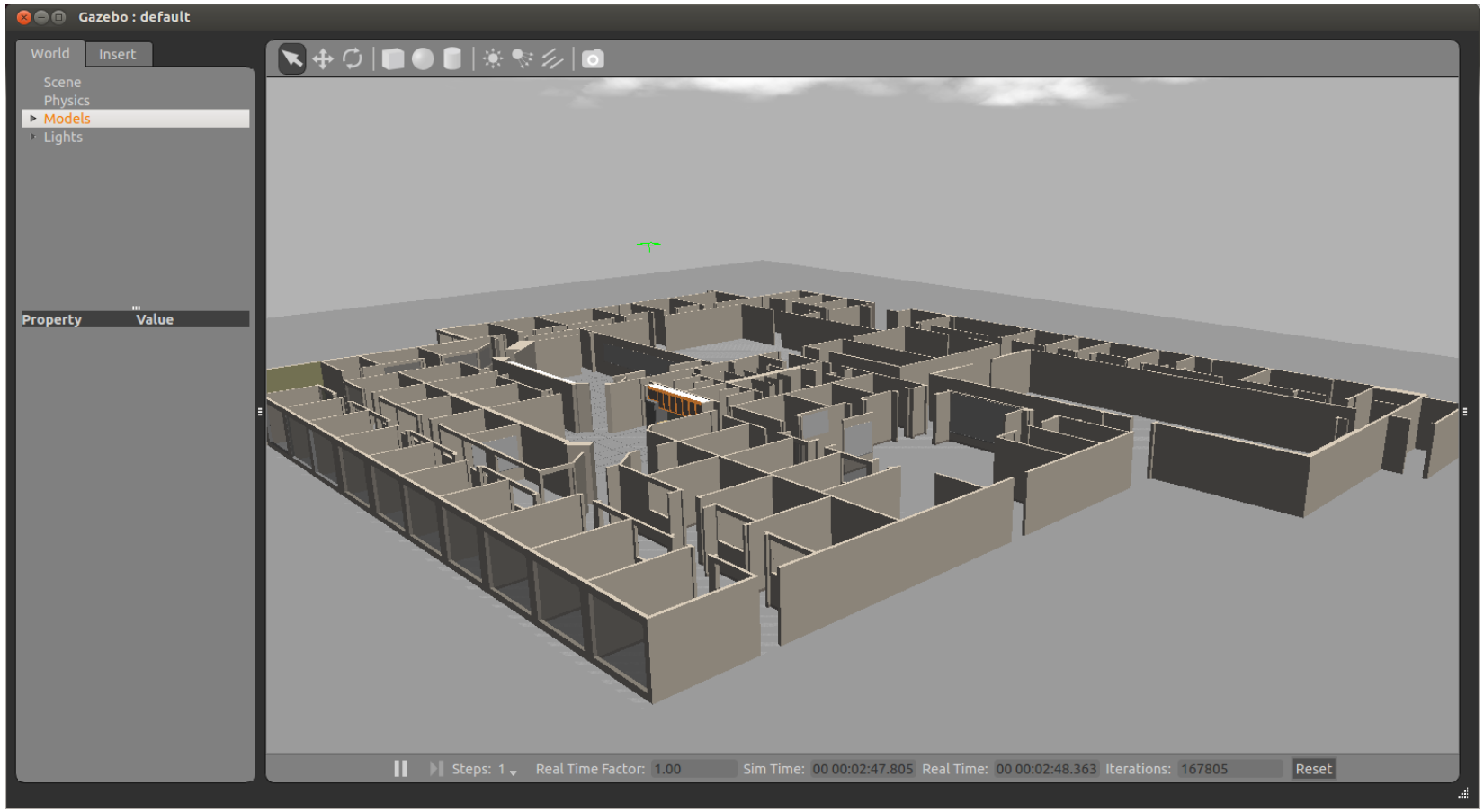
```
$ roslaunch gazebo_ros willowgarage_world.launch
```

willowgarage_world.launch

```
<launch>
  <!-- We resume 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 robo 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

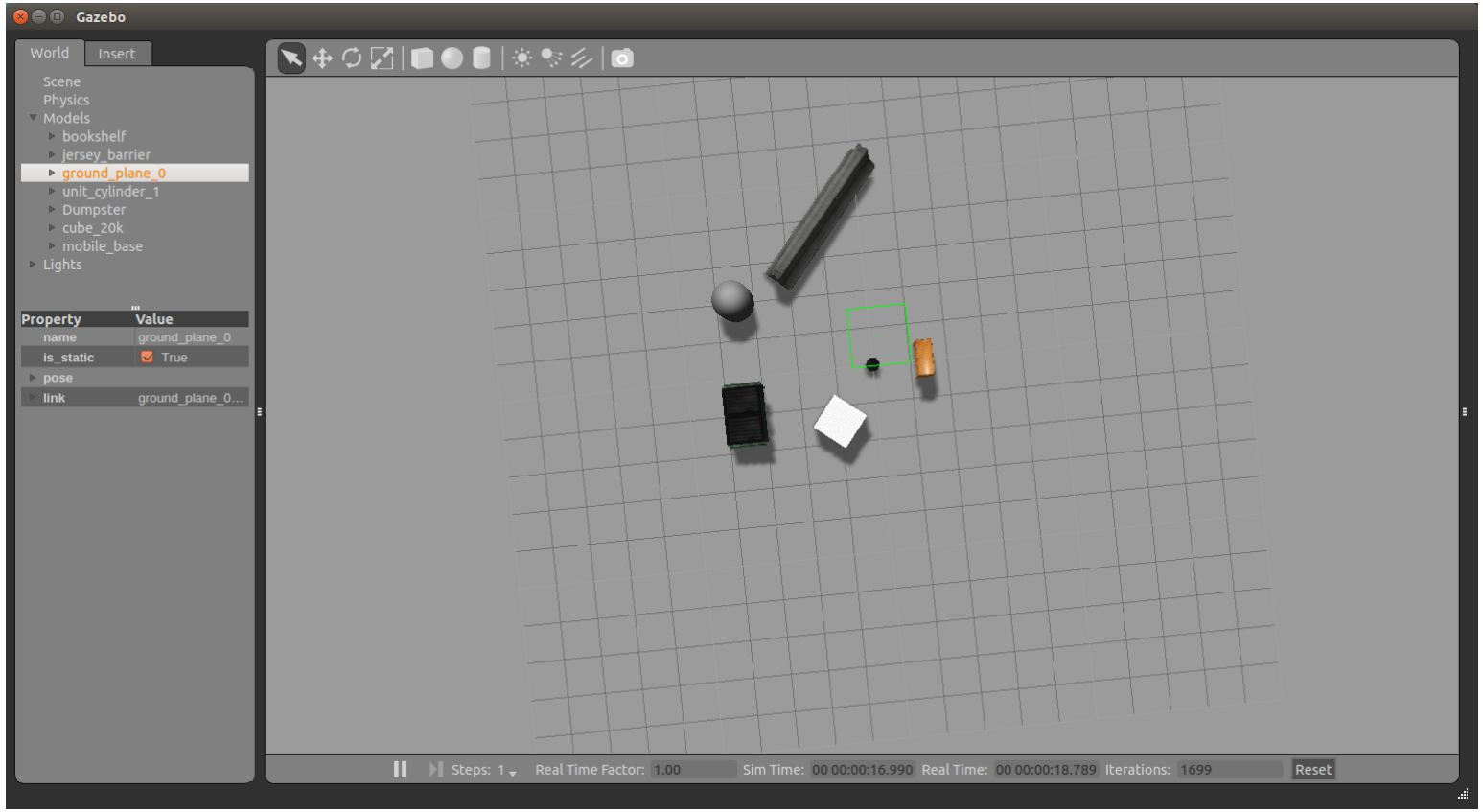
- To install Turtlebot simulation stack type:

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

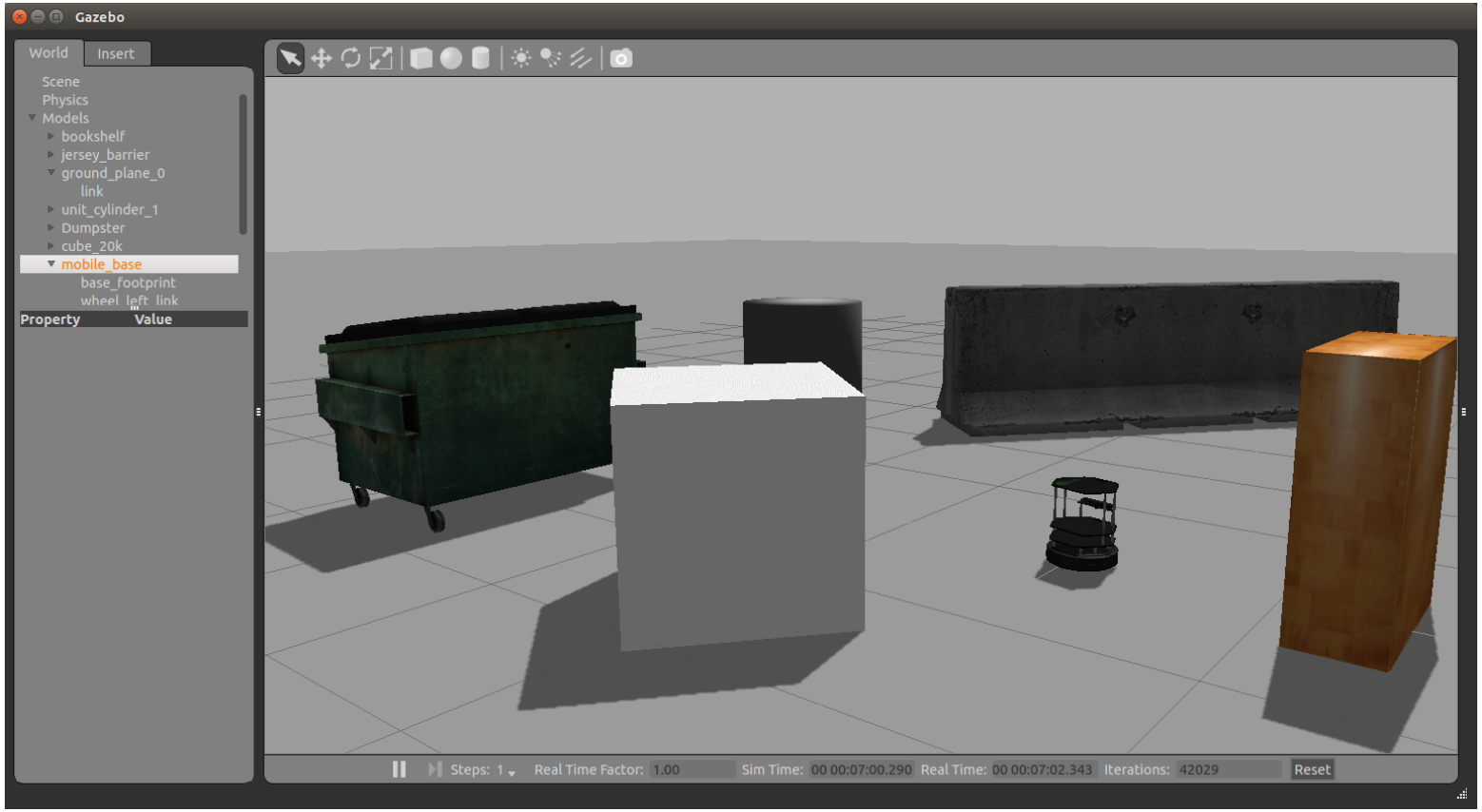
- To launch a simple world with a Turtlebot, type:

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

Turtlebot Simulation



Turtlebot Simulation

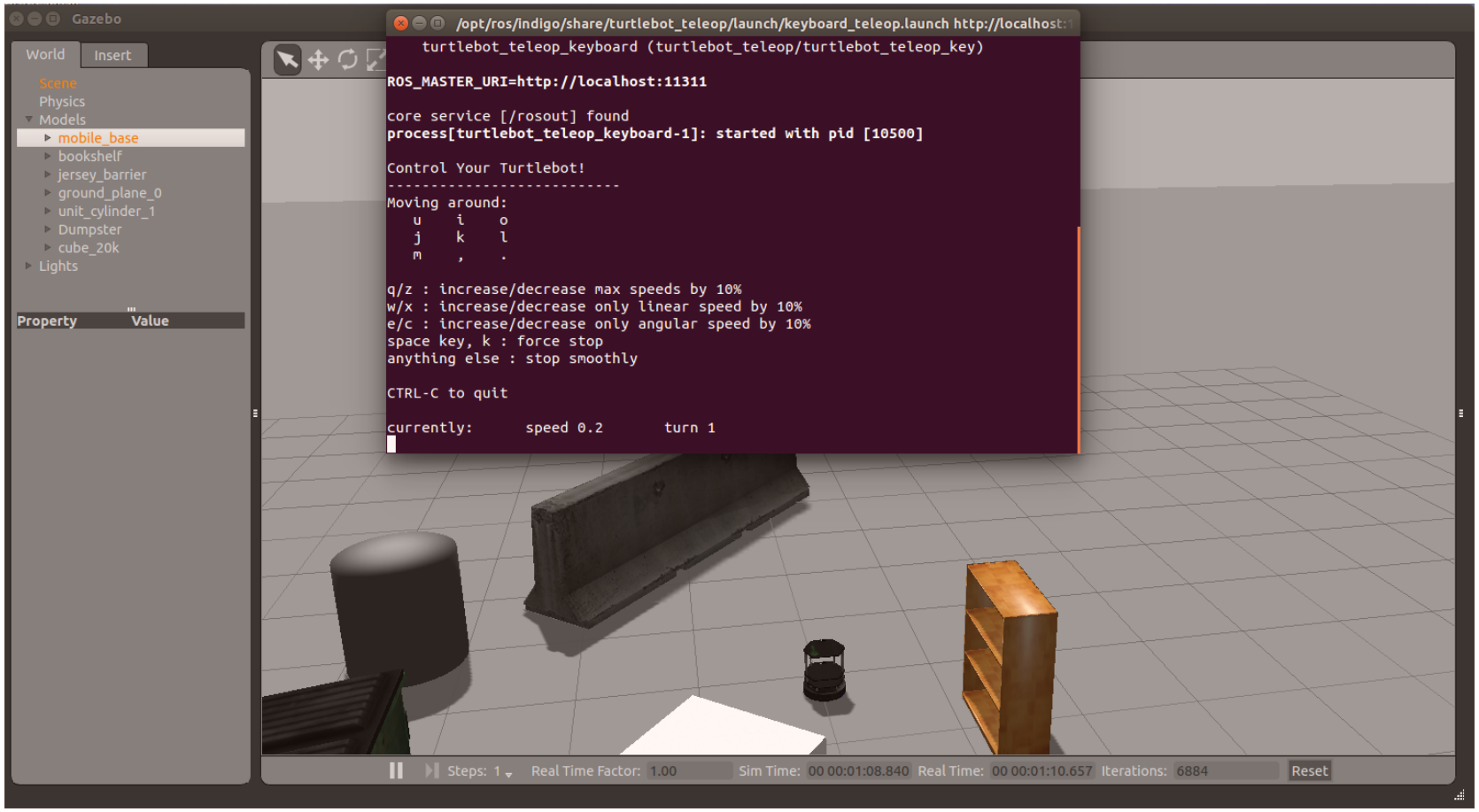


Moving Turtlebot with Teleop

- 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

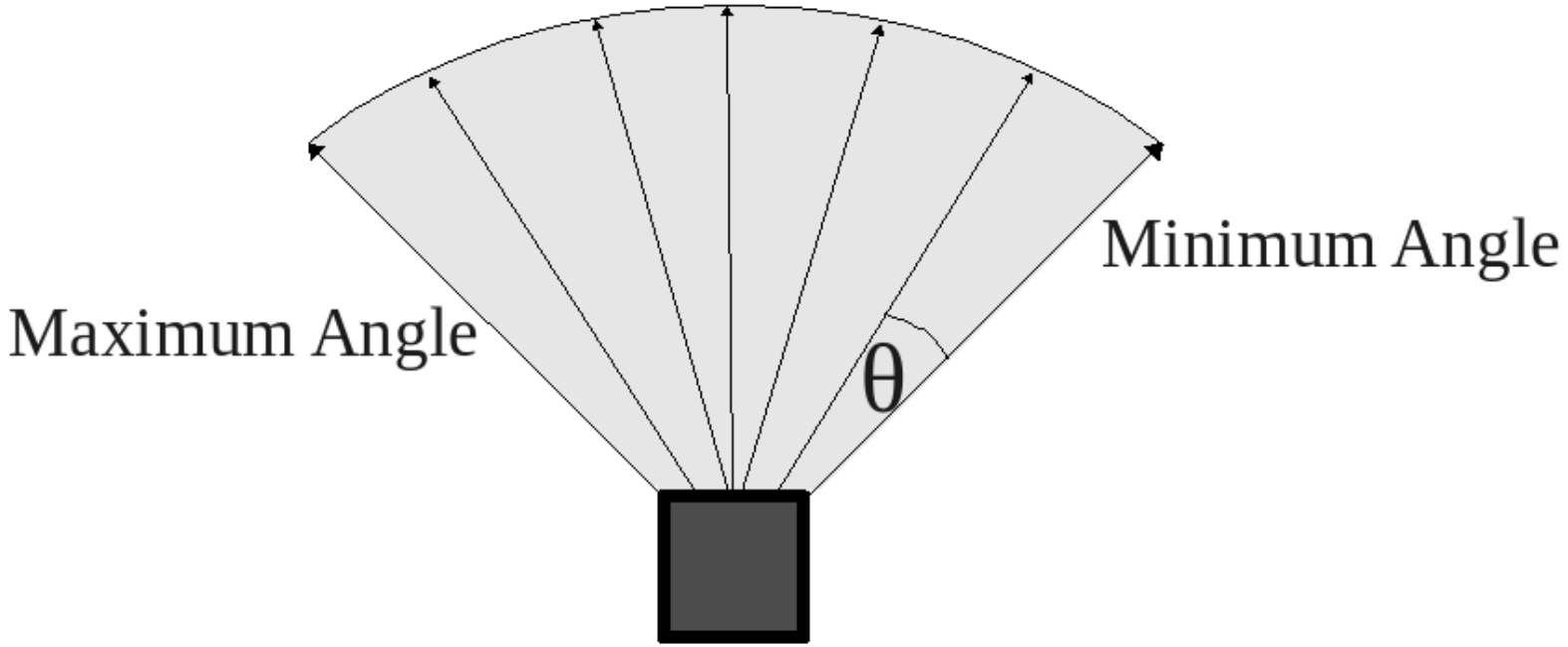
```
$ rosmsg show sensor_msgs/LaserScan
```

LaserScan Message

- http://docs.ros.org/api/sensor_msgs/html/msg/LaserScan.html

