Experimental Robotics Laboratory – Sensors in Gazebo and URDF

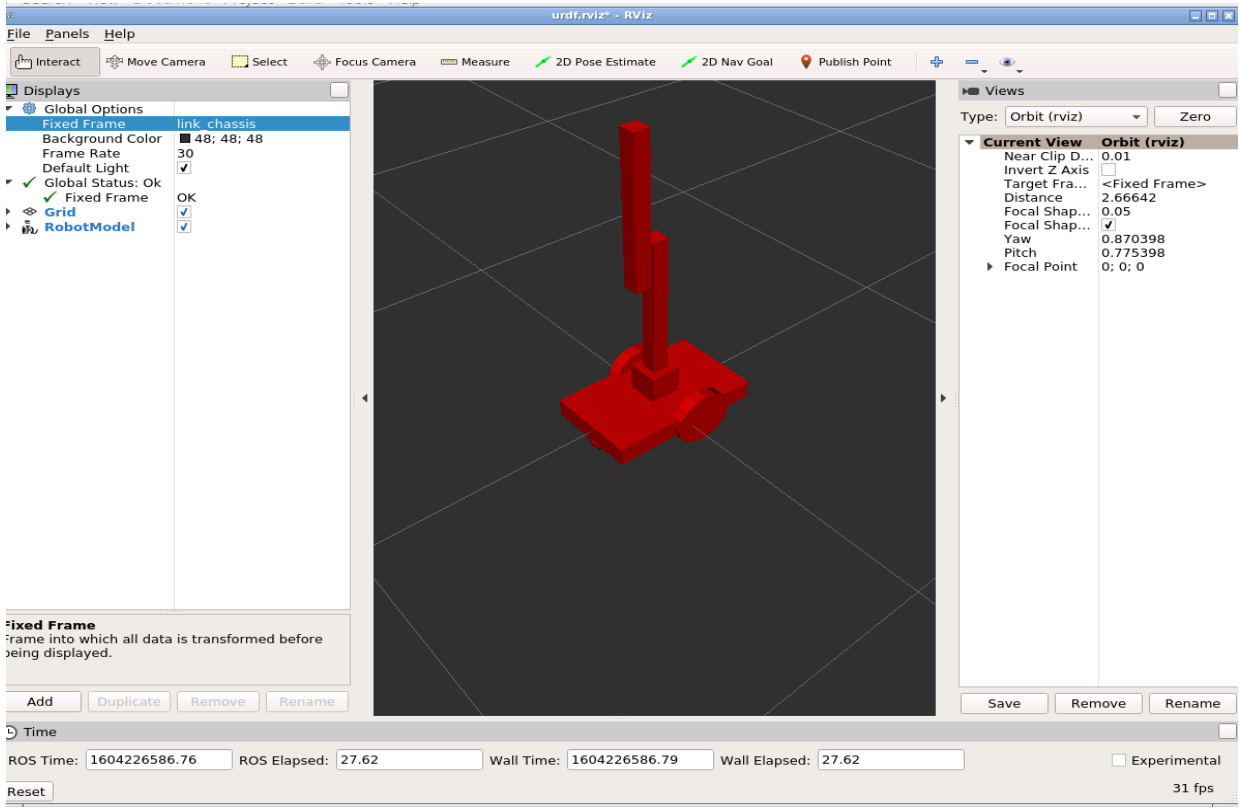
Carmine Tommaso Recchiuto

Exercise 1

Starting from the robot build so far, add the following links/joints:

- a rotating base (i.e., a continuous joint rotating along the z axis)
- a link connected to the base with a revolute joint, rotating along the x axis (i.e., pitch)
- an additional link, connected to the previous one with a revolute joint (again along the x axis)

Visualize the robot with Rviz

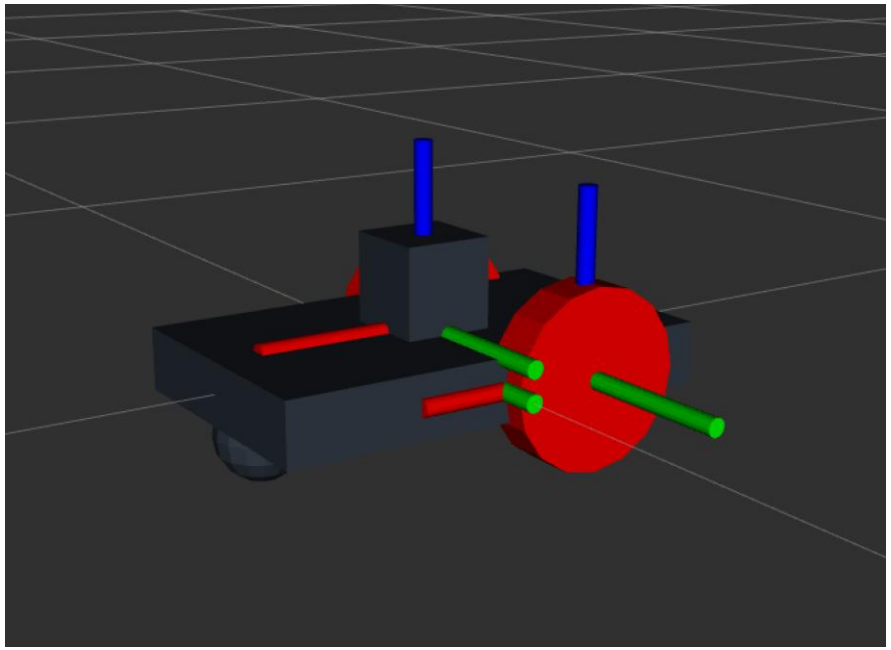


Exercise 1

The folder with all we have done so far can be cloned here: https://github.com/CarmineD8/erl1

```
<link name="link_cube">
  <visual name="visual_cube">
    <origin xyz="0 0 0.05" rpy="0 0 0"/>
    <geometry>
      <box size="0.1 0.1 0.1"/>
    </geometry>
    <material name="blue"/>
  </visual>
  <collision>
  <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="0.1 0.1 0.1"/>
    </geometry>
  </collision>
 <inertial>
  <mass value="0.3"/>
  <origin xyz="0 0 0" rpy="0 0 0"/>
  <inertia
      ixx="0.000125" ixy="0" ixz="0"
      iyy="0.0010625" iyz="0"
      izz="0.0010625"/>
</inertial>
</link>
```

```
<joint name="cube_joint" type="revolute">
  <parent link="link_chassis"/>
  <child link="link_cube"/>
  <origin xyz="0 0 0.035" rpy="0 0 0"/>
  <axis xyz="0 0 1"/>
  <limit lower="-1.57" upper="1.57" effort="5"
velocity="1.0"/>
</joint>
```



