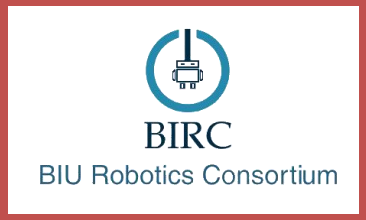
October 2016



ROS – Lecture 5

Mapping in ROS
rviz
ROS Services

Lecturer: Roi Yehoshua
roiyeho@gmail.com

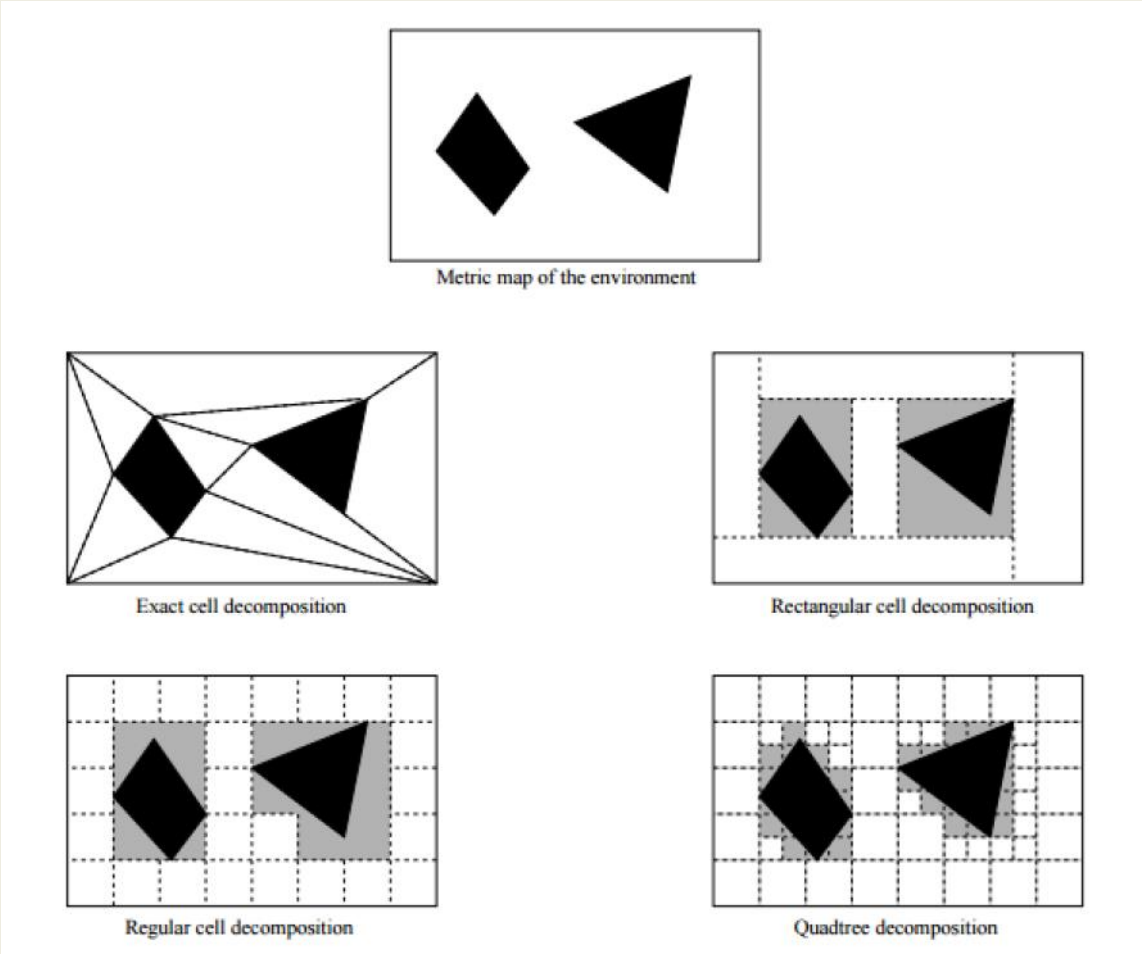
Why Mapping?

- 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

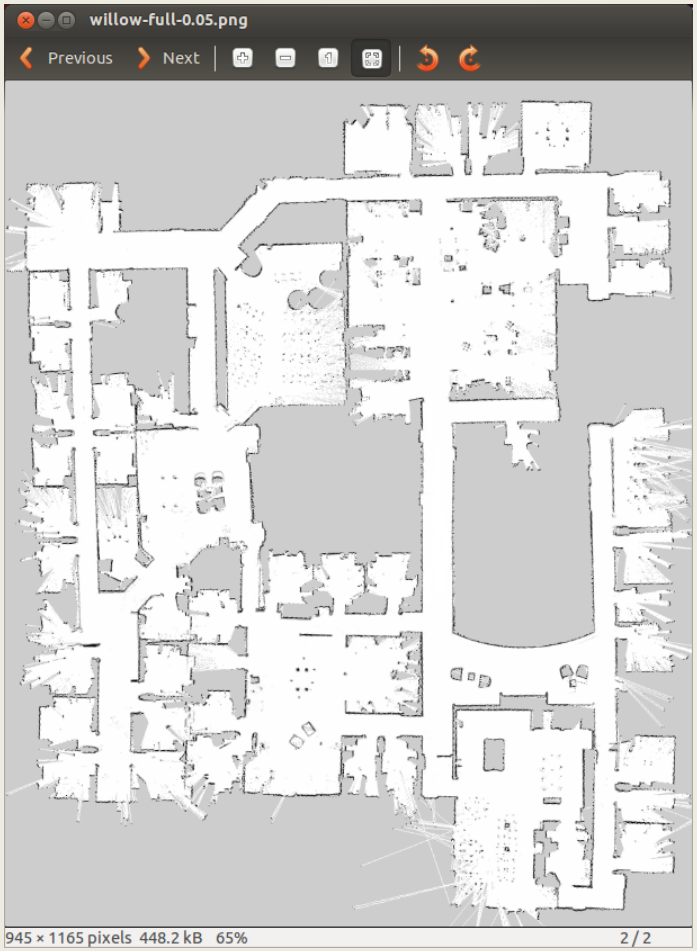
Cellular Decomposition



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)