```
# 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

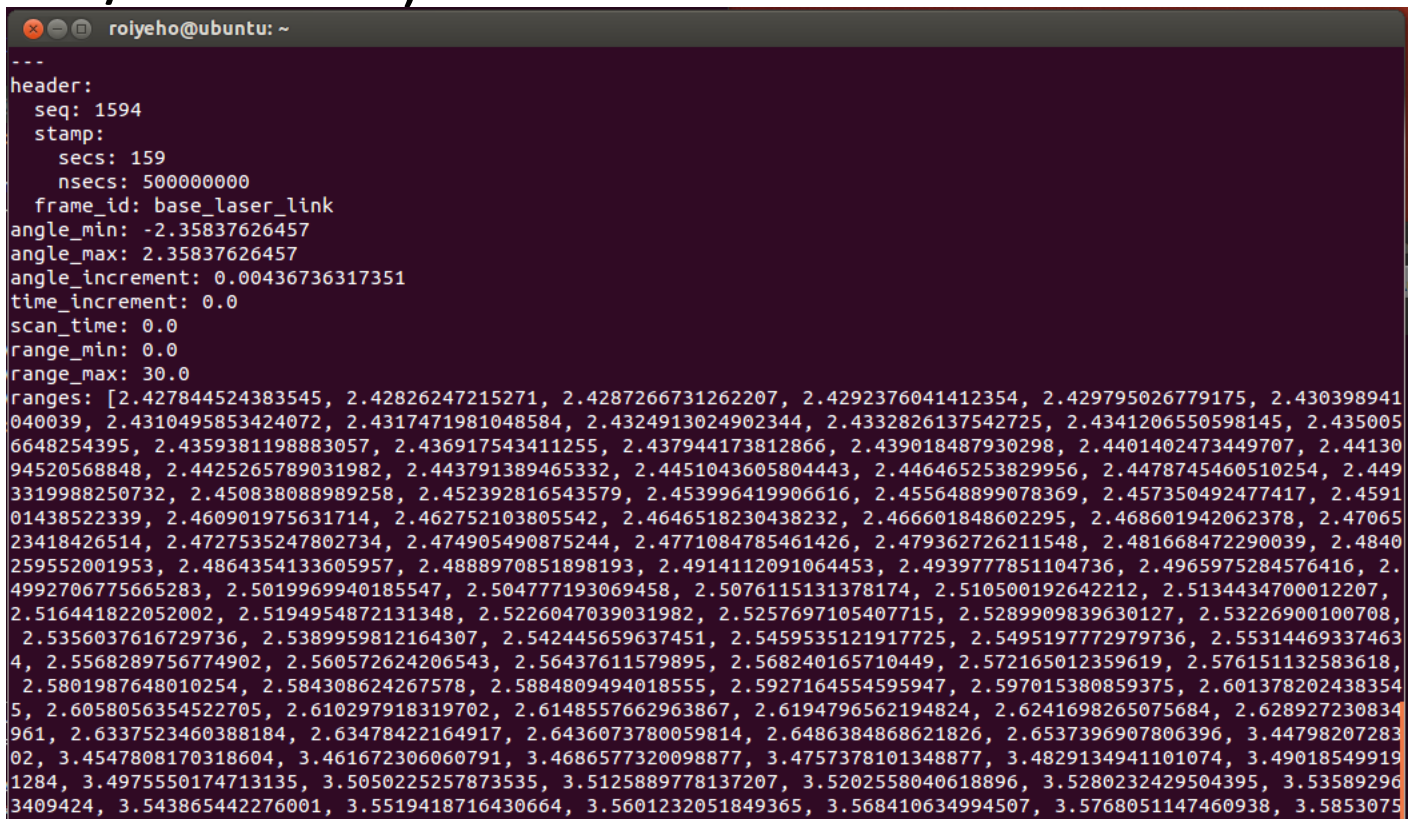
- A common laser sensor used in robotics
  http://www.hokuyo-aut.jp/02sensor/07scanner/urg_04lx.html

Hokuyo Laser

Model No.	URG-04LX
Power source	5VDC ± 5%*1
Current consumption	500mA or less(800mA when start-up)
Measuring area	60 to 4095mm(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(λ =785nm), 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)*2
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^2, 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*3

LaserScan Message

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



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)

```
viki@c3po: ~
viki@c3po:~$ rostopic echo /scan -n1
header:
  seq: 115
  stamp:
    secs: 27
    nsecs: 290000000
  frame_id: /camera_depth_frame
angle_min: -0.521567881107
angle_max: 0.524276316166
angle_increment: 0.00163668883033
time_increment: 0.0
scan_time: 0.0329999998212
range_min: 0.449999988079
range_max: 10.0
ranges: [nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
an, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, n
```