Exercise 1

```
<link name="link_arm">
  <visual>
    <origin xyz="0.5 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="1.0 0.05 0.05"/>
    </geometry>
    <material name="blue"/>
  </visual>
  <collision>
    <geometry>
      <box size="1.0 0.05 0.05"/>
    </geometry>
  </collision>
  <inertial>
  <mass value="1.0"/>
  <origin xyz="0 0 0" rpy="0 0 0"/>
  <inertia
      ixx="0.0004167" ixy="0" ixz="0"
      iyy="0.0835417" iyz="0"
      izz="0.0835417"/>
</inertial>
</link>
```

```
<joint name="arm_joint" type="revolute">
  <parent link="link_cube"/>
  <child link="link_arm"/>
  <origin xyz="0 0 0.1" rpy="0 0 0"/>
  <axis xyz="0 1 0"/>
  <limit lower="-3.0" upper="-0.1" effort="5"
velocity="1.0"/>
</joint>
```

Gazebo Plugins

More details here:
https://github.com/gazebosim/ros_gz/tree/ros2/ros_gz_bridge

To use Gazebo plugins within ROS we have to further update the launch file, by also starting the gz_bridge node!

```
# Node to bridge messages like /cmd_vel and /odom
    gz_bridge_node = Node(
        package="ros_gz_bridge",
        executable="parameter_bridge",
        arguments=[

"/clock@rosgraph_msgs/msg/Clock[gz.msgs.Clock",

"/cmd_vel@geometry_msgs/msg/Twist@gz.msgs.Twist",

"/odom@nav_msgs/msg/Odometry@gz.msgs.Odometry",

"/joint_states@sensor_msgs/msg/JointState@gz.msgs.M
odel",

"/tf@tf2_msgs/msg/TFMessage@gz.msgs.Pose_V"
        ],
        output="screen",
        parameters=[
            {'use_sim_time': True},
        ]
    )
```

We forward the following topics:

•/clock: The topic used for tracking simulation time or any custom time source.

•/cmd_vel: We'll control the simulated robot from this ROS topic.

•/odom: Gazebo's diff drive plugin provides this odometry topic for ROS consumers.

•/joint_states: Gazebo's other plugin provides the dynamic transformation of the wheel joints.

•/tf: Gazebo provides the real-time computation of the robot's pose and the positions of its links, sensors, etc.

remove the joint_state_publisher from the spawn_robot.launch.py, and add gz_bridge_node!

Gazebo Plugins

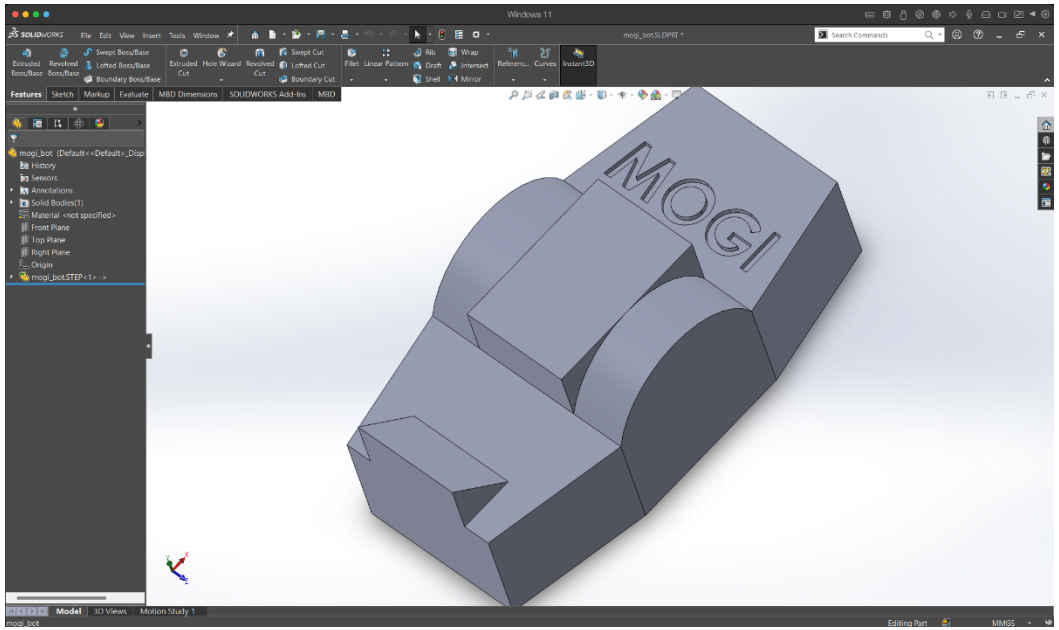
The friction between the wheels and the ground plane can be unrealistic so we can adjust it inside our URDF, let's add the following physical simulation parameters to the end of our .gazebo file before the </robot> tag is closed:

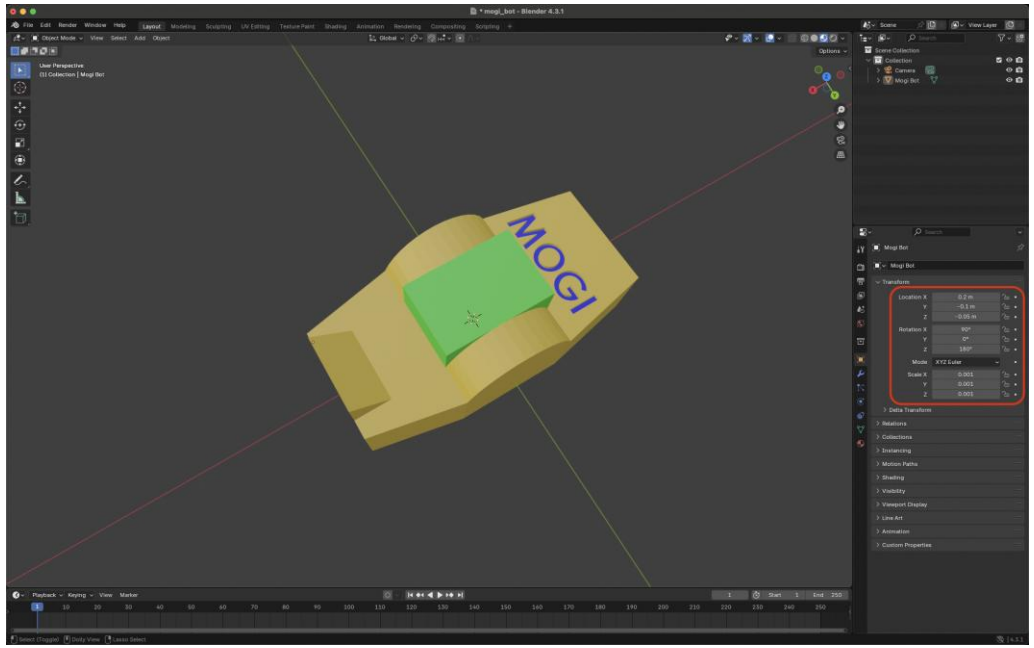
```
<gazebo reference="left_wheel">
    <mu1>0.5</mu1>
    <mu2>0.5</mu2>
    <kp>1000000.0</kp>
    <kd>100.0</kd>
    <minDepth>0.0001</minDepth>
    <maxVel>1.0</maxVel>
  </gazebo>

  <gazebo reference="right_wheel">
    <mu1>0.5</mu1>
    <mu2>0.5</mu2>
    <kp>1000000.0</kp>
    <kd>100.0</kd>
    <minDepth>0.0001</minDepth>
    <maxVel>1.0</maxVel>
  </gazebo>

  <gazebo reference="link_chassis">
    <mu1>0.000002</mu1>
    <mu2>0.000002</mu2>
  </gazebo>
```