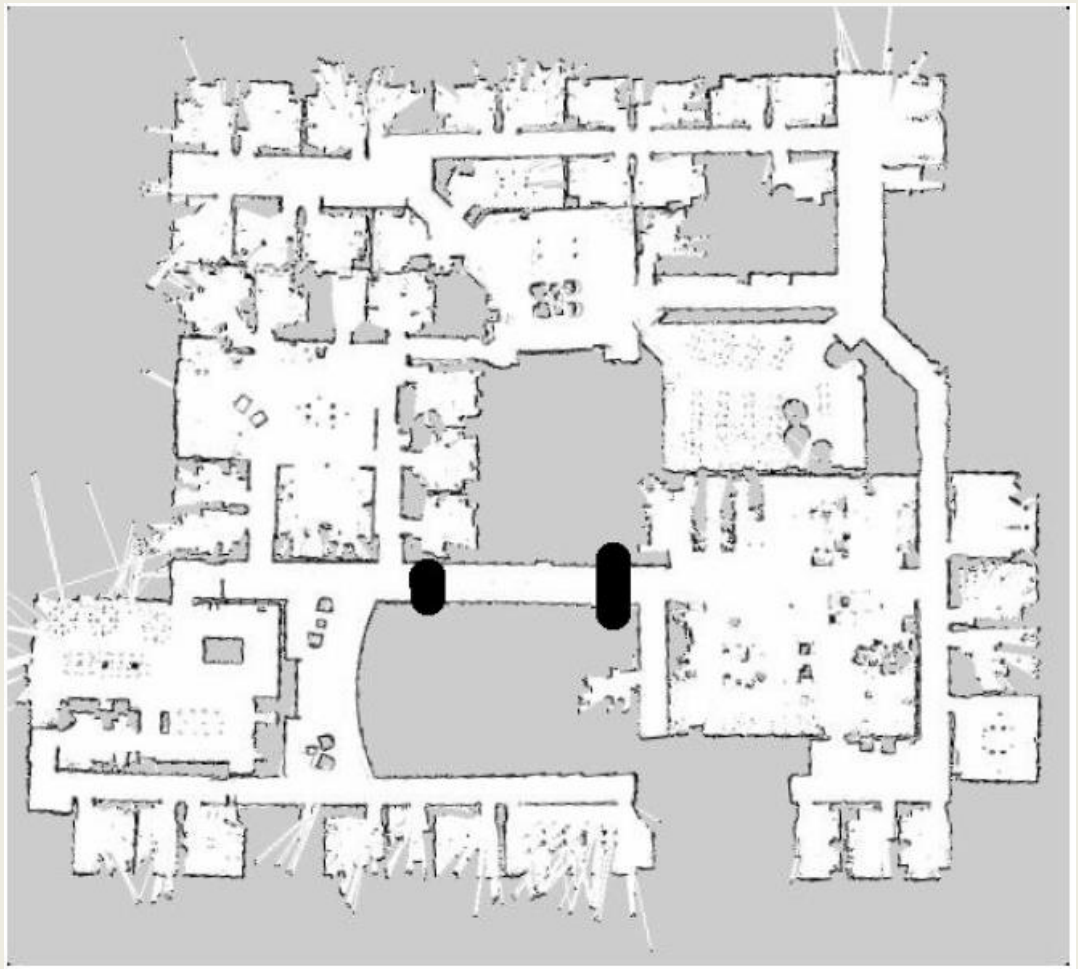
occupied_thresh: Pixels with occupancy probability greater than this threshold are considered completely occupied