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

A Stopper Node

- We'll start with a node called **stopper** that will make the robot move forward until it detects an obstacle in front of it
- We will use the laser sensor to detect the obstacle
- Create a new package called wander_bot

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg wander_bot std_msgs rospy roscpp
```

A Stopper Node

- In Eclipse under the package's src directory add a new class named Stopper
  - This will create Stopper.h and Stopper.cpp
- Add a source file run_stopper.cpp
  - This will contain the main function

Stopper.h


```
#include "ros/ros.h"
#include "sensor_msgs/LaserScan.h"

class Stopper {
public:
    // Tunable parameters
    const static double FORWARD_SPEED = 0.5;
    const static double MIN_SCAN_ANGLE = -30.0/180*M_PI;
    const static double MAX_SCAN_ANGLE = +30.0/180*M_PI;
    const static float MIN_DIST_FROM_OBSTACLE = 0.5; // Should be smaller
than sensor_msgs::LaserScan::range_max

    Stopper();
    void startMoving();

private:
    ros::NodeHandle node;
    ros::Publisher commandPub; // Publisher to the robot's velocity command
topic
    ros::Subscriber laserSub; // Subscriber to the robot's laser scan topic
    bool keepMoving; // Indicates whether the robot should continue moving

    void moveForward();
    void scanCallback(const sensor_msgs::LaserScan::ConstPtr& scan);
};
```

Stopper.cpp (1)

```
#include "Stopper.h"
#include "geometry_msgs/Twist.h"

Stopper::Stopper()
{
    keepMoving = true;

    // Advertise a new publisher for the robot's velocity command topic
    commandPub = node.advertise<geometry_msgs::Twist>("
/cmd_vel_mux/input/teleop", 10);

    // Subscribe to the simulated robot's laser scan topic
    laserSub = node.subscribe("scan", 1, &Stopper::scanCallback, this);
}

// Send a velocity command
void Stopper::moveForward() {
    geometry_msgs::Twist msg; // The default constructor will set all
commands to 0
    msg.linear.x = FORWARD_SPEED_MPS;
    commandPub.publish(msg);
}
```

Stopper.cpp (2)

```
// Process the incoming laser scan message
void Stopper::scanCallback(const sensor_msgs::LaserScan::ConstPtr& scan)
{
    bool isObstacleInFront = false;

    // Find the closest range between the defined minimum and maximum angles
    int minIndex = ceil((MIN_SCAN_ANGLE - scan->angle_min) / scan-
>angle_increment);
    int maxIndex = floor((MAX_SCAN_ANGLE - scan->angle_min) / scan-
>angle_increment);

    for (int currIndex = minIndex + 1; currIndex <= maxIndex; currIndex++) {
        if (scan->ranges[currIndex] < MIN_DIST_FROM_OBSTACLE) {
            isObstacleInFront = true;
            break;
        }
    }

    if (isObstacleInFront) {
        ROS_INFO("Stop!");
        keepMoving = false;
    }
}
```

Stopper.cpp (3)

```
void Stopper::startMoving()
{
    ros::Rate rate(10);
    ROS_INFO("Start moving");

    // Keep spinning loop until user presses Ctrl+C or the robot got too
close to an obstacle
    while (ros::ok() && keepMoving) {
        moveForward();
        ros::spinOnce(); // Need to call this function often to allow ROS to
process incoming messages
        rate.sleep();
    }
}
```

run_stopper.cpp

```
#include "Stopper.h"

int main(int argc, char **argv) {
    // Initiate new ROS node named "stopper"
    ros::init(argc, argv, "stopper");

    // Create new stopper object
    Stopper stopper;

    // Start the movement
    stopper.startMoving();

    return 0;
};
```

CMakeLists.txt

- Edit the following lines in CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8.3)
project(my_stage)

…

## Declare a cpp executable
add_executable(stopper src/Stopper.cpp src/run_stopper.cpp)

## Specify libraries to link a library or executable target against
target_link_libraries(stopper ${catkin_LIBRARIES})
```

Launch File

- Launch file for launching both Gazebo and the stopper node:

```
<launch>
   <param name="/use_sim_time" value="true" />
   <!-- Launch turtle bot world -->
   <include file="$(find turtlebot_gazebo)/launch/turtlebot_world.launch"/>

   <!-- Launch stopper node -->
   <node name="stopper" pkg="wander_bot" type="stopper" output="screen"/>
</launch>
```

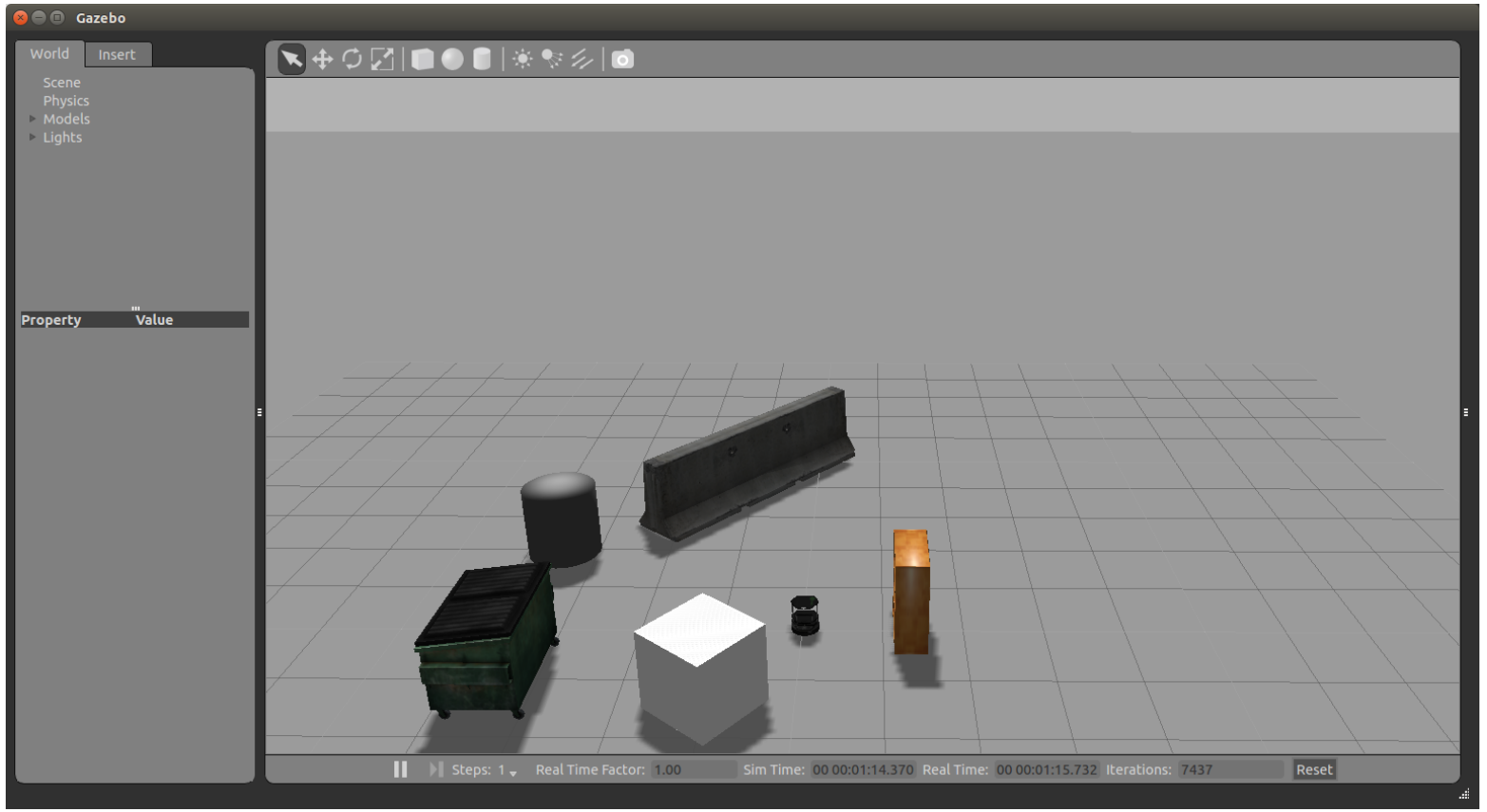
- To run the launch file:

```
$ roslaunch wander_bot stopper.launch
```

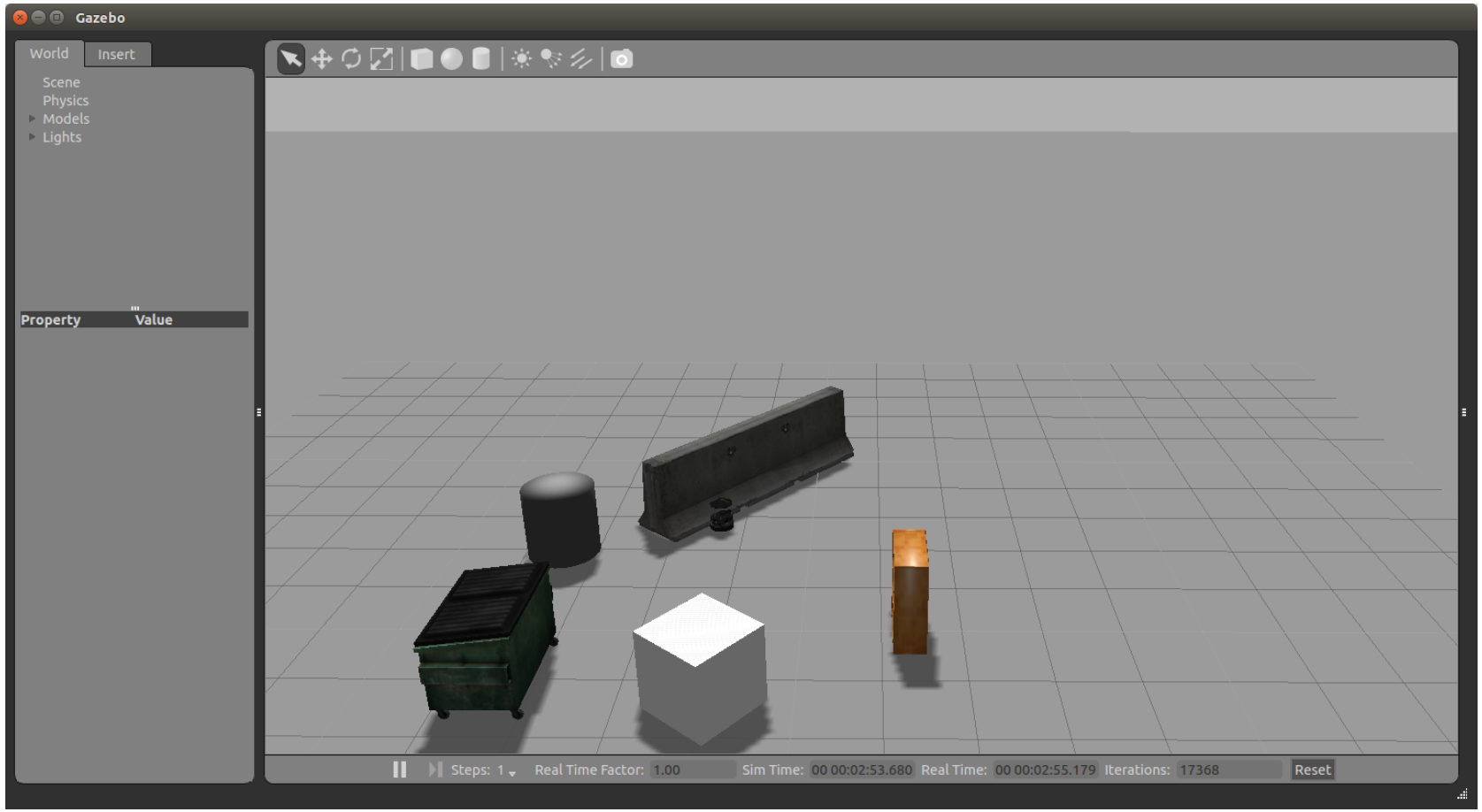
ROS Clock

- Normally, ROS client libraries use the computer's system clock as the "wall-clock“
- When you are running a simulation or playing back logged data, it is often desirable to instead have the system use a simulated clock so that you can have accelerated, slowed, or stepped control over your system's perceived time
- To support this, the ROS client libraries can listen to the /clock topic that is used to publish "simulation time“
- For that purpose, set the **/use_sim_time** parameter to true *before the node is initialized*

Turtlebot Initial Position



Turtlebot Final Position



Ex. 4 (For Submission)

- Implement a simple walker algorithm much like a Roomba robot vaccum 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

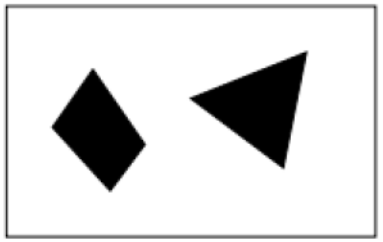
Why Mapping?