3D Models and Textures

The robot can be made visually more appealing with some 3D models.

We can either use .stl files or .dae collada meshes. With collada meshes you can individually color certain areas of meshes (e.g. the tyre, the hub and spokes in case of the wheel). With .stl files we can only assign a single color for the model.





taken from: https://github.com/MOGI-ROS/Week-3-4-Gazebo-basics

3D Models and Textures

You can clone meshes from: https://github.com/CarmineD8/meshes

Put the folder in the ROS2 package. Of course we need to modify the urdf, and the CMakeLists.txt

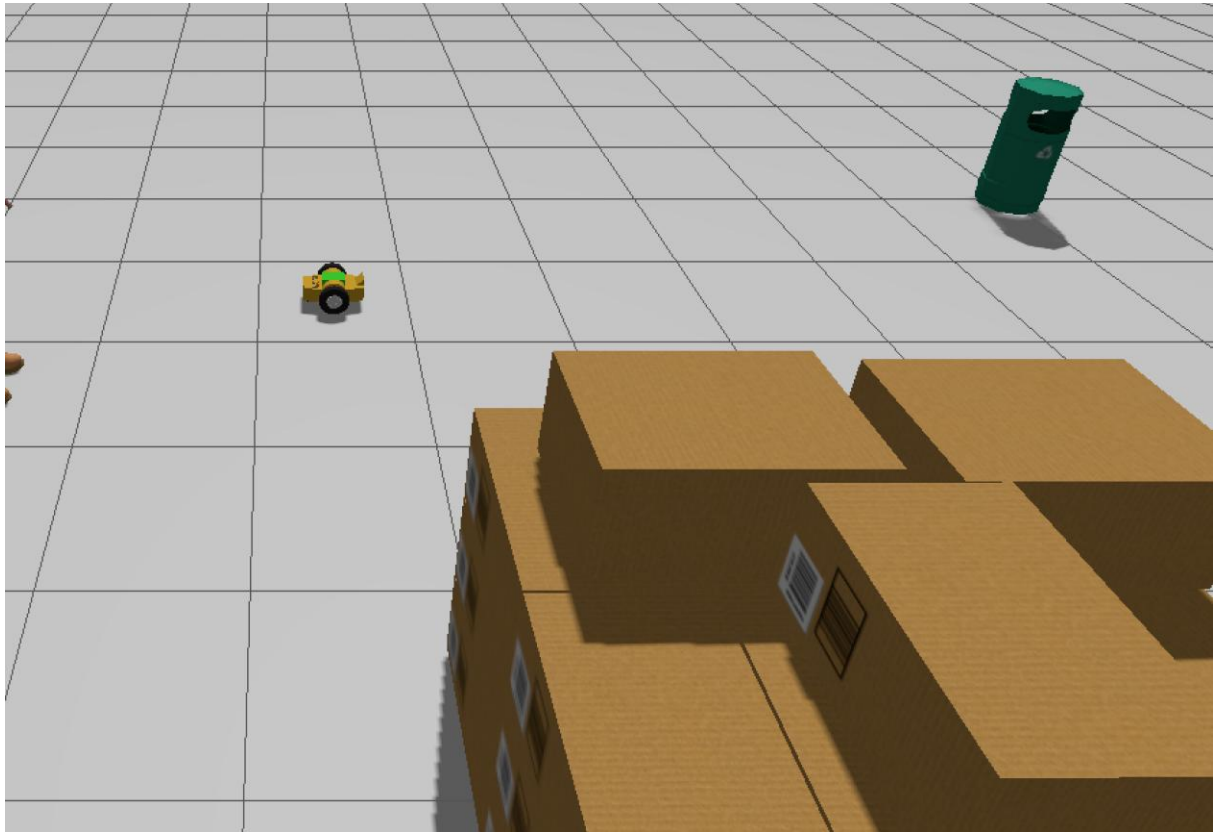
```
 <collision name="collision_chassis">
      <geometry>
        <mesh filename =
"package://erl1/meshes/mogi_bot.dae"/>
      </geometry>
    </collision>
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename =
"package://erl1/meshes/mogi_bot.dae"/>
      </geometry>
      <material name="blue"/>
    </visual>
```

```
install(DIRECTORY
  launch
  worlds
  rviz
  urdf
  meshes
  DESTINATION share/${PROJECT_NAME}
)
```

taken from: https://github.com/MOGI-ROS/Week-3-4-Gazebo-basics

What about sensors?

Let’s start from this package: https://github.com/CarmineD8/erl1_sensors
(inspired from https://github.com/MOGI-ROS/Week-5-6-Gazebo-sensors/tree/starter-branch)



What about sensors?

Let's see recap existing files and folders:

config: We usually store parameters and large configuration files for ROS packages which aren't comfortable to handle from the launchfiles directly.

launch: Default launch files are already part of the starting package, we can test the package with spawn_robot.launch.py.

meshes: this folder contains the 3D models in dae format (collada mesh) that we use for our robot's body, wheels and lidar sensor.

rviz: Pre-configured RViz2 layouts

urdf: The URDF models of our robot

worlds: default Gazebo worlds that we'll use in the simulations.

Camera sensors

For sensors, we will need again to modify our robots

To add a camera - and every other sensors later - we have to change 2 files:

The ***.urdf: we have to define the position, orientation and other physical properties of the camera in this file. This is not necessarily simulation dependent, we have to do these same changes in the urdf in case of a real robot with a real sensor.

The ***.gazebo: this is fully simulation dependent, we have to define the properties of the simulated camera in this file.

In particular, for cameras, we usually have to add two links!