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



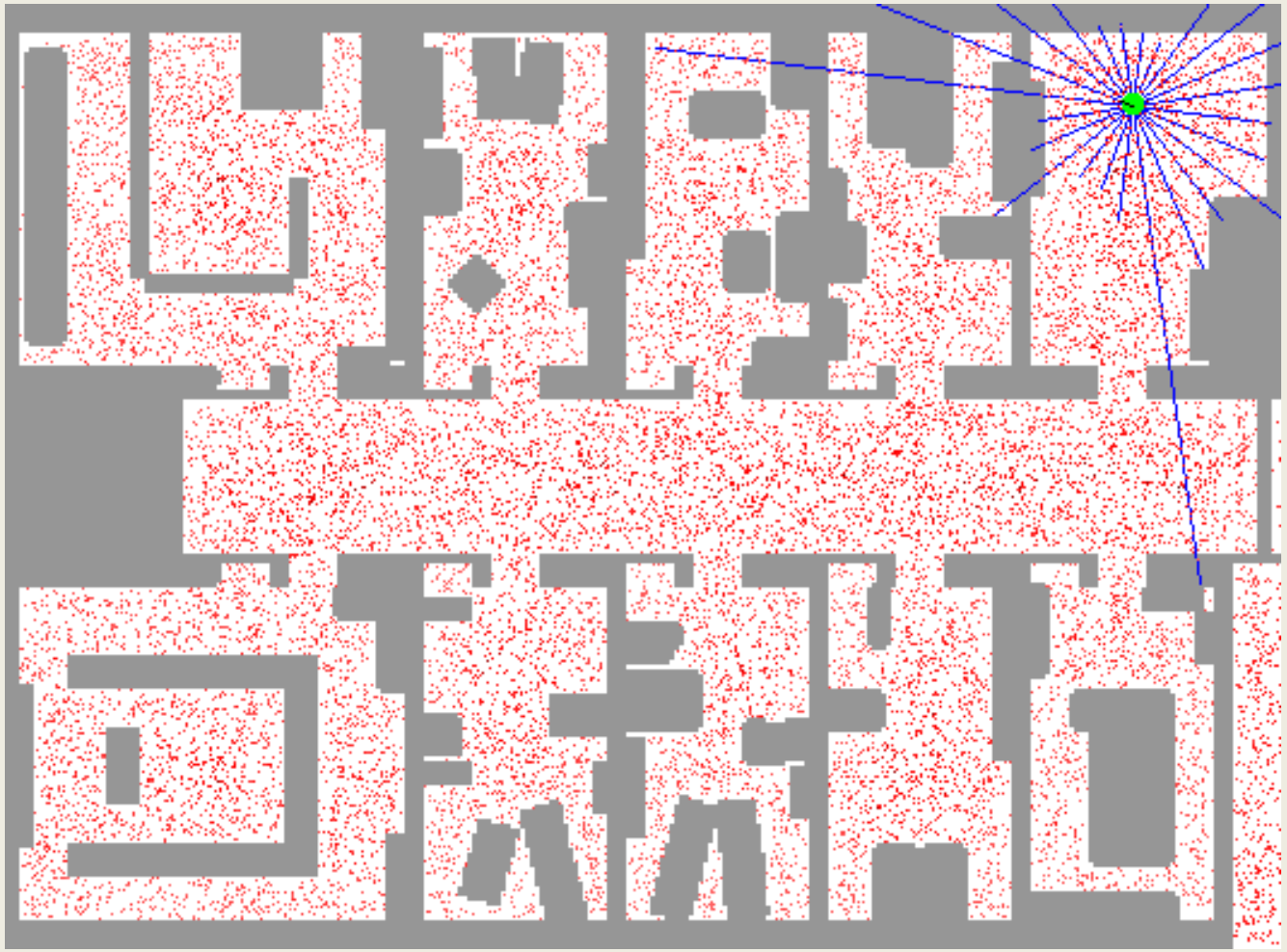
SLAM

- **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:

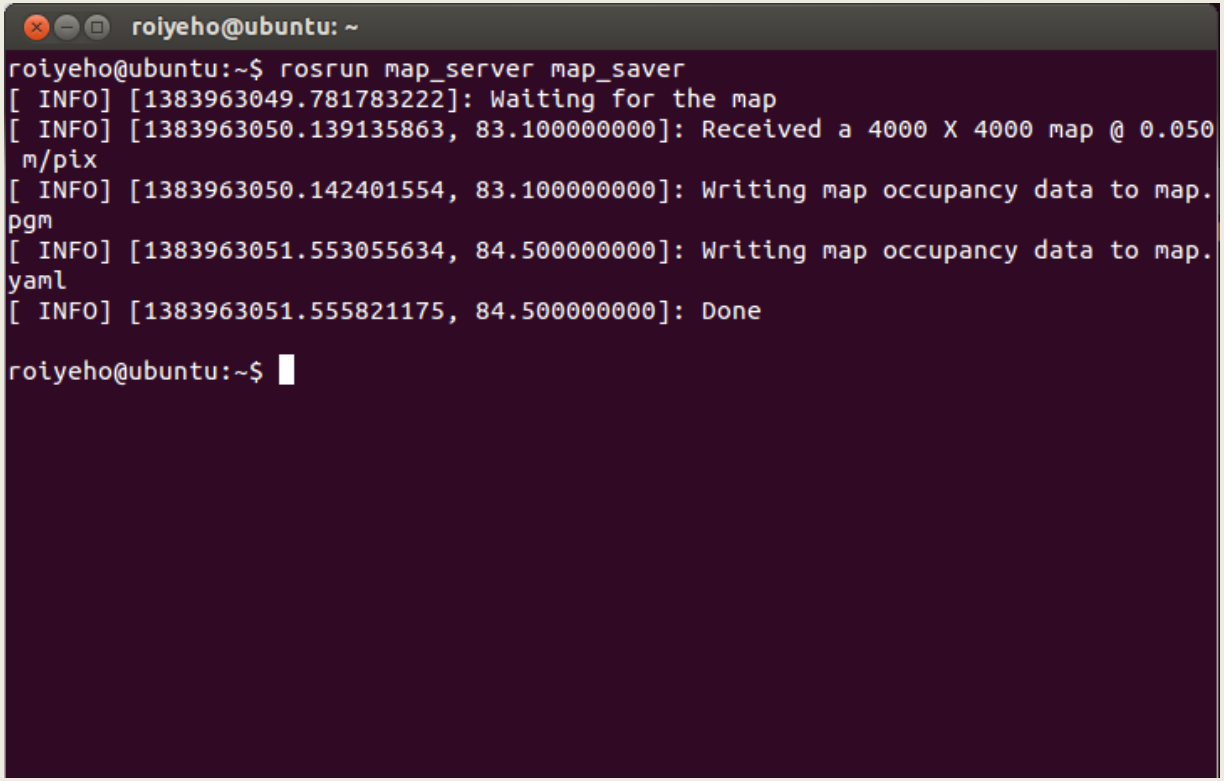
```
$ sudo apt-get install ros-indigo-map-server
```