- Building maps is one of the fundamental problems in mobile robotics
- Maps allow robots to efficiently carry out their tasks, such as localization, path planning, activity planning, etc.
- There are different ways to create a map of the environment

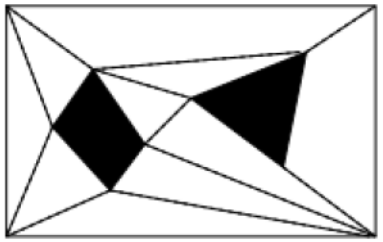
Cellular Decomposition

- Decompose free space for path planning
- Exact decomposition
  - Cover the free space exactly
  - Example: trapezoidal decomposition, meadow map
- Approximate decomposition
  - Represent part of the free space, needed for navigation
  - Example: grid maps, quadtrees, Voronoi graphs

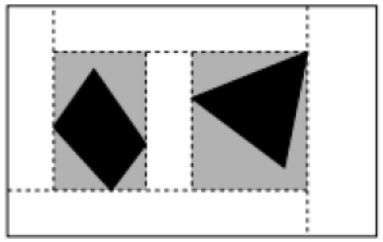
Cellular Decomposition



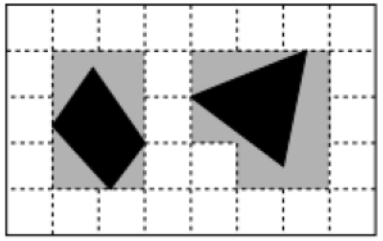
Metric map of the environment



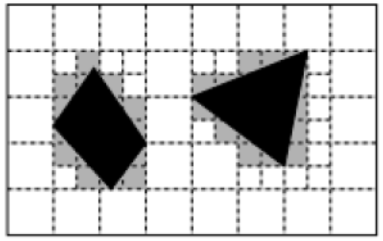
Exact cell decomposition



Rectangular cell decomposition



Regular cell decomposition

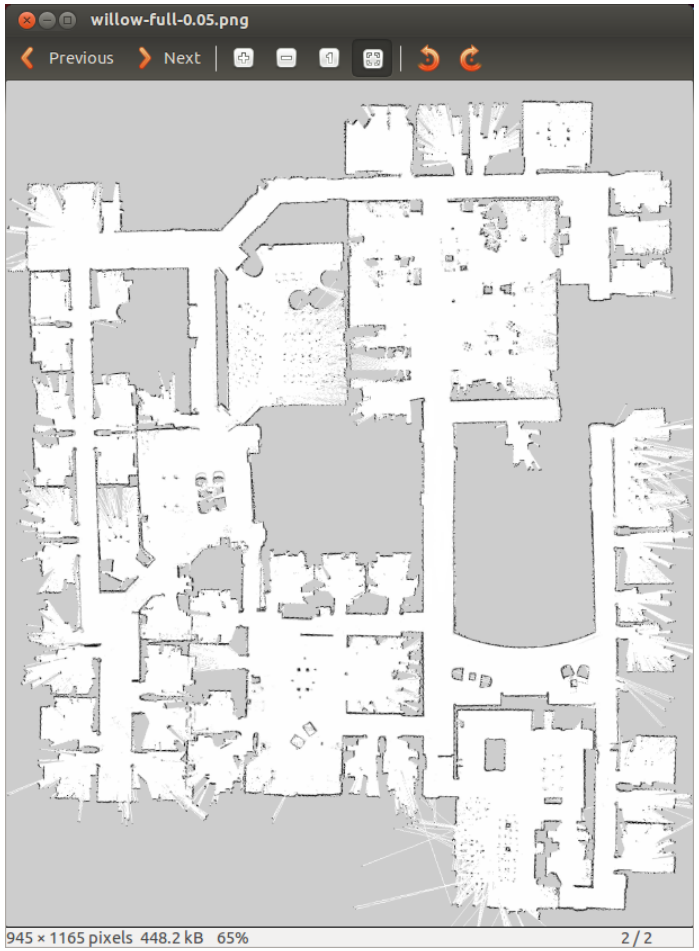


Quadtree decomposition

Occupancy Grid Map (OGM)

- Maps the environment as a grid of cells
  - Cell sizes typically range from 5 to 50 cm
- Each cell holds a probability value that the cell is occupied in the range [0,100]
- Unknown is indicated by -1
  - Usually unknown areas are areas that the robot sensors cannot detect (beyond obstacles)

Occupancy Grid Map



White pixels represent free cells
Black pixels represent occupied cells
Gray pixels are in unknown state

Occupancy Grid Maps

- Pros:
  - Simple representation
  - Speed
- Cons:
  - Not accurate - if an object falls inside a portion of a grid cell, the whole cell is marked occupied
  - Wasted space

Maps in ROS

- Map files are stored as images, with a variety of common formats being supported (such as PNG, JPG, and PGM)
- Although color images can be used, they are converted to grayscale images before being interpreted by ROS
- Associated with each map is a YAML file that holds additional information about the map

Map YAML File

```
image: map.pgm
resolution: 0.050000
origin: [-100.000000, -100.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

resolution: Resolution of the map, meters / pixel

origin: 2D pose of the lower-left pixel as (x, y, yaw)

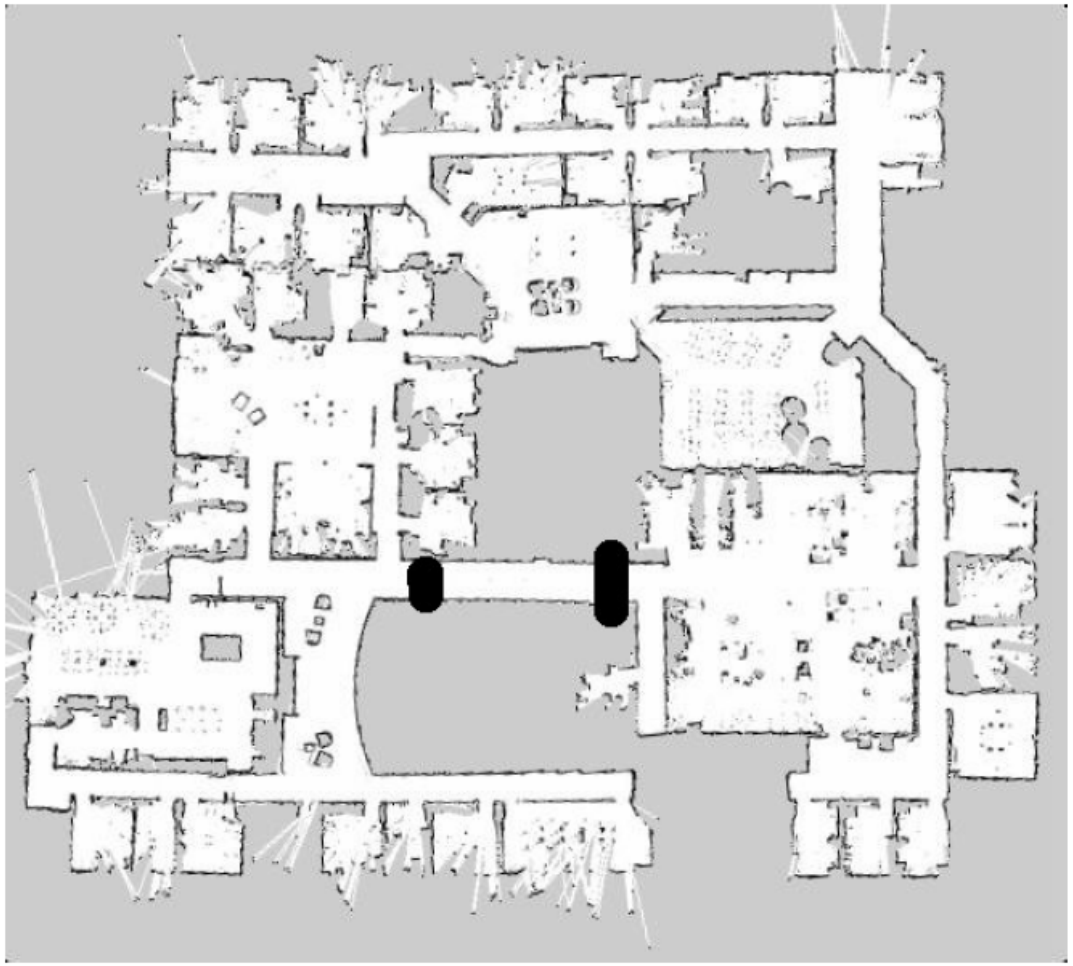
occupied_thresh: Pixels with occupancy probability greater than this threshold are considered completely occupied

free_thresh: Pixels with occupancy probability less than this threshold are considered completely free

Editing Map Files

- Since maps are represented as image files, you can edit them in your favorite image editor
- This allows you to tidy up any maps that you create from sensor data, removing things that shouldn’t be there, or adding in fake obstacles to influence path planning
- For example, you can stop the robot from planning paths through certain areas of the map by drawing a line across a corridor you don’t want to the robot to drive through

Editing Map Files



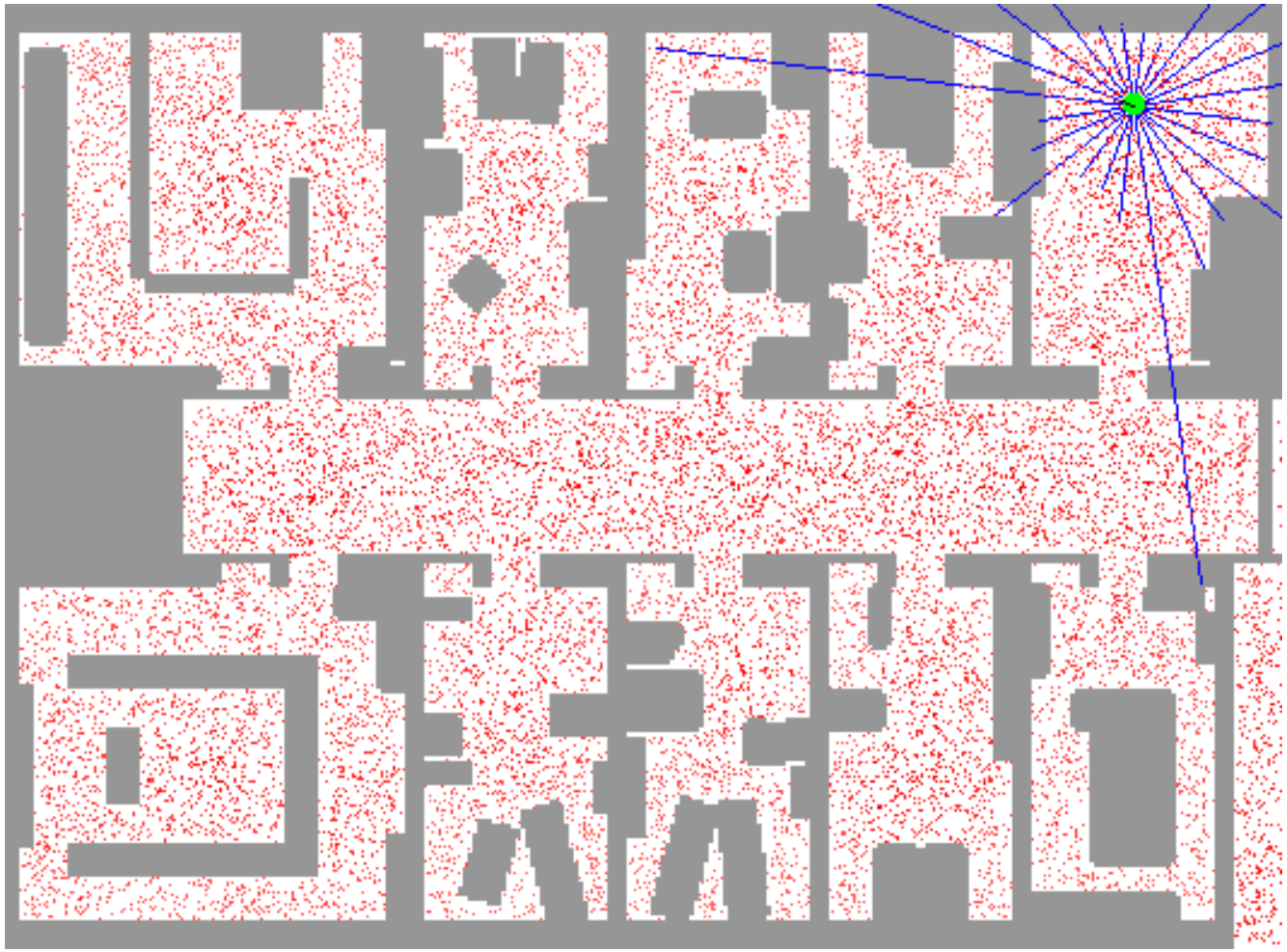
SLAM

- **Simultaneous localization and mapping** (**SLAM**) is a technique used by robots to build up a map within an unknown environment while at the same time keeping track of their current location
- A chicken or egg problem: An unbiased map is needed for localization while an accurate pose estimate is needed to build that map

Particle Filter – FastSLAM

- Represent probability distribution as a set of discrete particles which occupy the state space
- Main steps of the algorithm:
  - Start with a random distribution of particles
  - Compare particle's prediction of measurements with actual measurements
  - Assign each particle a weight depending on how well its estimate of the state agrees with the measurements
  - Randomly draw particles from previous distribution based on weights creating a new distribution
- **Efficient**: scales logarithmically with the number of landmarks in the map

Particle Filter



gmapping

- http://wiki.ros.org/gmapping
- The gmapping package provides laser-based SLAM as a ROS node called **slam_gmapping**
- Uses the FastSLAM algorithm
- It takes the laser scans and the odometry and builds a 2D occupancy grid map
- It updates the map state while the robot moves
- ROS with gmapping video

Install gmapping

- gmapping is not part of ROS Indigo installation
- To install gmapping run:

```
$ sudo apt-get install ros-indigo-slam-gmapping
```

  - You may need to run sudo apt-get update before that to update package repositories list

Run gmapping

- First launch Gazebo with turtlebot

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

- Now start gmapping in a new terminal window

```
$ rosrun gmapping slam_gmapping
```

```
viki@c3po:~$ rosrun gmapping slam_gmapping
[ INFO] [1482222664.730535530, 36.260000000]: Laser is mounted upwards.
 -maxUrange 9.99 -maxUrange 9.99 -sigma     0.05 -kernelSize 1 -lstep 0.05 -lobs
Gain 3 -astep 0.05
 -srr 0.1 -srt 0.2 -str 0.1 -stt 0.2
 -linearUpdate 1 -angularUpdate 0.5 -resampleThreshold 0.5
 -xmin -100 -xmax 100 -ymin -100 -ymax 100 -delta 0.05 -particles 30
[ INFO] [1482222664.748575358, 36.280000000]: Initialization complete
update frame 0
update ld=0 ad=0
Laser Pose= -0.0858969 0.0494185 -0.022192
m_count 0
Registering First Scan
```

Run gmapping

- Now move the robot using teleop

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

- Check that the map is published to the topic /map

```
$ rostopic echo /map -n1
```

- Message type is nav_msgs/OccupancyGrid
- Occupancy is represented as an integer with:
  - 0 meaning completely free
  - 100 meaning completely occupied
  - the special value -1 for completely unknown

Run gmapping

map_server

- map_server allows you to load and save maps
- To install the package:

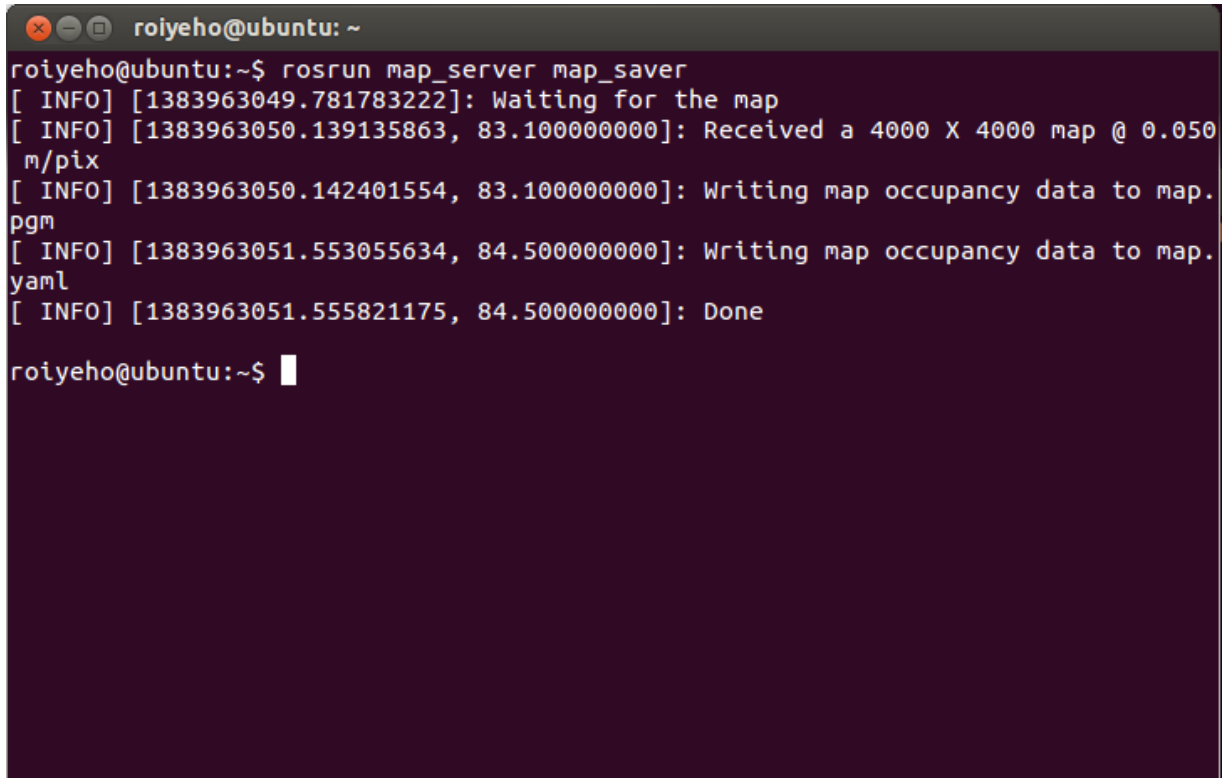
```
$ sudo apt-get install ros-kinetic-map-server
```

- To save dynamically generated maps to a file:

```
$ rosrun map_server map_saver [-f mapname]
```

- map_saver generates the following files in the current directory:
  - **map.pgm** – the map itself
  - **map.yaml** – the map's metadata

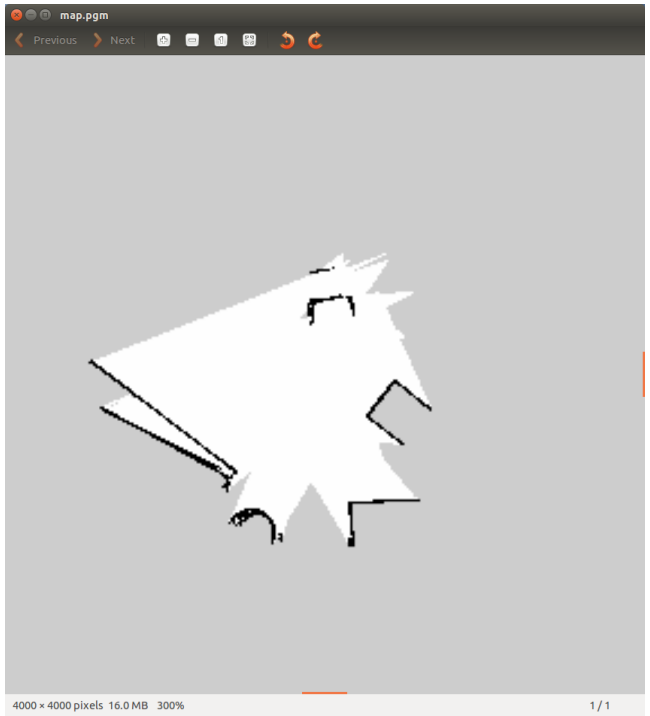
Saving the map using map_server



map_server

- You can open the pgm file with the default Ubuntu image viewer program (eog)

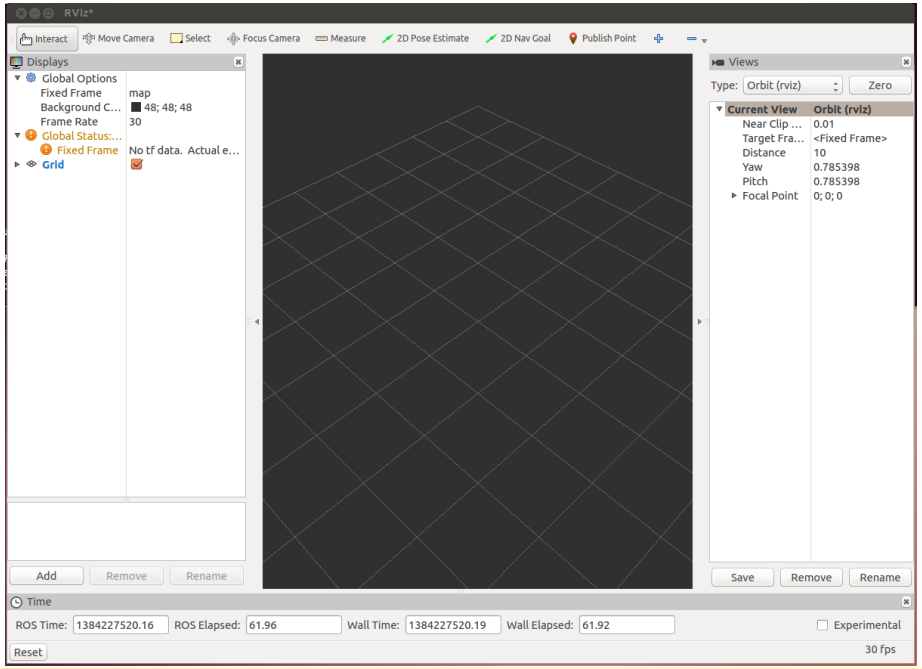
```
$ eog map.pgm
```



rviz

- rviz is a ROS 3D visualization tool that lets you see the world from a robot's perspective