Camera sensors

1 – *camera_link*: this is the actual physical sensors (it represents the actual camera)

2 – *optical_camera_link*: this is a “fake” link, which does not actually exist in the robot’s model. Its purpose is to solve the conflict between 2 different conventional coordinate systems.

By default, URDF uses the right-handed coordinate system with X forward, Y left, and Z up. However, many ROS drivers and vision processing pipelines expect a camera’s optical axis to be aligned with Z forward, X to the right, and Y down.

The joint connecting the two links applies a static rotation so that the camera data will be interpreted correctly by ROS tools that assume the Z-forward convention for image and depth sensors.

What does this mean practically?

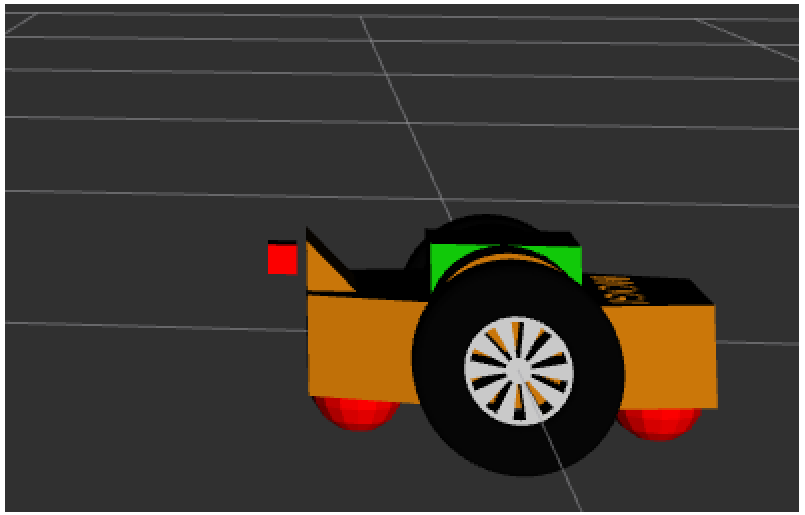
Camera sensors

```
<link name='camera_link'>
   <pose>0 0 0 0 0 0</pose>
   <inertial>
     <mass value="0.1"/>
     <origin xyz="0 0 0" rpy="0 0 0"/>
     <inertia
         ixx="1e-6" ixy="0" ixz="0"
         iyy="1e-6" iyz="0"
         izz="1e-6"
     />
   </inertial>


   <collision name='collision'>
     <origin xyz="0 0 0" rpy="0 0 0"/>
     <geometry>
       <box size=".03 .03 .03"/>
     </geometry>
   </collision>


   <visual name='camera_link_visual'>
     <origin xyz="0 0 0" rpy="0 0 0"/>
     <geometry>
       <box size=".03 .03 .03"/>
     </geometry>
   </visual>


 </link>
```

```
<joint type="fixed" name="camera_joint">
    <origin xyz="0.225 0 0.075" rpy="0 0 0"/>
    <child link="camera_link"/>
    <parent link="base_link"/>
    <axis xyz="0 1 0" />
  </joint>
```

```
  <joint type="fixed"
 name="camera_optical_joint">
    <origin xyz="0 0 0" rpy="-1.5707 0 -
1.5707"/>
    <child link="camera_link_optical"/>
    <parent link="camera_link"/>
  </joint>


  <link name="camera_link_optical">
  </link>
```

Camera sensors



We have now the physical component. To allow the component to work as a camera, we should add a plugin, as we have done for the controller!

```
<gazebo reference="camera_link">
  <sensor name="camera" type="camera">
    <camera>
      <horizontal_fov>1.3962634</horizontal_fov>
      <image>
        <width>640</width>
        <height>480</height>
        <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.1</near>
        <far>15</far>
      </clip>
      <noise>
        <type>gaussian</type>
        <!-- Noise is sampled independently per pixel on each frame.
             That pixel's noise value is added to each of its color
             channels, which at that point lie in the range [0,1]. -->
        <mean>0.0</mean>
        <stddev>0.007</stddev>
      </noise>
      <optical_frame_id>camera_link_optical</optical_frame_id>
      <camera_info_topic>camera/camera_info</camera_info_topic>
    </camera>
    <always_on>1</always_on>
    <update_rate>20</update_rate>
    <visualize>true</visualize>
    <topic>camera/image</topic>
  </sensor>
</gazebo>
```

Camera sensors

```
<gazebo reference="camera_link">
  <sensor name="camera" type="camera">
    <camera>
      <horizontal_fov>1.3962634</horizontal_fov>
      <image>
        <width>640</width>
        <height>480</height>
        <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.1</near>
        <far>15</far>
      </clip>
      <noise>
        <type>gaussian</type>
        <!-- Noise is sampled independently per pixel on each frame.
             That pixel's noise value is added to each of its color
             channels, which at that point lie in the range [0,1]. -->
        <mean>0.0</mean>
        <stddev>0.007</stddev>
      </noise>
      <optical_frame_id>camera_link_optical</optical_frame_id>
      <camera_info_topic>camera/camera_info</camera_info_topic>
    </camera>
    <always_on>1</always_on>
    <update_rate>20</update_rate>
    <visualize>true</visualize>
    <topic>camera/image</topic>
  </sensor>
</gazebo>
```

<gazebo reference="camera_link"> -> we have to refer to the camera_link that we defined in the urdf

<horizontal_fov>1.3962634</horizontal_fov> -> the field of view of the simulated camera

width, height, format and update_rate -> properties of the video stream

<optical_frame_id>camera_link_optical</optical_frame_id> -> we have to use the camera_link_optical that we checked in details above to ensure the right static transformations between the coordinate systems

<camera_info_topic> camera/camera_info</camera_info_topic> -> certain tools like rviz requires a camera_info topic that describes the physical properties of the camera. The topic's name must match camera's topic (in this case both are camera/...)

<topic>camera/image</topic> -> we define the camera topic here

Camera sensors

Keep in mind that with the new Gazebo simulator topics are not automatically forwarded as we already saw it in the previous lesson:

we have to use the parameter_bridge of the ros_gz_bridge package. In other words, we have to extend the arguments of the parameter_bridge in our launch file:

```
gz_bridge_node = Node(
    package="ros_gz_bridge",
    executable="parameter_bridge",
    arguments=[
        "/clock@rosgraph_msgs/msg/Clock[gz.msgs.Clock",
        "/cmd_vel@geometry_msgs/msg/Twist@gz.msgs.Twist",
        "/odom@nav_msgs/msg/Odometry@gz.msgs.Odometry",
        "/joint_states@sensor_msgs/msg/JointState@gz.msgs.Model",
        "/tf@tf2_msgs/msg/TFMessage@gz.msgs.Pose_V",
        "/camera/image@sensor_msgs/msg/Image@gz.msgs.Image",
        "/camera/camera_info@sensor_msgs/msg/CameraInfo@gz.msgs.CameraInfo",


    ],
    output="screen",
    parameters=[
        {'use_sim_time': LaunchConfiguration('use_sim_time')},
    ]
)
```

Very important: you should have this plugin in your world description!