- 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

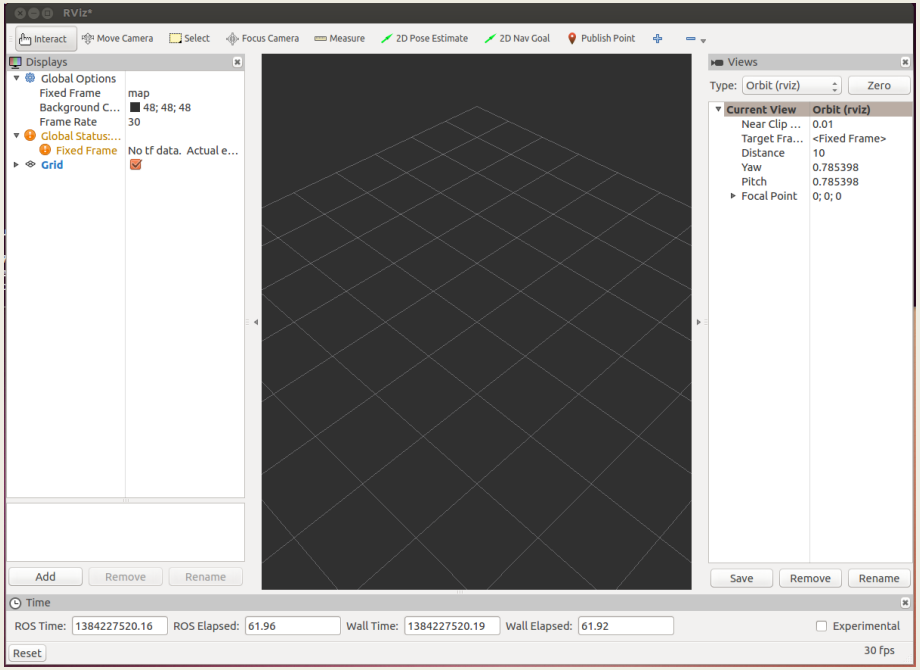
- You can open the pgm file with the default Ubuntu image viewer program (eog)

```
$ 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

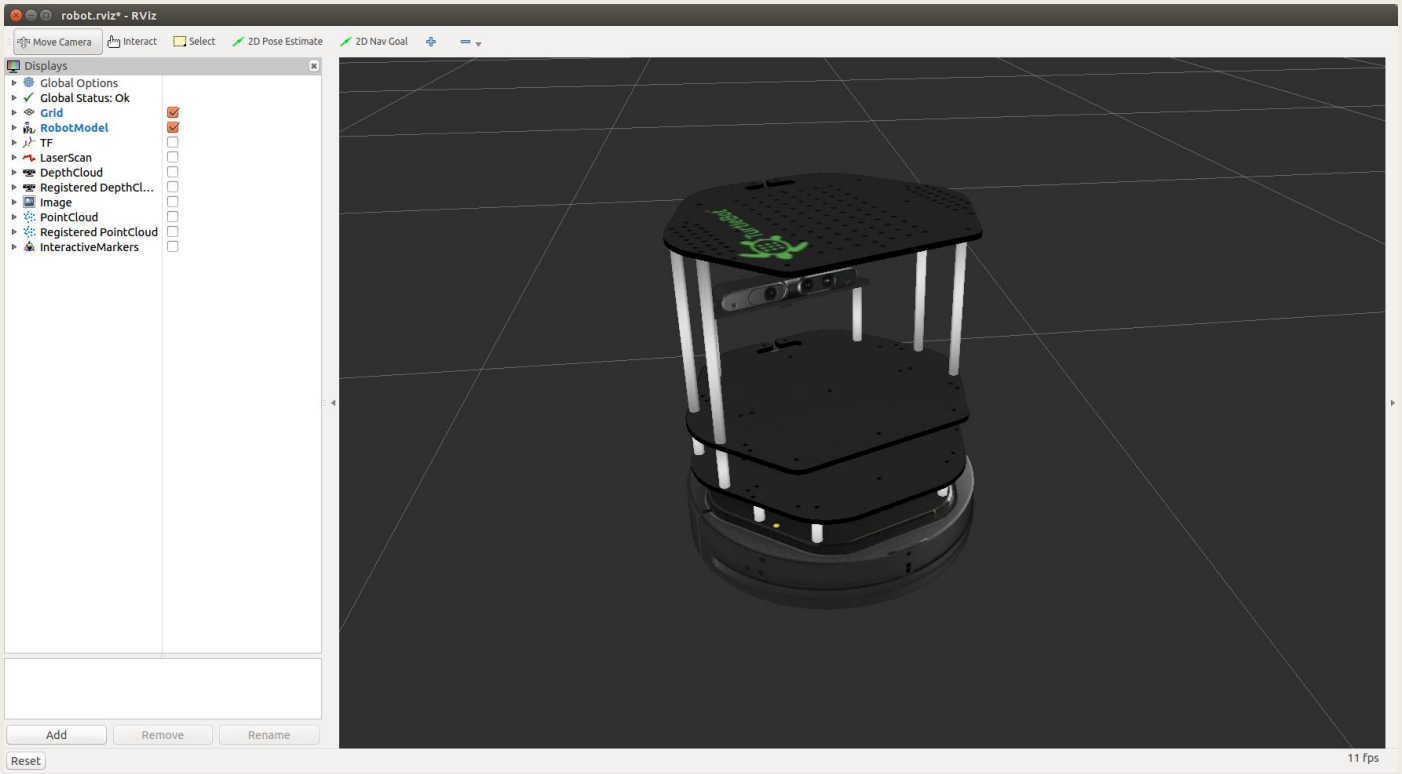
rviz Displays