```
$ rosrun rviz rviz
```



rviz Useful Commands

- Use right mouse button or scroll wheel to zoom in or out
- Use the left mouse button to pan (shift-click) or rotate (click)

rviz Displays

- The first time you open rviz you will see an empty 3D view
- On the left is the **Displays** area, which contains a list of different elements in the world, that appears in the middle
  - Right now it just contains global options and grid
- Below the Displays area, we have the **Add** button that allows the addition of more elements

rviz Displays

Display name	Description	Messages Used
Axes	Displays a set of Axes	
Effort	Shows the effort being put into each revolute joint of a robot.	sensor_msgs/JointStates
Camera	Creates a new rendering window from the perspective of a camera, and overlays the image on top of it.	sensor_msgs/Image sensor_msgs/CameraInfo
Grid	Displays a 2D or 3D grid along a plane	
Grid Cells	Draws cells from a grid, usually obstacles from a costmap from the navigation stack.	nav_msgs/GridCells
Image	Creates a new rendering window with an Image.	sensor_msgs/Image
LaserScan	Shows data from a laser scan, with different options for rendering modes, accumulation, etc.	sensor_msgs/LaserScan
Map	Displays a map on the ground plane.	nav_msgs/OccupancyGrid

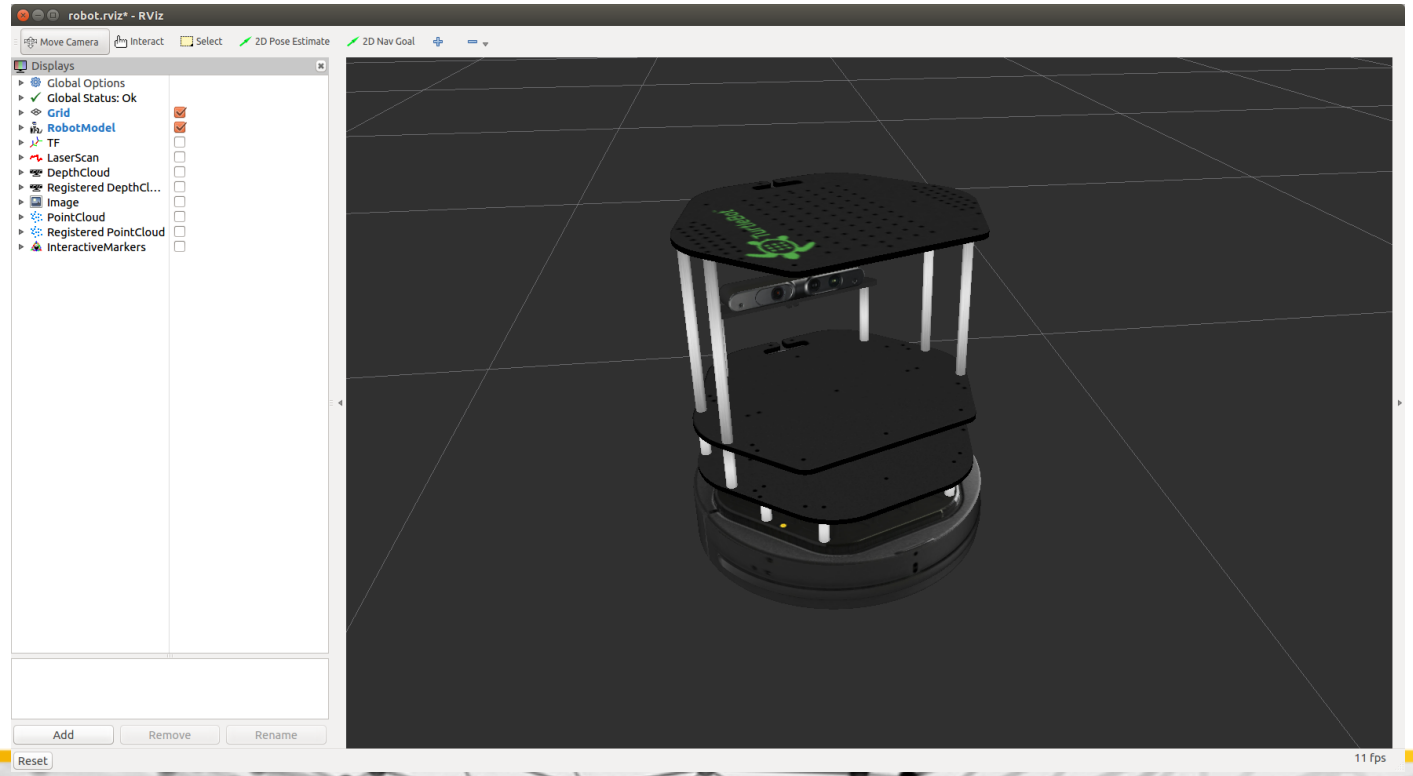
rviz Displays

Display name	Description	Messages Used
Markers	Allows programmers to display arbitrary primitive shapes through a topic	visualization_msgs/Marker visualization_msgs/MarkerArray
Path	Shows a path from the navigation stack.	nav_msgs/Path
Pose	Draws a pose as either an arrow or axes	geometry_msgs/PoseStamped
Point Cloud(2)	Shows data from a point cloud, with different options for rendering modes, accumulation, etc.	sensor_msgs/PointCloud sensor_msgs/PointCloud2
Odometry	Accumulates odometry poses from over time.	nav_msgs/Odometry
Range	Displays cones representing range measurements from sonar or IR range sensors.	sensor_msgs/Range
RobotModel	Shows a visual representation of a robot in the correct pose (as defined by the current TF transforms).	
TF	Displays the tf transform hierarchy.	

rviz with TurtleBot

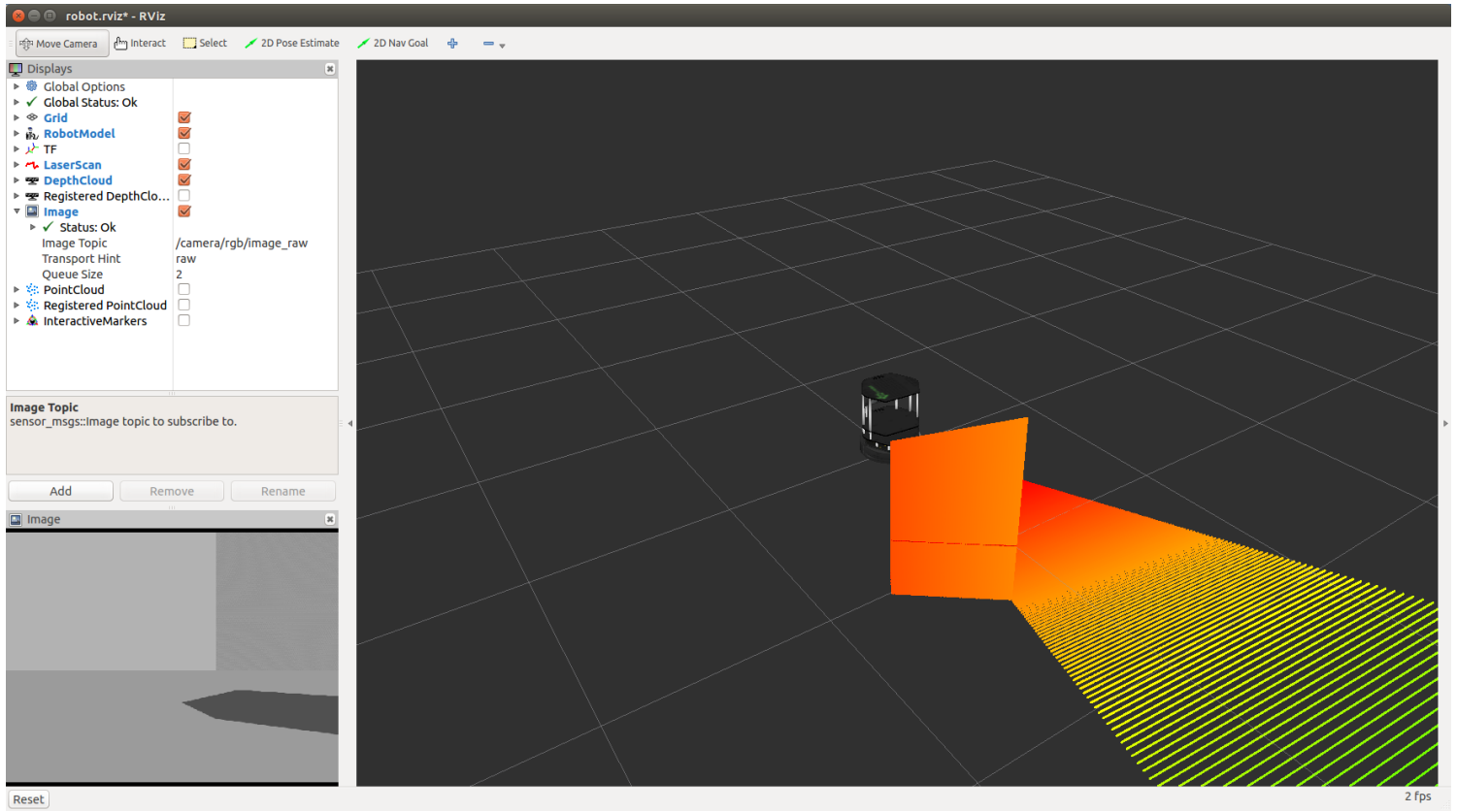
- You can start rviz already configured to visualize the robot and its sensor's output:

```
$ roslaunch turtlebot_rviz_launchers view_robot.launch
```



TurtleBot Image Display

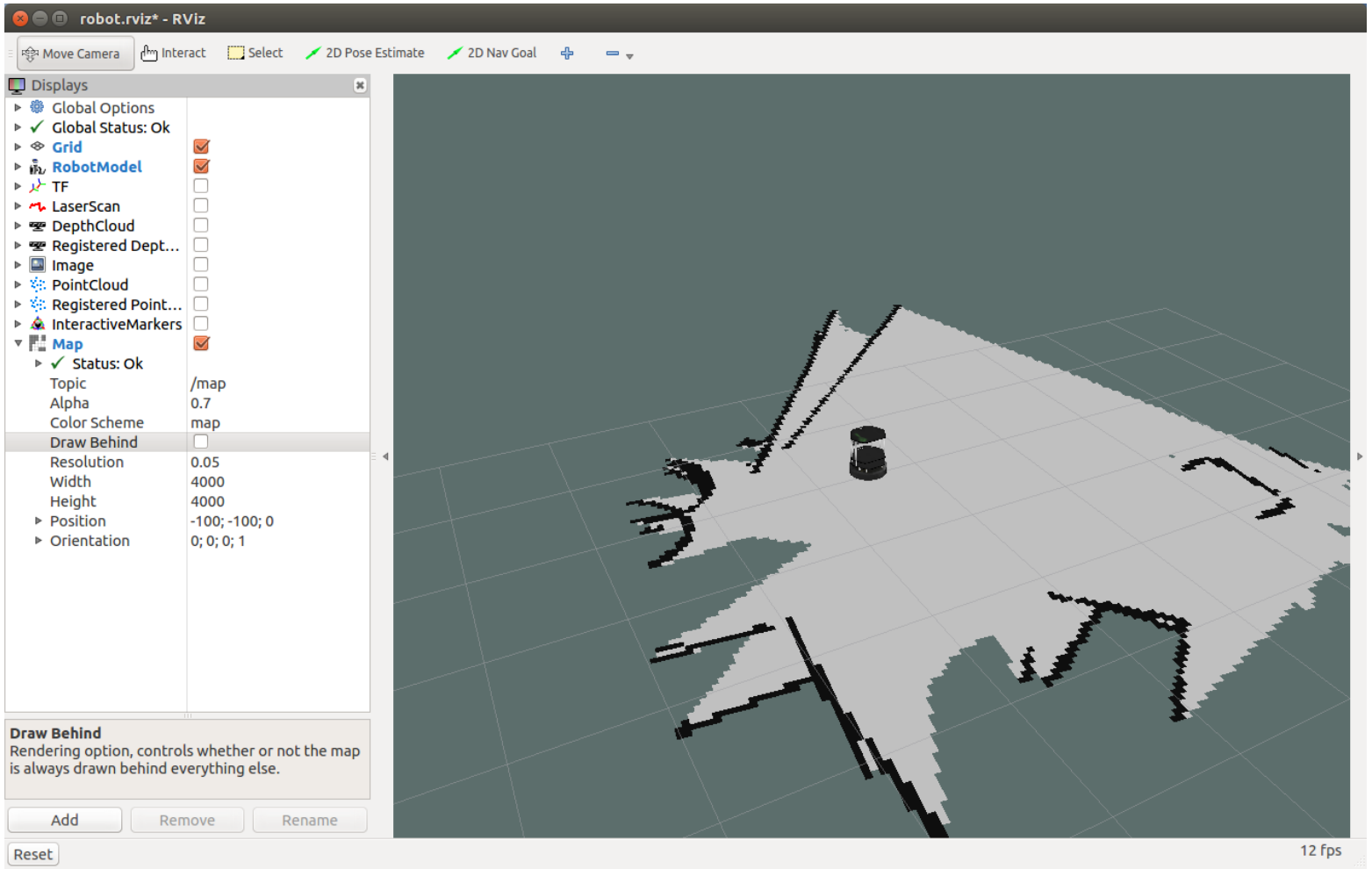
- To visualize any display you want, just click on its check button



Map Display

- Add the Map display
- Set the **topic** to /map
- Now you will be able to watch the mapping progress in rviz

Map Display


robot.rviz* - RViz
Move Camera
Interact
Select
2D Pose Estimate
2D Nav Goal
Displays
Global Options
Global Status: Ok
Grid
RobotModel
TF
LaserScan
DepthCloud
Registered Dept...
Image
PointCloud
Registered Point...
InteractiveMarkers
Map
Status: Ok
Topic /map
Alpha 0.7
Color Scheme map
Draw Behind
Resolution 0.05
Width 4000
Height 4000
Position -100; -100; 0
Orientation 0; 0; 0; 1
Draw Behind
Rendering option, controls whether or not the map is always drawn behind everything else.
Add
Remove
Rename
Reset
12 fps

Loading and Saving Configuration

- You can save your rviz settings by choosing File > Save Config from the menu
- Your settings will be saved to a .rviz file
- Then, you can start rviz with your saved configuration:

```
$ rosrun rviz rviz -d my_config.rviz
```

Launch File for gmapping

```
<launch>
  <!-- Run Gazebo with turtlebot -->
  <include file="$(find turtlebot_gazebo)/launch/turtlebot_world.launch"/>

  <!-- Run gmapping -->
  <node name="gmapping" pkg="gmapping" type="slam_gmapping"/>

  <!-- Open rviz -->
  <include file="$(find turtlebot_rviz_launchers)/launch/view_robot.launch"/>
</launch>
```

Loading an Existing Map

- Now instead of running gmapping, we will load the existing Turtlebot's playground map
- The map_server node in map_server package allows you to load existing maps
  - Takes as arguments the path to the map file and the map resolution (if not specified in the YAML file)

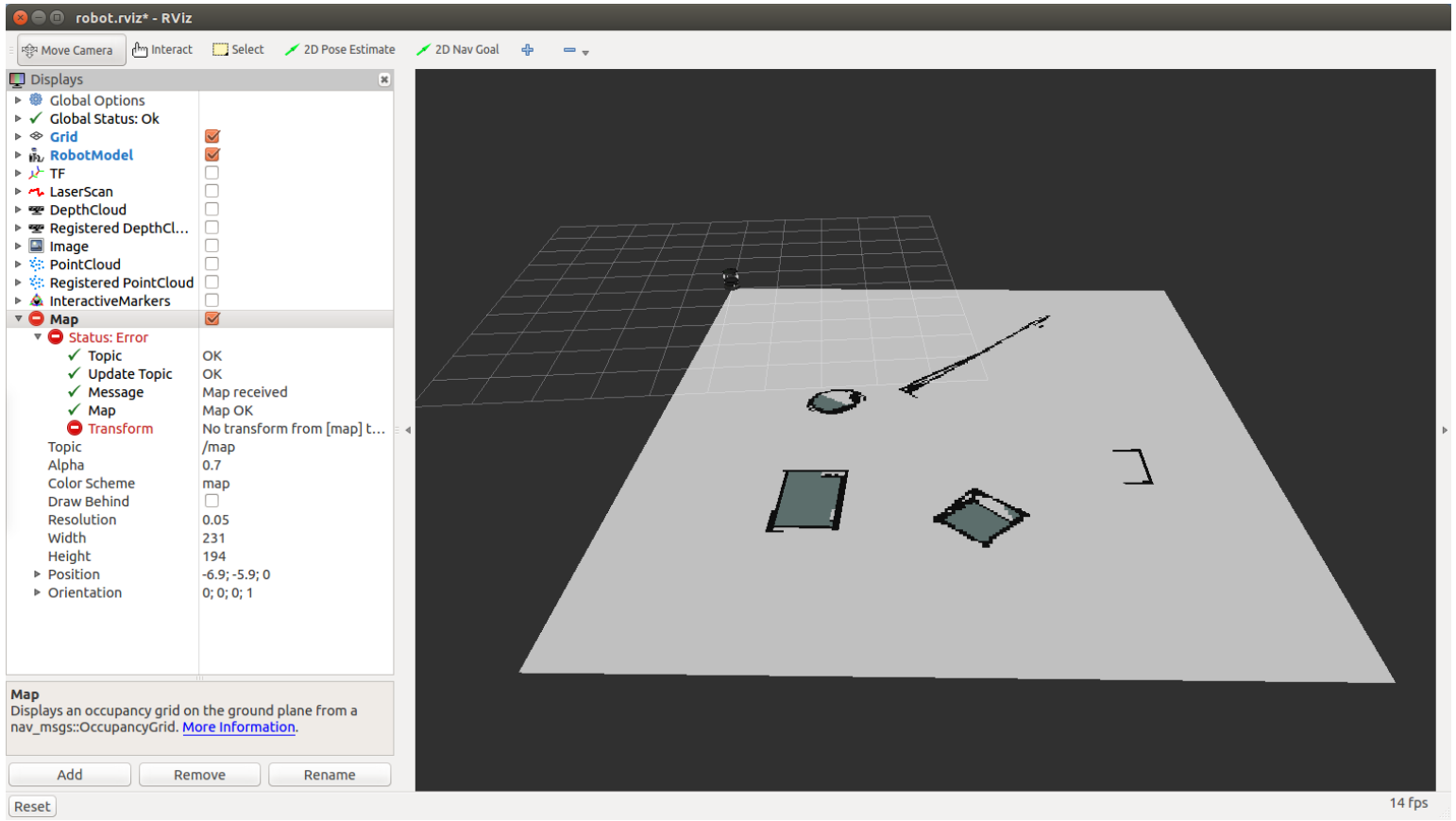
Loading an Existing Map

- Now instead of running gmapping, we will learn how to load an existing map
- The map_server node in map_server package allows you to load existing maps
  - Takes as arguments the path to the map file and the map resolution (if not specified in the YAML file)
- Let’s load Turtlbot’s playground map and display it in rviz

Loading an Existing Map

```
<launch>
  <!-- Run Gazebo with turtlebot -->
  <include file="$(find turtlebot_gazebo)/launch/turtlebot_world.launch"/>

  <!-- Load existing map -->
  <node name="map_server" pkg="map_server" type="map_server" args="$(find
turtlebot_gazebo)/maps/playground.yaml" />

  <!-- Open rviz -->
  <include file="$(find turtlebot_rviz_launchers)/launch/view_robot.launch"/>
</launch>
```

Loading an Existing Map



Loading an Existing Map

```
<launch>
  <!-- Run Gazebo with turtlebot -->
  <include file="$(find turtlebot_gazebo)/launch/turtlebot_world.launch"/>

  <!-- Load existing map -->
  <node name="map_server" pkg="map_server" type="map_server" args="$(find
turtlebot_gazebo)/maps/playground.yaml" />

  <!-- Open rviz -->
  <include file="$(find turtlebot_rviz_launchers)/launch/view_robot.launch"/>
</launch>
```

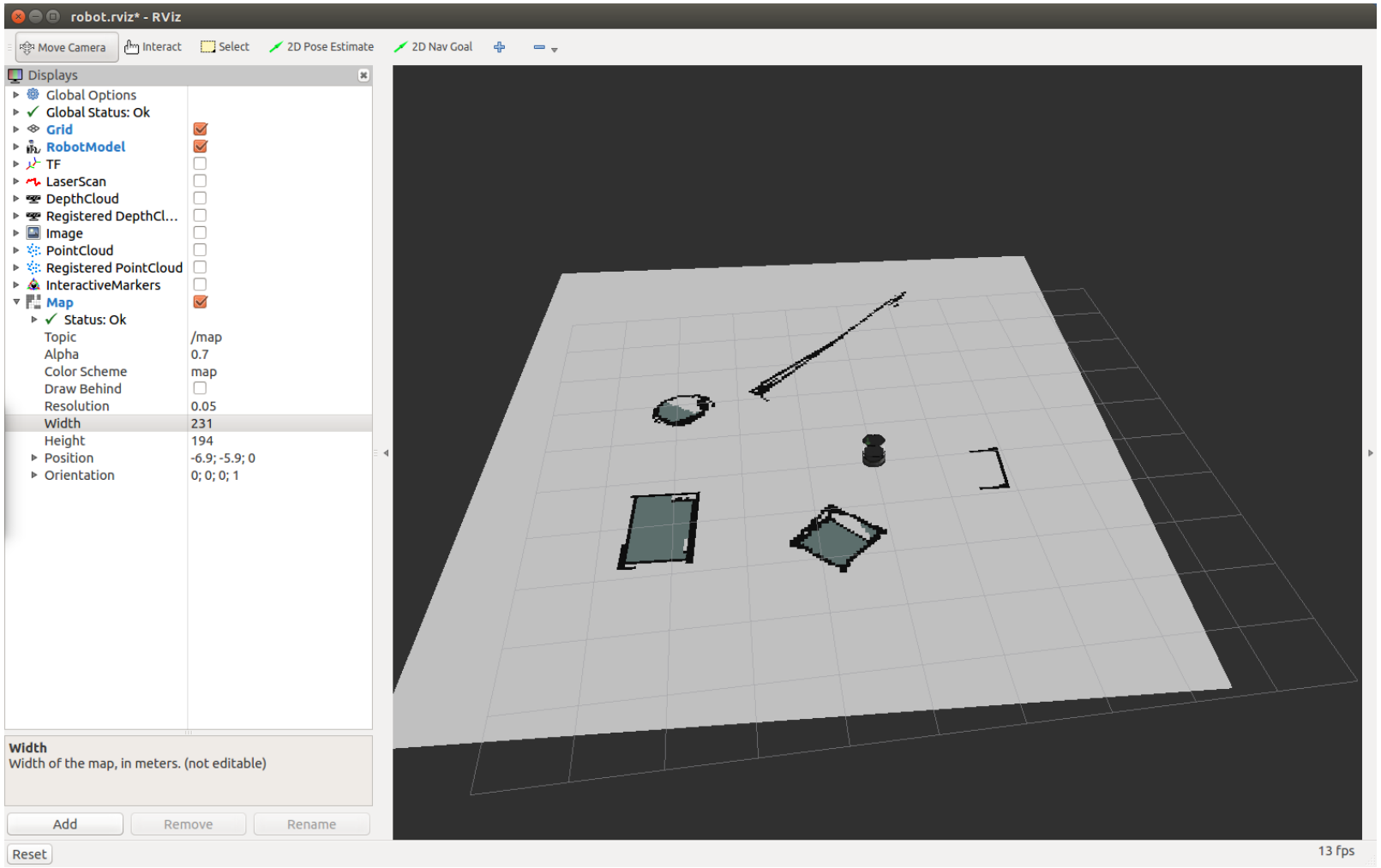
Loading an Existing Map

- Note that the robot doesn’t know its location relative to the map, that’s why rviz doesn’t show its location properly
- In the next lesson we will learn how to provide this information to the robot
- To fix this problem meanwhile we will add a static transform from /map to /odom (will be explained next lesson)

Loading an Existing Map