<plugin name='gz::sim::systems::Sensors' filename='gz-sim-sensors-system'/>

Camera sensors

We can see that both /camera/camera_info and /camera/image topics are forwarded. But this is still not the ideal way to forward the camera image from Gazebo. Without compression the 640x480 camera stream consumes almost 20 MB/s network bandwith which is unacceptable for a wireless mobile robot!!

ROS has a very handy feature with it's image transport protocol plugins, it's able to automatically compress the video stream in the background without any additional work. This feature however doesn't work with the parameter_bridge

To activate this feature, we need to run an additional node, the *image_bridge node,* which is defined in the the ros_gz_image package.

Camera sensors

We remove #"/camera/image@sensor_msgs/msg/Image@gz.msgs.Image", from the parameter bridge, since we do not need to forward directly that topic for the reason seen before.

We add the image_bridge node:

```
gz_image_bridge_node = Node(
        package="ros_gz_image",
        executable="image_bridge",
        arguments=[
            "/camera/image",
        ],
        output="screen",
        parameters=[
            {'use_sim_time': LaunchConfiguration('use_sim_time'),
             'camera.image.compressed.jpeg_quality': 75},
        ],
    )
```

Very important: remember to add the new node to the launchDescription:

launchDescriptionObject.add_action(gz_image_bridge_node)

Camera sensors

While you cannot see compressed images from rviz, you can check the differences between the bandwidth of the two topics (raw and compressed) and the related images by using rqt.

If compressed images are not visible in rqt, you have to install the plugins you want to use:

sudo apt install ros-jazzy-compressed-image-transport: for jpeg and png compression
sudo apt install ros-jazzy-theora-image-transport: for theora compression
sudo apt install ros-jazzy-zstd-image-transport: for zstd compression

Camera sensors

We already set up the jpeg quality in the image_bridge node with the following parameter:

'camera.image.compressed.jpeg_quality': 75

But how do we know what is the name of the parameter and what other settings do we can change? To see that we will use the rqt_reconfigure node:

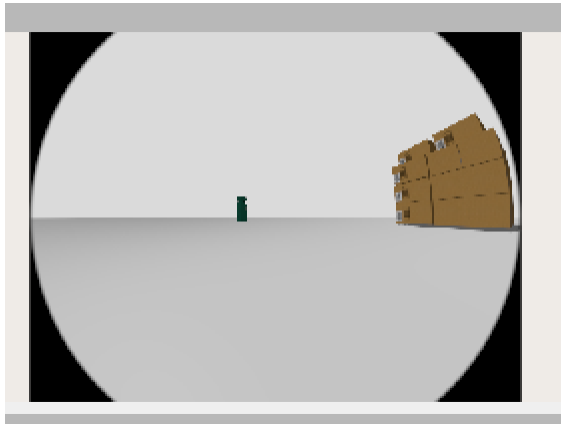
ros2 run rqt_reconfigure rqt_reconfigure

We can play with the parameters here, change the compression or the algorithm as we wish and we can monitor it's impact with rqt.

Camera sensors

Using wide angle or fisheye lens on mobile robots is quite common, but with the conventional camera simulation we cannot simply increase the field of view to get a wide angle distortion. To achieve this we need a different plugin, the wideangle_camera

```
<gazebo reference="camera_link">
  <sensor name="wideangle_camera" type="wideanglecamera">
    <camera>
      <horizontal_fov>3.14</horizontal_fov>
      <image>
        <width>640</width>
        <height>480</height>
      </image>
      <clip>
        <near>0.1</near>
        <far>15</far>
      </clip>
      <optical_frame_id>camera_link_optical</optical_frame_id>
      <camera_info_topic>camera/camera_info</camera_info_topic>
    </camera>
    <always_on>1</always_on>
    <update_rate>20</update_rate>
    <topic>camera/image</topic>
    <gz_frame_id>camera_link</gz_frame_id>
  </sensor>
</gazebo>
```



IMU

An Inertial Measurement Unit (IMU) typically consists of a 3-axis accelerometer, 3-axis gyroscope, and sometimes a 3-axis magnetometer.

It measures linear acceleration, angular velocity, and possibly magnetic heading (orientation). It's important to remember that it's not possible to measure unform motion with an IMU where the velocity is constant (acceleration is zero) and there is no change in the orientation.

Therefore we cannot replace the odometry of the robot with an IMU but with the right technique we can combine these two into a more precise measurement unit.
But first, let's add our IMU to the urdf:

```
<joint name="imu_joint" type="fixed">
  <origin xyz="0 0 0" rpy="0 0 0" />
  <parent link="base_link"/>
  <child link="imu_link" />
</joint>

<link name="imu_link">
</link>
```

, which is a simple link and a fixed joint in the center of the base link

IMU

What about the Gazebo plugin?

```
<gazebo reference="imu_link">
  <sensor name="imu" type="imu">
    <always_on>1</always_on>
    <update_rate>50</update_rate>
    <visualize>true</visualize>
    <topic>imu</topic>
    <enable_metrics>true</enable_metrics>
    <gz_frame_id>imu_link</gz_frame_id>
  </sensor>
</gazebo>
```

Remember to update the parameter bridge node:

"imu@sensor_msgs/msg/Imu@gz.msgs.IMU",

To properly visualize IMU in RViz install the following plugin:
sudo apt install ros-jazzy-rviz-imu-plugin

Also in this case, an additional plugin should be added in the world file, to enable the simulator to estimate IMU outputs

```
<plugin name="gz::sim::systems::Imu" filename="gz-sim-imu-system"/>
```

Localization with IMU!

Sensor fusion is the process of combining data from multiple sensors (possibly of different types) to obtain a more accurate or more complete understanding of a system or environment than could be achieved by using the sensors separately. By merging redundant and complementary information, sensor fusion reduces uncertainty, mitigates individual sensor errors, and provides robust state estimates (e.g., position, velocity, orientation).

- A Kalman Filter (KF) is a mathematical algorithm that estimates the internal state of a system (e.g., position, velocity) based on noisy measurements and a predictive model of how the system behaves. The standard (linear) Kalman Filter assumes the system dynamics (state transitions) and measurement models are linear.