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

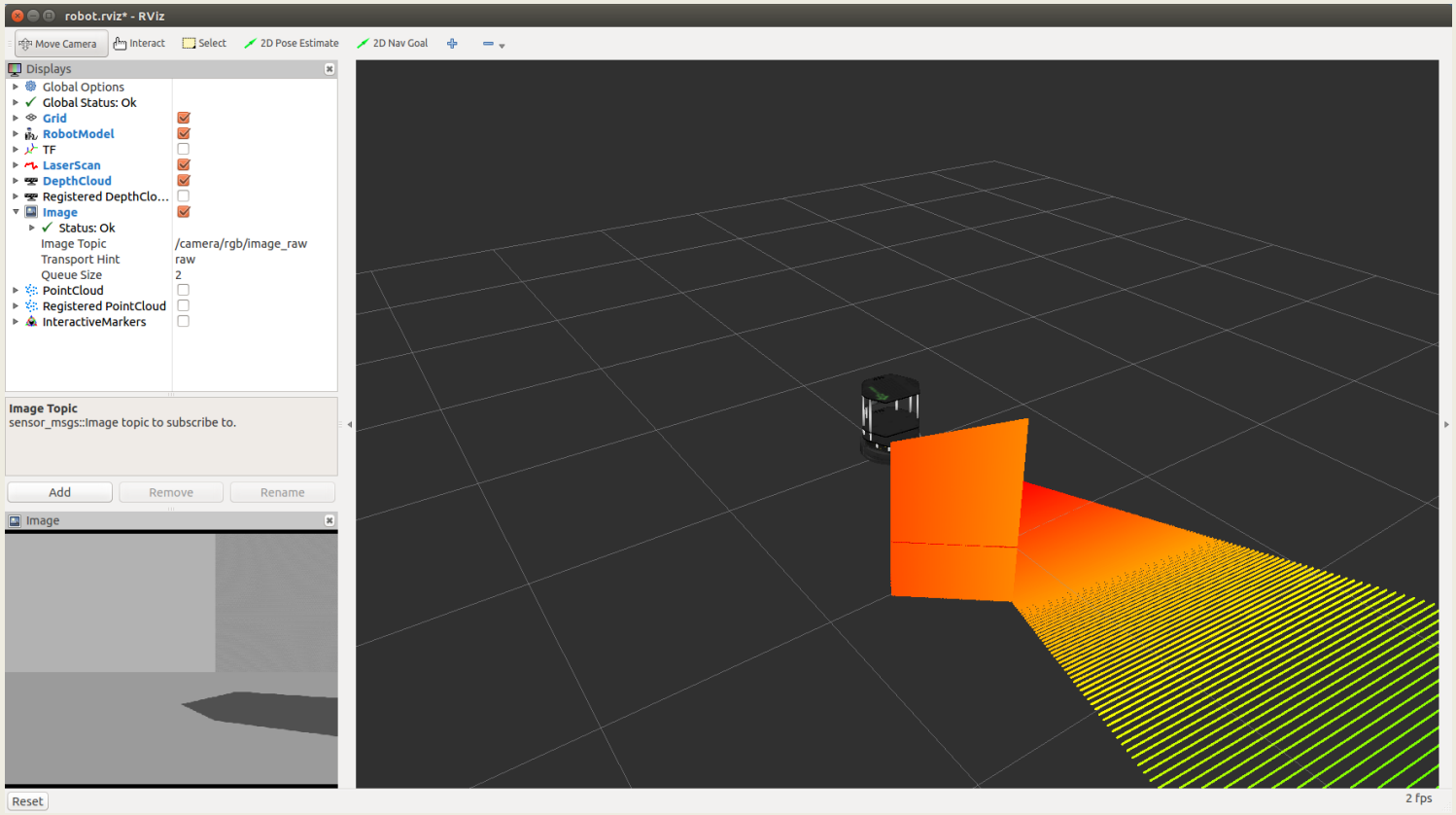
- You can start rviz already configured to visualize the robot and its sensor's output:

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



TurtleBot Image Display

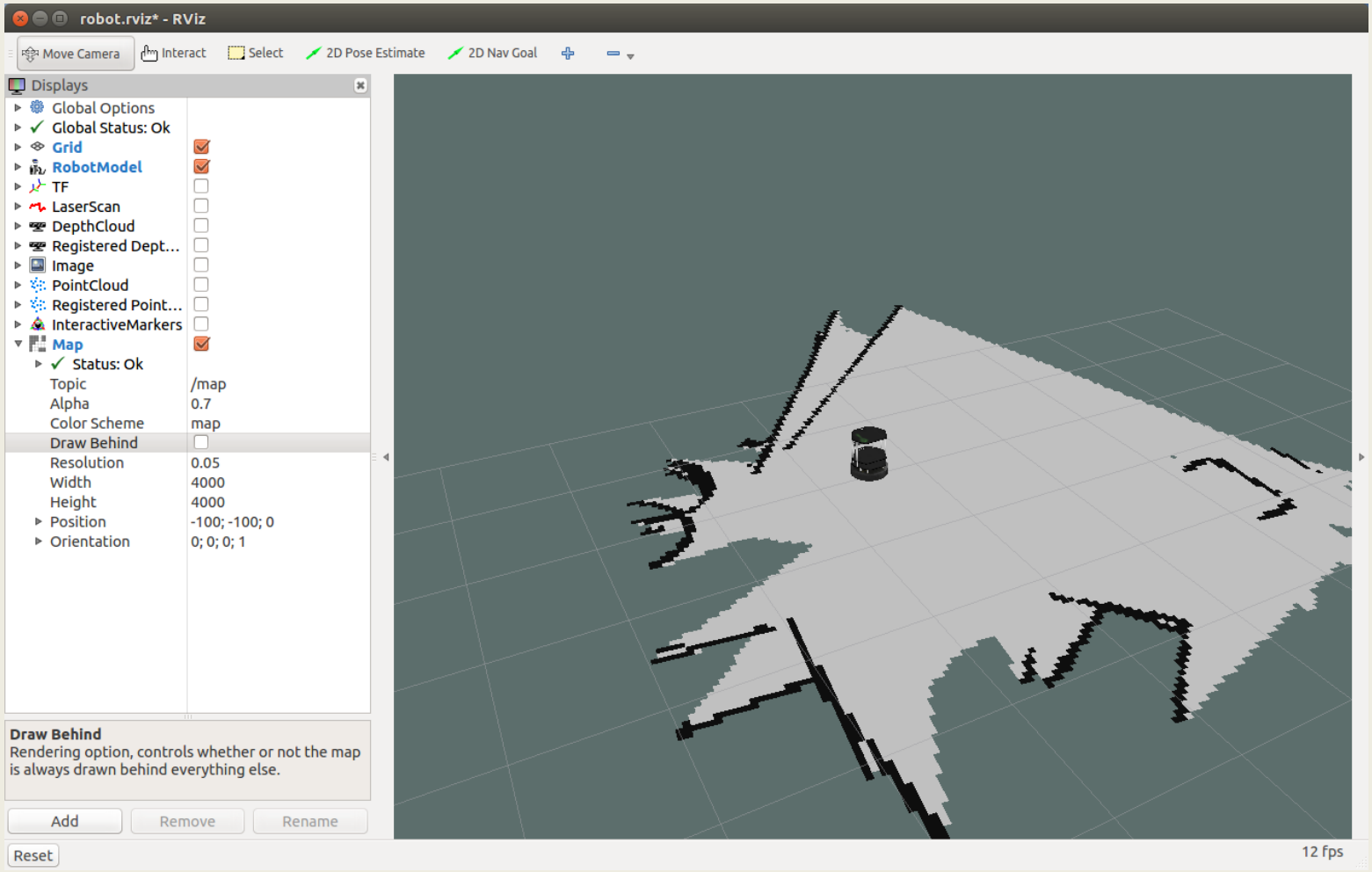