```
<launch>
  <!-- Run Gazebo with turtlebot -->
  <include file="$(find turtlebot_gazebo)/launch/turtlebot_world.launch"/>

  <!-- Load existing map -->
  <node name="map_server" pkg="map_server" type="map_server" args="$(find
turtlebot_gazebo)/maps/playground.yaml" />

  <!-- Publish a static transformation between /odom and /map -->
 <node name="tf" pkg="tf" type="static_transform_publisher" args="0 0 0 0 0 0 /map
/odom 100" />

  <!-- Open rviz -->
  <include file="$(find turtlebot_rviz_launchers)/launch/view_robot.launch"/>
</launch>
```

Loading an Existing Map



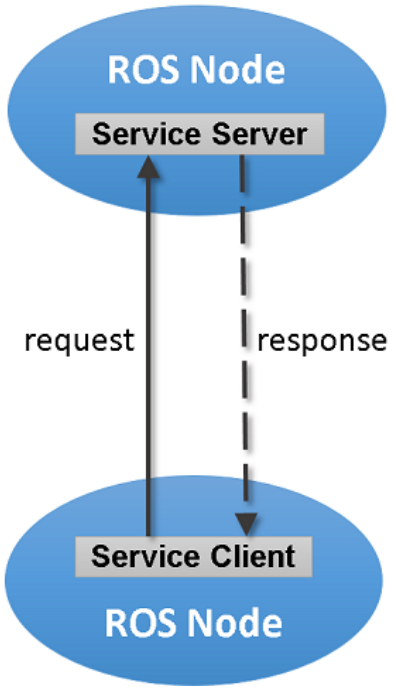
ROS Services

- The next step is to learn how to load the map into the memory in our own code
  - So we can use it to plan a path for the robot
- For that purpose we will use a ROS service called static_map provided by the map_server node

ROS Services

- Services are just synchronous remote procedure calls
  - They allow one node to call a function that executes in another node
- We define the inputs and outputs of this function similarly to the way we define new message types
- The server (which provides the service) specifies a callback to deal with the service request, and advertises the service.
- The client (which calls the service) then accesses this service through a local proxy

ROS Services



Service Name: **/example_service**
Service Type: **roscpp_tutorials/TwoInts**

Request Type: **roscpp_tutorials/TwoIntsRequest**
Response Type: **roscpp_tutorials/TwoIntsResponse**

Using a Service

- To call a service programmatically:

```
ros::NodeHandle nh;
ros::ServiceClient client = nh.serviceClient<my_package::Foo>("my_service_name");
my_package::Foo foo;
foo.request.<var> = <value>;
...
if (client.call(foo)) {
    ...
}
```

  - Service calls are blocking
  - If the service call succeeded, call() will return true and the value in srv.response will be valid
  - If the call did not succeed, call() will return false and the value in srv.response will be invalid

static_map Service

- To get the OGM in a ROS node you can call the service static_map
- This service gets no arguments and returns a message of type nav_msgs/OccupancyGrid
- The message consists of two main structures:
  - MapMetaData – metdata of the map, contains:
    - resolution – map resolution in m/cell
    - width – number of cells in the y axis
    - height – number of cells in the x axis
  - int8[] data – the map's data

Loading a Map in C++ (1)

```
#include <ros/ros.h>
#include <nav_msgs/GetMap.h>
#include <vector>

using namespace std;

// grid map
int rows;
int cols;
double mapResolution;
vector<vector<bool> > grid;

bool requestMap(ros::NodeHandle &nh);
void readMap(const nav_msgs::OccupancyGrid& msg);
void printGridToFile();
```

Loading a Map in C++ (2)

```
int main(int argc, char** argv)
{
    ros::init(argc, argv, "mapping");
    ros::NodeHandle nh;

    if (!requestMap(nh))
        exit(-1);

    printGridToFile();

    return 0;
}
```

Loading a Map in C++ (3)

```
bool requestMap(ros::NodeHandle &nh)
{
    nav_msgs::GetMap::Request req;
    nav_msgs::GetMap::Response res;

    while (!ros::service::waitForService("static_map", ros::Duration(3.0))) {
         ROS_INFO("Waiting for service static_map to become available");
    }

    ROS_INFO("Requesting the map...");
    ros::ServiceClient mapClient =
nh.serviceClient<nav_msgs::GetMap>("static_map");

    if (mapClient.call(req, res)) {
        readMap(res.map);
        return true;
    }
    else {
        ROS_ERROR("Failed to call map service");
        return false;
    }
}
```

Loading a Map in C++ (4)

```
void readMap(const nav_msgs::OccupancyGrid& map)
{
    ROS_INFO("Received a %d X %d map @ %.3f m/px\n",
            map.info.width,
            map.info.height,
            map.info.resolution);
    rows = map.info.height;
    cols = map.info.width;
    mapResolution = map.info.resolution;

    // Dynamically resize the grid
    grid.resize(rows);
    for (int i = 0; i < rows; i++) {
        grid[i].resize(cols);
    }
    int currCell = 0;
    for (int i = 0; i < rows; i++)  {
        for (int j = 0; j < cols; j++)      {
            if (map.data[currCell] == 0) // unoccupied cell
                grid[i][j] = false;
            else
                grid[i][j] = true; // occupied (100) or unknown cell (-1)
            currCell++;
        }
    }
}
```

Loading a Map in C++ (5)

```
void printGridToFile() {
    ofstream gridFile;
    gridFile.open("grid.txt");

    for (int i = grid.size() - 1; i >= 0; i--) {
        for (int j = 0; j < grid[0].size(); j++) {
        gridFile << (grid[i][j] ? "1" : "0");
        }
        gridFile << endl;
    }
    gridFile.close();
}
```

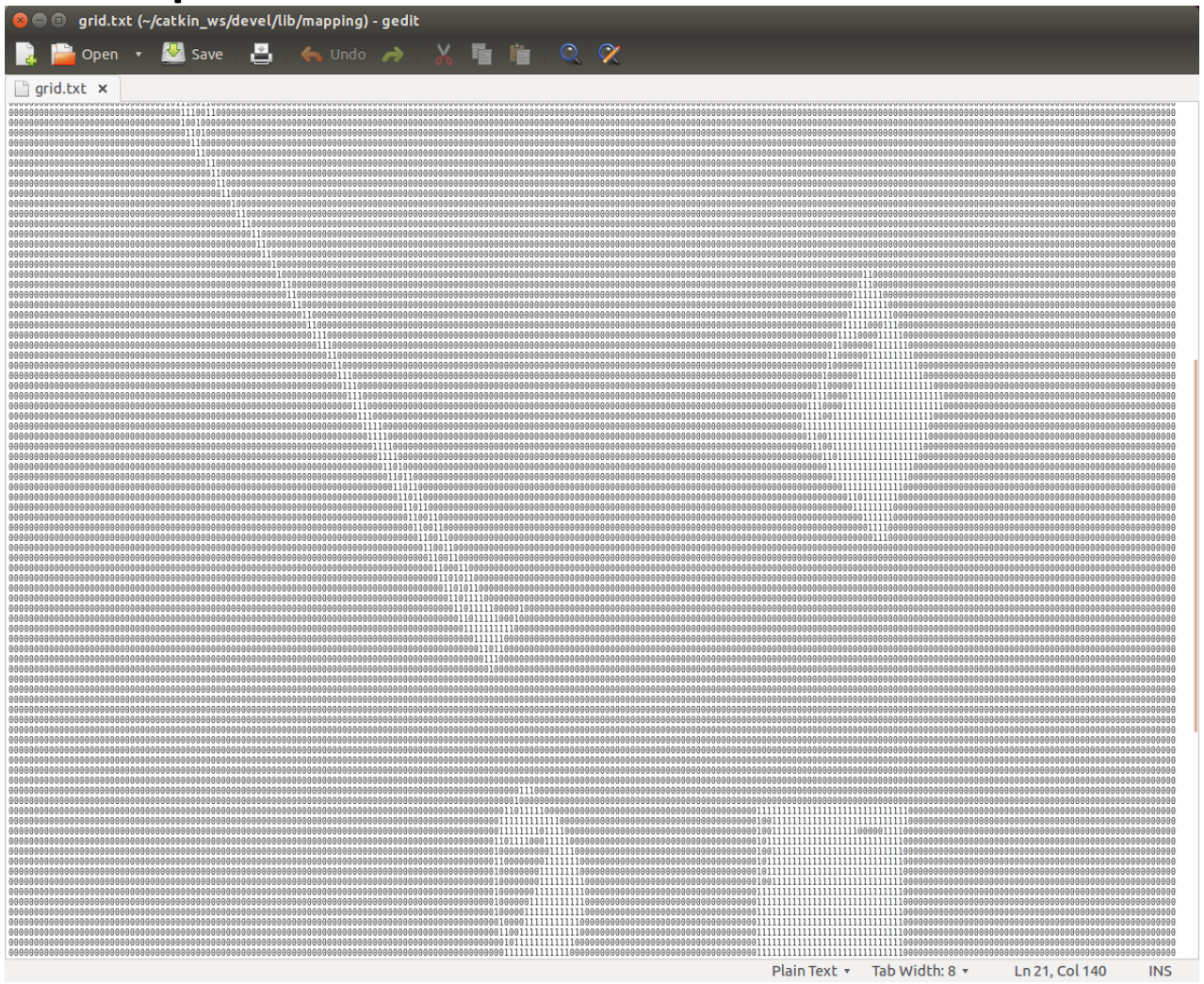
Launch File

```
<launch>
  <!-- Run Gazebo with turtlebot -->
  <include file="$(find turtlebot_gazebo)/launch/turtlebot_world.launch"/>

  <!-- Load existing map -->
  <node name="map_server" pkg="map_server" type="map_server" args="$(find
turtlebot_gazebo)/maps/playground.yaml" />

  <!-- Publish a static transformation between /odom and /map -->
 <node name="tf" pkg="tf" type="static_transform_publisher" args="0 0 0 0 0 0 /map
/odom 100" />

  <!-- Open rviz -->
  <include file="$(find turtlebot_rviz_launchers)/launch/view_robot.launch"/>

  <!-- Run load_map node -->
  <node name="load_map" pkg="mapping" type="load_map" output="screen"
cwd="node" />
</launch>
```

Output File

- If you want to write the output file to the directory your node is running from, you can use the “cwd” attribute in the <node> tag
  - The default behavior is to use $ROS_HOME directory.
- In our example, grid.txt will be written to ~/catkin_ws/devel/lib/mapping

Loading a Map



Ex. 5

- Load playground.pgm map from turtlebot_gazebo/maps into memory
- Inflate the obstacles in the map, so that the robot will not get too close to the obstacles
- Save the inflated map into a new file by publishing the new map to /map topic and using map_saver