- Real-world systems—especially those involving orientation, rotations, or non-linear sensor models (e.g., fusing IMU acceleration, odometry, GPS position, magnetometer) often do not follow purely linear equations. That's where the Extended Kalman Filter (EKF) comes in

EKF

EKF is a widely used sensor fusion algorithm that handles non-linear system and measurement models by locally linearizing them. Luckily we don't have to bother too much about its implementation in this lesson because we'll use a package that is widely used in robotics applications for years around the world. It's the robot localization package that you can install with the following command:

```
sudo apt install ros-jazzy-robot-localization
```

To configure robot_localization package can be tricky in the beginning, but you find already a config yaml file in the config folder based on the official guidelines.

Important to notice that robot_localization will publish a filtered odometry topic and also a transformation between the robot's base_link and this improved odometry coordinate system.

As you could see from the config file, the robot_localization node will publish a new frame related to the odometry.

EKF

For this reason, after we add the robot_localization node to the launch file:

```
ekf_node = Node(
        package='robot_localization',
        executable='ekf_node',
        name='ekf_filter_node',
        output='screen',
        parameters=[
            os.path.join(pkg_bme_gazebo_sensors, 'config', 'ekf.yaml'),
            {'use_sim_time': LaunchConfiguration('use_sim_time')},
             ]
    )


launchDescriptionObject.add_action(ekf_node)
```

We also remove the tf topic from the parameter bridge

```
#"/tf@tf2_msgs/msg/TFMessage@gz.msgs.Pose_V",
```

EKF

If you look at the list of /tf publishers (ros2 topic info /tf –verbose), you could see that now tf is published by the ekf_filter node. You can also see that there is a /odometry/filtered topic published by the ekf_filter_node.

If you want to actually check how this works, you need to install a few additional nodes:

https://github.com/bit-bots/ros2_python_extension
https://github.com/bit-bots/bitbots_tf_buffer
https://github.com/MOGI-ROS/mogi_trajectory_server

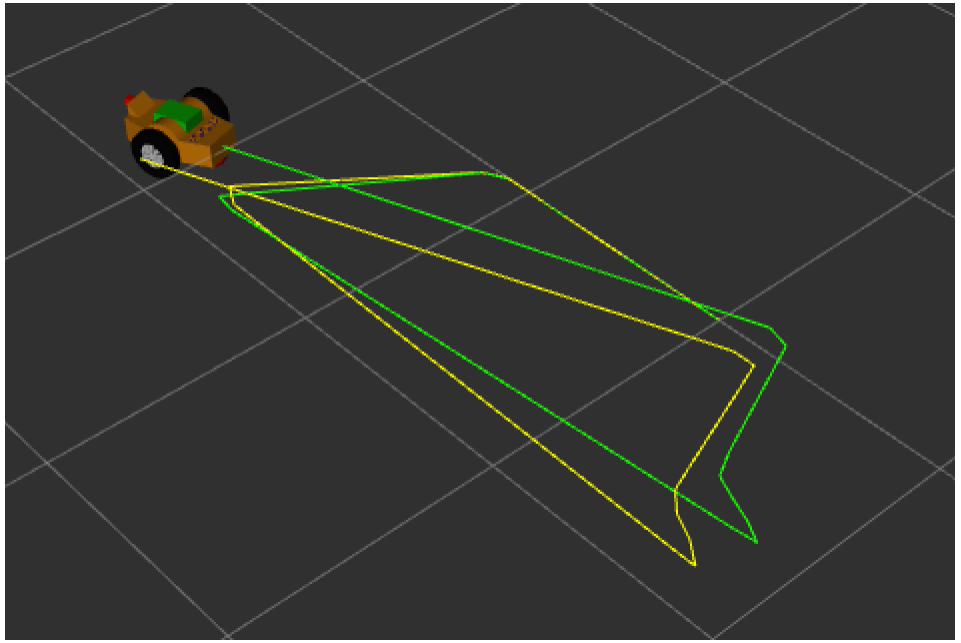
And add two additional nodes in the launch file

```
  trajectory_odom_topic_node = Node(
        package='mogi_trajectory_server',
        executable='mogi_trajectory_server_topic_ba
sed',
        name='mogi_trajectory_server_odom_topic',
        parameters=[{'trajectory_topic':
'trajectory_raw'},
                    {'odometry_topic': 'odom'}]
    )
```

```
        trajectory_node = Node(
            package='mogi_trajectory_server',
            executable='mogi_trajectory_server_topic_based',
            name='mogi_trajectory_server_odomfilt_topic',
            parameters=[{'trajectory_topic':
'filt_trajectory_raw'},
                        {'odometry_topic': "odometry/filtered"}]
        )
```

EKF

We can see that the yellow (raw) odometry starts drifting away from the corrected one very quickly and we can easily bring the robot into a special situation if we drive on a curve and hit the wall. In this case the robot is unable to move and the wheels are slipping. The raw odometry believes from the encoder signals that the robot is still moving on a curve while the odometry after the ekf sensor fusion will believe that the robot moves forward straight. Although none of them are correct, but remember, neither the IMU and neither the odometry can tell if the robot is doing an uniform movement or it's stand still. At least the ekf is able to properly tell that the robot's orientation is not changing regardless what the encoders measure.



GPS

Since we are talking about localization, we can also add a simulated gps in a very simple way:

URDF

```
  <joint name="navsat_joint"
type="fixed">
    <origin xyz="0 0 0" rpy="0 0 0"
/>
    <parent link="base_link"/>
    <child link="navsat_link" />
  </joint>


  <link name="navsat_link">
  </link>
```

Gazebo

```
<gazebo reference="navsat_link">
  <sensor name="navsat" type="navsat">
    <always_on>1</always_on>
    <update_rate>1</update_rate>
    <topic>navsat</topic>
    <gz_frame_id>navsat_link</gz_frame_id>
  </sensor>
</gazebo>
```

World file

```
<plugin
    filename="gz-sim-navsat-system"
    name="gz::sim::systems::NavSat">
  </plugin>
  <!-- Set the coordinates for the world origin -->
  <spherical_coordinates>
    <surface_model>EARTH_WGS84</surface_model>
    <world_frame_orientation>ENU</world_frame_orientation>
    <latitude_deg>44.403373</latitude_deg>
    <longitude_deg>8.9481068</longitude_deg>
    <elevation>0</elevation>
    <heading_deg>0</heading_deg>
  </spherical_coordinates>
```

Parameter bridge (launch file)

```
"/navsat@sensor_msgs/msg/NavSatFix@gz.msgs.NavSat",
```