- To visualize any display you want, just click on its check button



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



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)

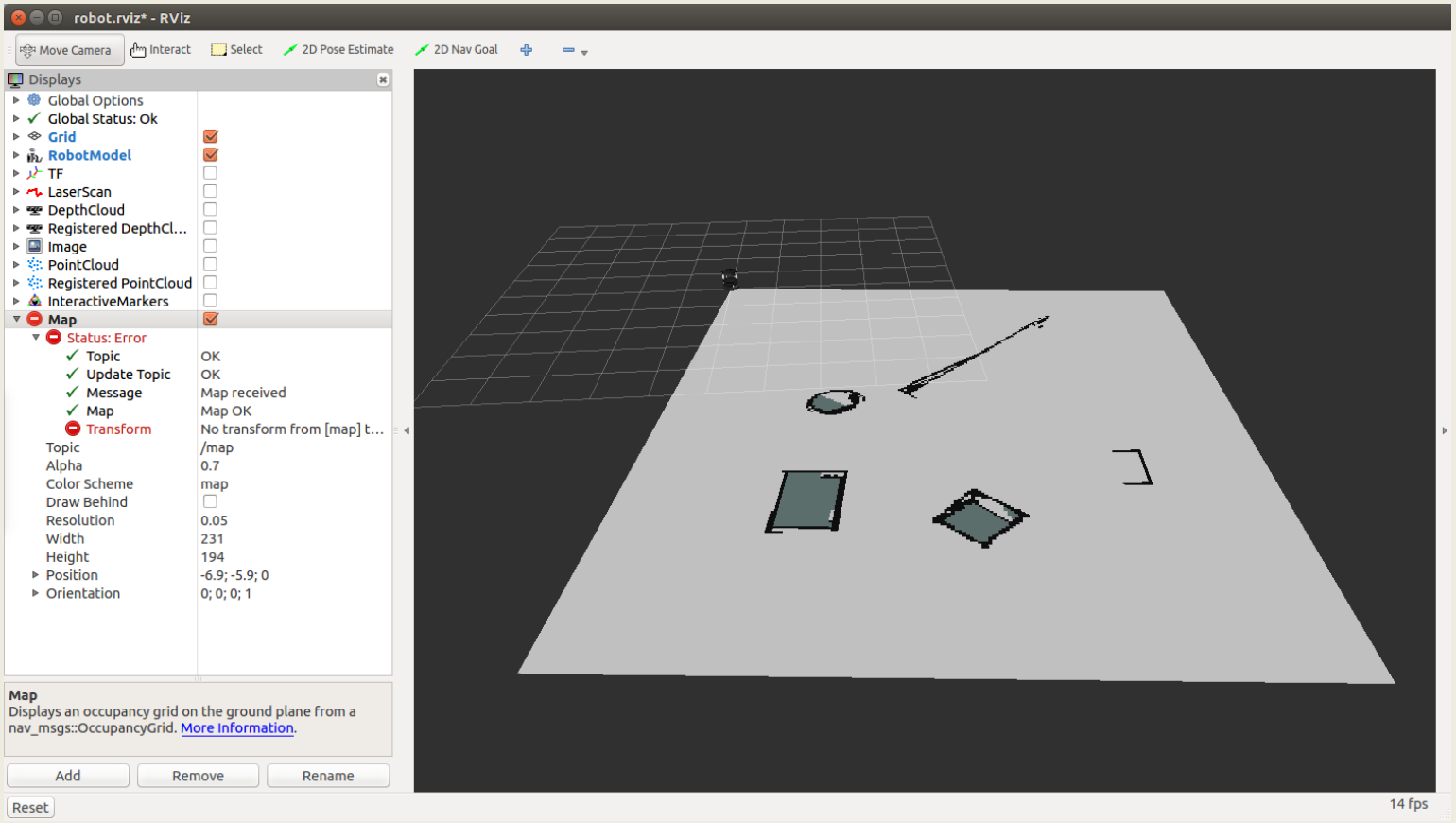
Loading an Existing Map

- 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>
```