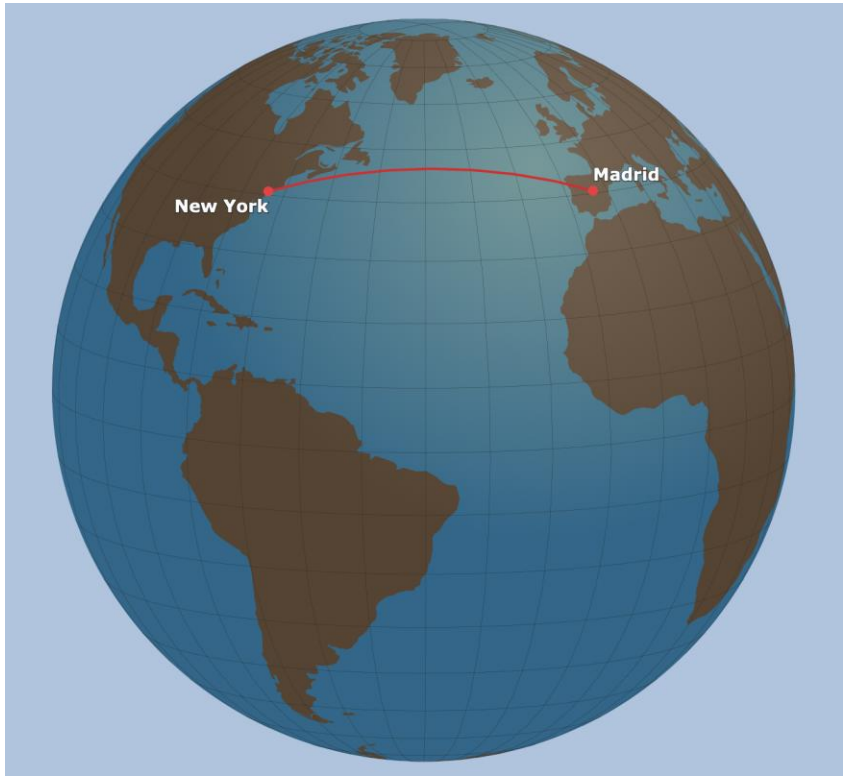
GPS

What we can do with this data?

We can use it for robot localization, by using the Haversine formula.

It is used to calculate the great-circle distance between two points on a sphere (e.g. the Earth) from their latitudes and longitudes. It accounts for the spherical shape of the planet, making it more accurate than simple Euclidean distance formulas when dealing with geographical coordinates.

The formula results in a distance and bearing between 2 points on the sphere, where the bearing is measured clockwise from north. 0° bearing corresponds to North, 90° bearing corresponds to East, etc.



GPS

This formula is used in the gps_follower script that you find in the package.

Please install:

sudo apt install ros-jazzy-rviz-satellite
sudo apt install ros-jazzy-tf-transformations

The script uses the haversine function, computing a distance in meters and radians. To get the yaw angle of the robot we will have to convert the robot's orientation in quaternion into Euler angles.

Indeed, the node subscribes both to the imu data and gps information. It uses gps information for computing the correct bearing for reaching the target, and aligns it with IMU information. For the distance, it only uses GPS, but in principle we could fuse information also in this scenario.

To correctly start rviz, do the following:

ros2 launch bme_gazebo_sensors spawn_robot.launch.py world:=empty.sdf rviz_config:=gps.rviz x:=0.0 y:=0.0 yaw:=0.0

LIDAR

LIDAR (an acronym for “Light Detection and Ranging” or “Laser Imaging, Detection, and Ranging”) is a sensing technology that uses laser light to measure distances. LIDAR sensor typically emits pulses of laser light in a scanning pattern (2D or 3D) and measures how long it takes for the light to return after hitting nearby objects. From this, the system computes distances to obstacles or surfaces in the environment.

By continuously scanning the surroundings, the LIDAR provides a 2D or 3D map of distances to any objects around the robot. Lidars are simple and important sensors of almost every mobile robot applications, it's widely used in Simultaneous Localization and Mapping (SLAM) algorithms which use LIDAR scans to build a map of the environment in real time while also estimating the robot’s pose (position and orientation) within that map.

LIDAR

As usual we need to modify: - URDF

```
<joint type="fixed" name="scan_joint">
"/>
  <child link="scan_link"/>
  <parent link="base_link"/>
  <axis xyz="0 1 0" rpy="0 0 0"/>
</joint>


<link name='scan_link'>
  <inertial>
    <mass value="1e-5"/>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <inertia
        ixx="1e-6" ixy="0" ixz="0"
        iyy="1e-6" iyz="0"
        izz="1e-6"
    />
  </inertial>
  <collision name='collision'>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size=".1 .1 .1"/>
    </geometry>
  </collision>


  <visual name='scan_link_visual'>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
                                    <mesh     filename     =
"package://bme_gazebo_sensors/meshes/lidar.dae"/>
    </geometry>
  </visual>
</link>
```

Gazebo file

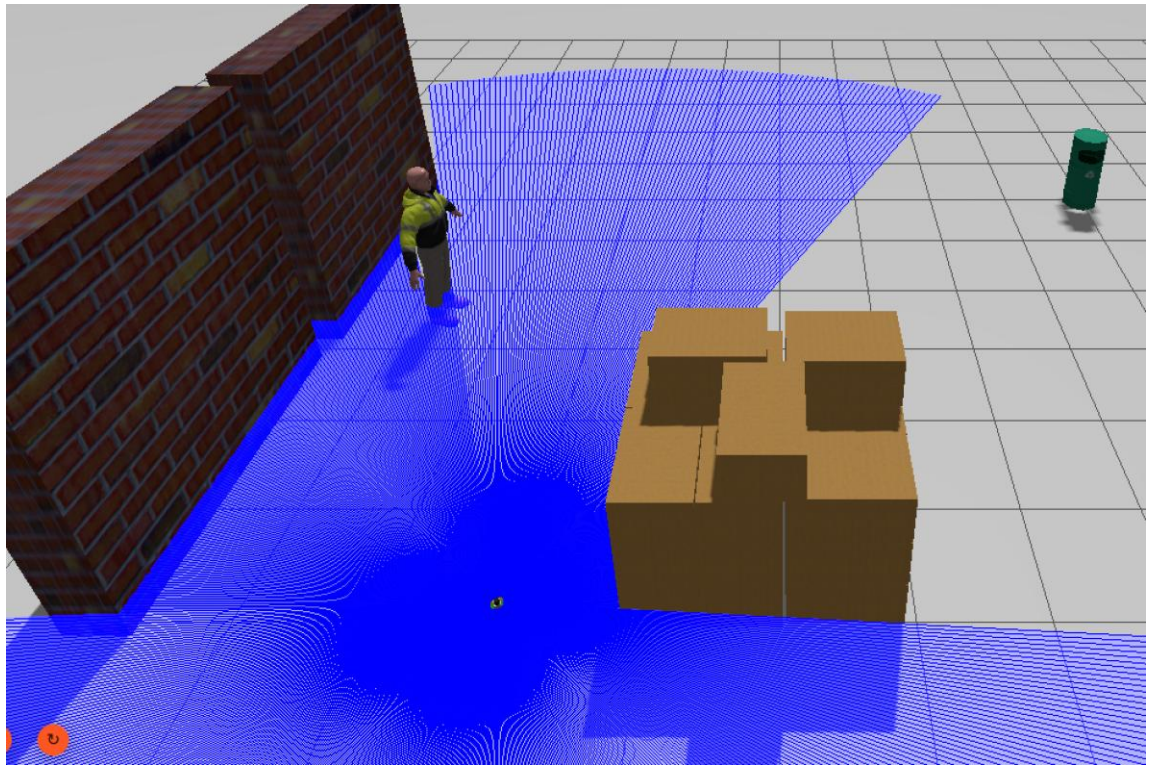
```
<gazebo reference="scan_link">
  <sensor name="gpu_lidar" type="gpu_lidar">
    <update_rate>10</update_rate>
    <topic>scan</topic>
    <gz_frame_id>scan_link</gz_frame_id>
    <lidar>
      <scan>
        <horizontal>
          <samples>720</samples>
          <!--(max_angle-min_angle)/samples * resolution -
->
          <resolution>1</resolution>
          <min_angle>-3.14156</min_angle>
          <max_angle>3.14156</max_angle>
        </horizontal>
        <!-- Dirty hack for fake lidar detections with
ogre 1 rendering in VM -->
        <!-- <vertical>
            <samples>3</samples>
            <min_angle>-0.001</min_angle>
            <max_angle>0.001</max_angle>
        </vertical> -->
      </scan>
      <range>
        <min>0.05</min>
        <max>10.0</max>
        <resolution>0.01</resolution>
      </range>
      <noise>
          <type>gaussian</type>
          <mean>0.0</mean>
          <stddev>0.01</stddev>
      </noise>
      <frame_id>scan_link</frame_id>
    </lidar>
    <always_on>1</always_on>
    <visualize>true</visualize>
  </sensor>
</gazebo>
```

LIDAR

…and add the topic in the parameter bridge:

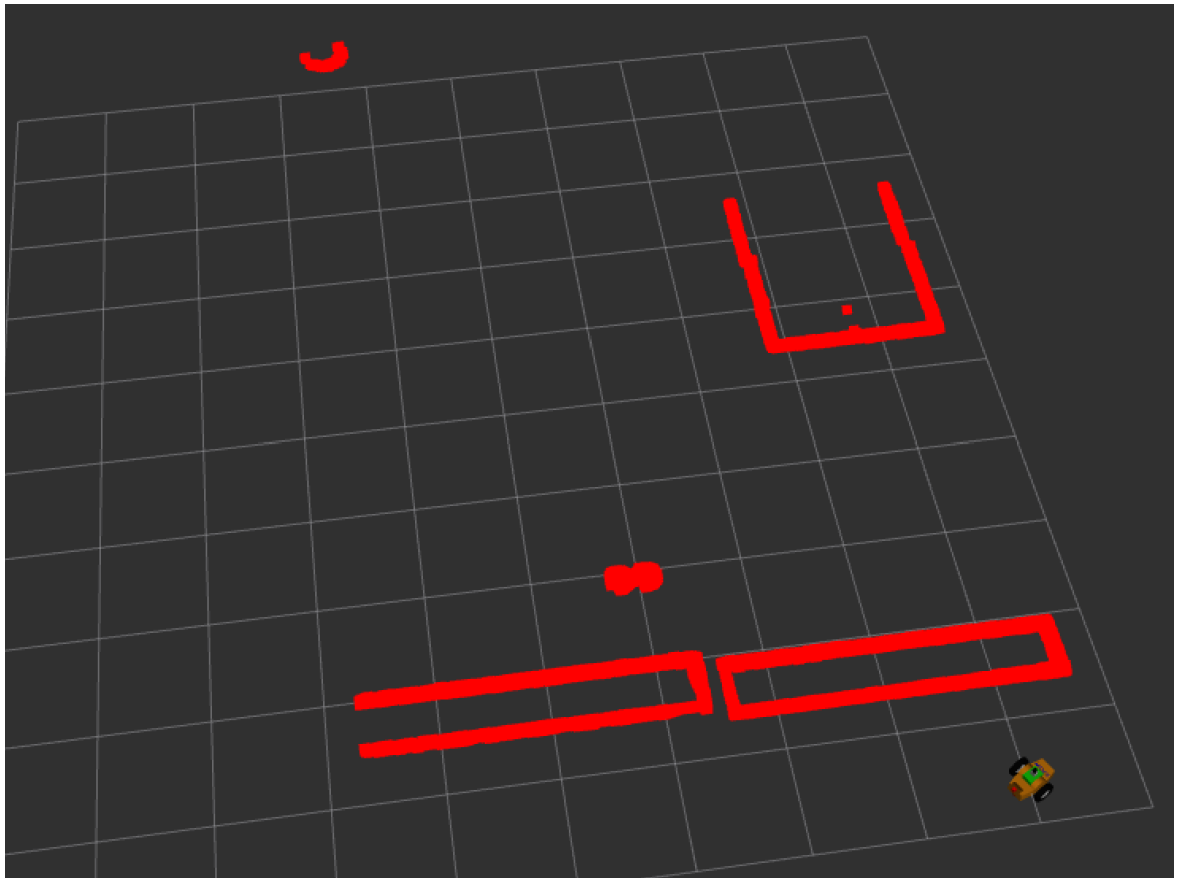
```
"/scan@sensor_msgs/msg/LaserScan@gz.msgs.LaserScan",
```

We can also verify the rendering of lidars in Gazebo with the Visualize Lidar tool



LIDAR

If we increase decay time of the visualization of lidar scans and we drive around the robot we can do a very simple "mapping" of the environment. Although in the course we will see that mapping algorithms are more complicated, usually this a good quick and dirty test on real robots if odometry, lidar scan and the other components are working well together



3D LIDAR

If we want to simulate a 3D lidar we only have to increase the number of vertical samples together with the minimum and maximum angles. For example the following vertical parameters are matching a Velodyne VLP-32 sensor:

```
<vertical>
      <samples>32</samples>
      <min_angle>-0.5353</min_angle>
      <max_angle>0.1862</max_angle>
</verticaI>
```

Again, we need to add one more topic in our launch file:

```
"/scan/points@sensor_msgs/msg/PointCloud2@gz.msgs.PointCloudPacked",
```

3D LIDAR

As before, we could increase the decay time just as we did with the 2D points.