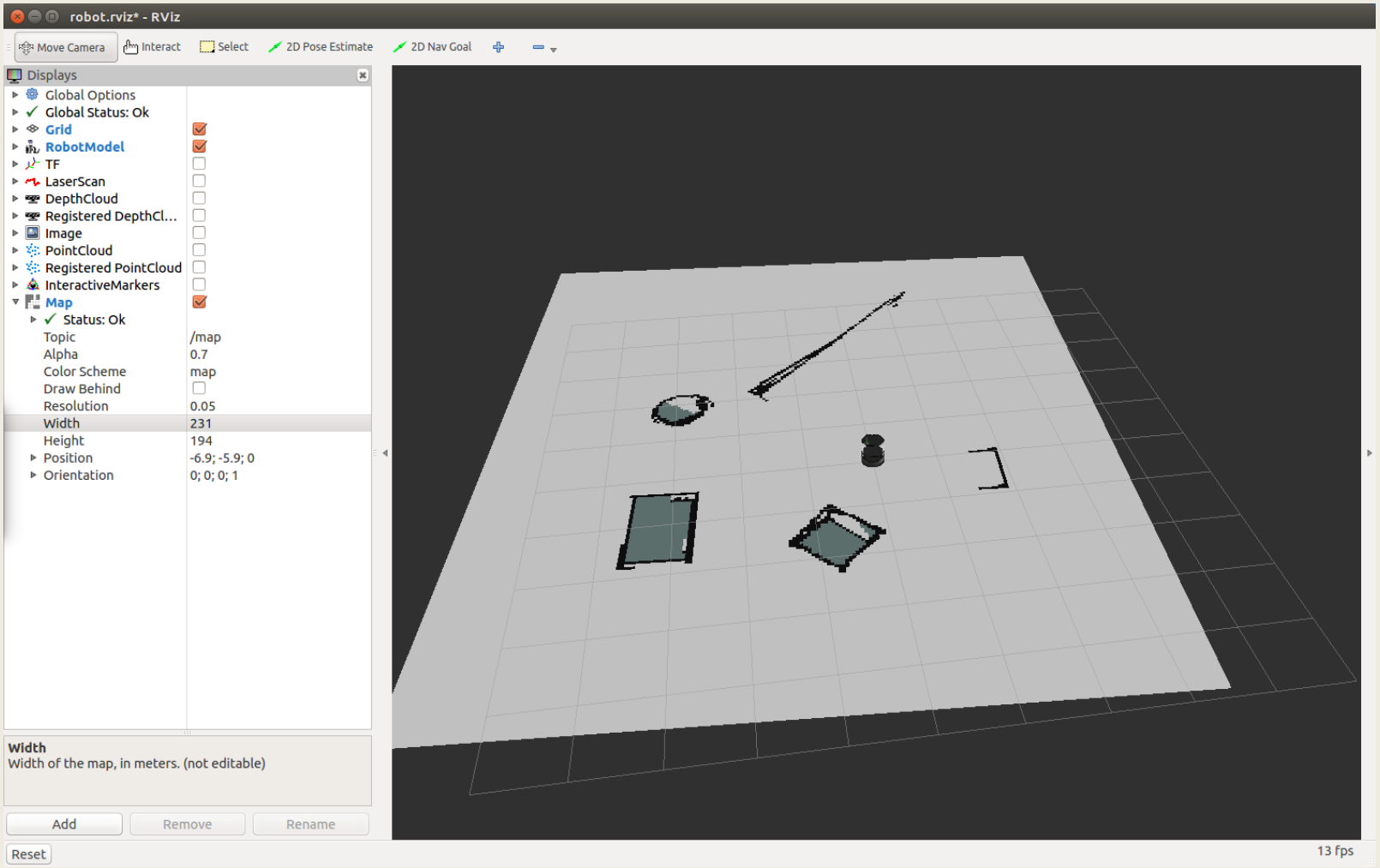
Loading an Existing Map

- 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



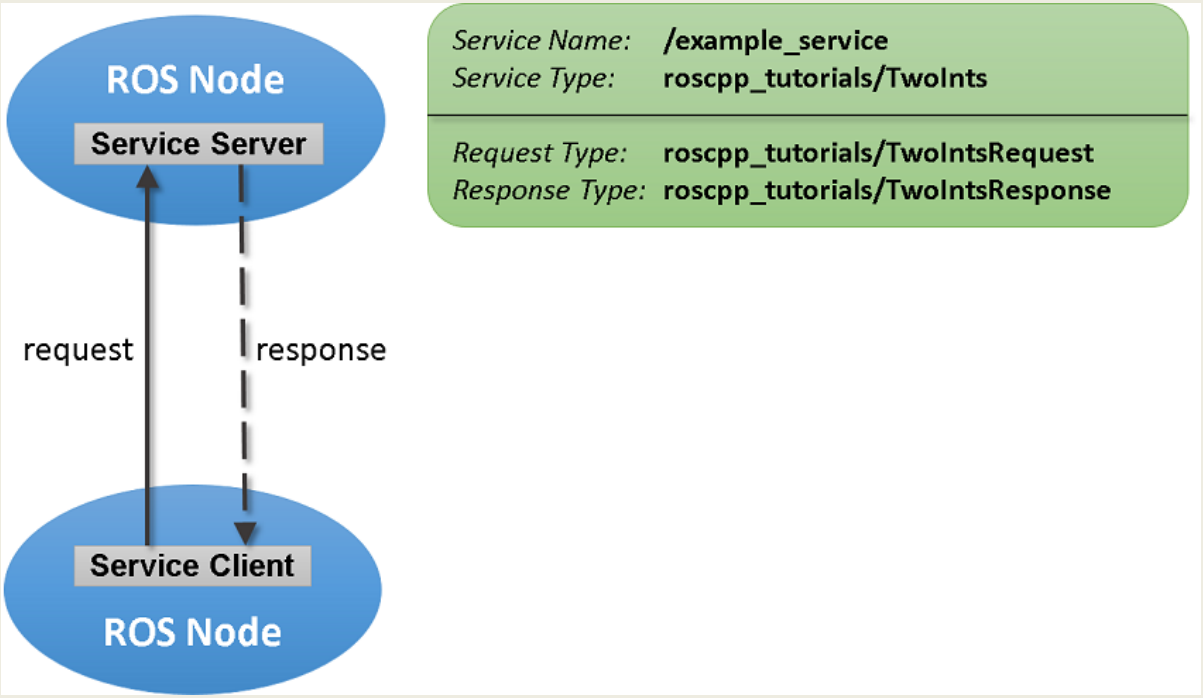
ROS Services

- 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



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

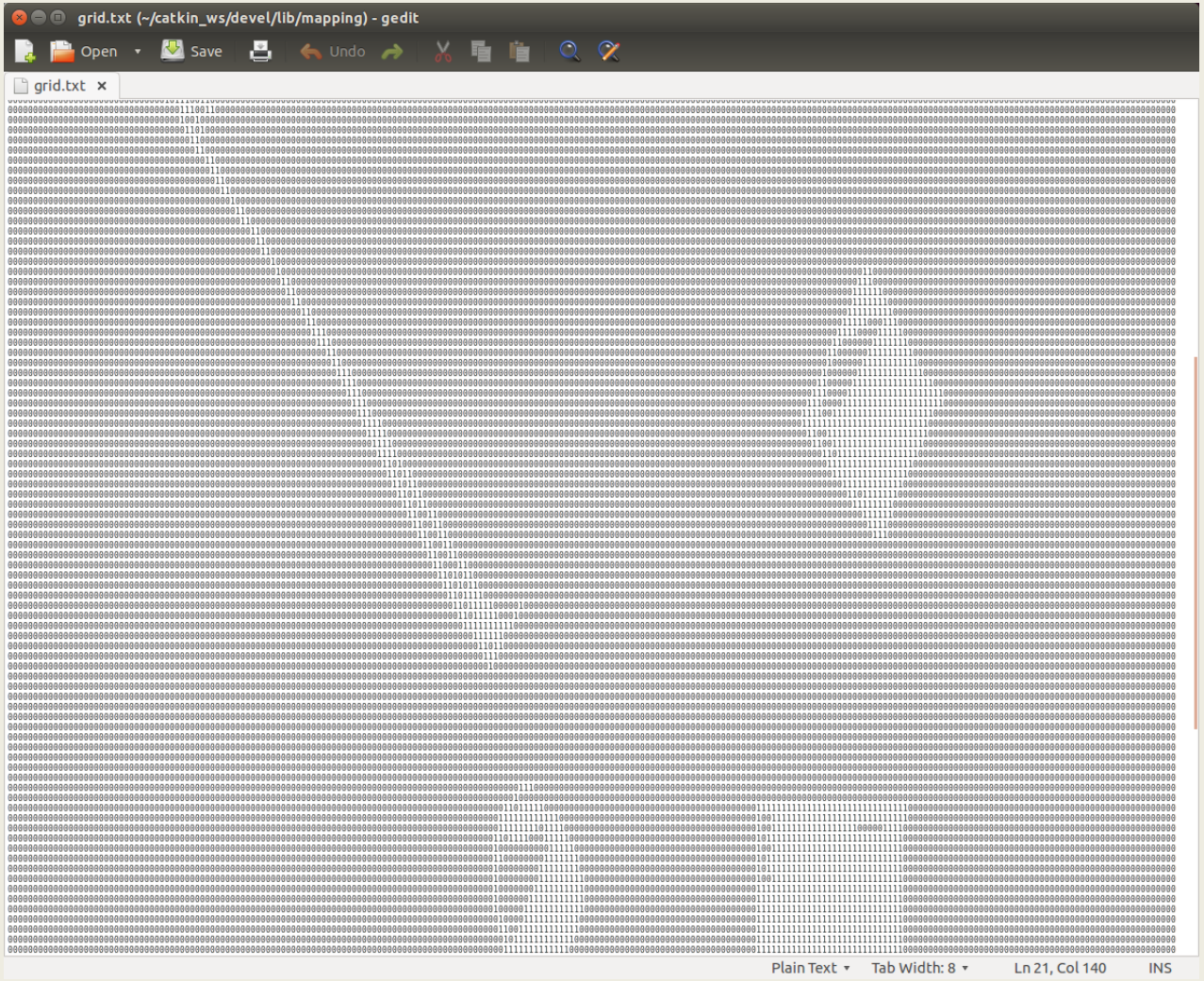
Launch File

```
<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