Robotics and Intelligent Systems Lab

Lecture 5

Bilal Wehbe

WiSe 21/22

1. Recap
2. Private Names in roscpp
3. Object Oriented Programming in roscpp
4. ROS Actions
5. Recording and Playing back Data
6. RViz

Recap

Publisher & Subscriber in roscpp

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");

  ros::NodeHandle n;

  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);

  ros::Rate loop_rate(10);

  int count = 0;
  while (ros::ok())
  {
    /**
     * This is a message object. You stuff it with data, and then publish it.
     */
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());

    chatter_pub.publish(msg);

    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }

  return 0;
}
```

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");

  ros::NodeHandle n;

  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

  ros::spin();

  return 0;
}
```

Server & Client in roscpp

```cpp
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"

bool add(beginner_tutorials::AddTwoInts::Request  &req,
         beginner_tutorials::AddTwoInts::Response &res)
{
  res.Sum = req.A + req.B;
  ROS_INFO("request: x=%ld, y=%ld", (long int)req.A, (long int)rec
  ROS_INFO("sending back response: [%ld]", (long int)res.Sum);
  return true;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_server");
  ros::NodeHandle n;

  ros::ServiceServer service = n.advertiseService("add_two_ints",
  ROS_INFO("Ready to add two ints.");
  ros::spin();

  return 0;
}
```

```cpp
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"
#include <cstdlib>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_client");
  if (argc != 3)
  {
    ROS_INFO("usage: add_two_ints_client X Y");
    return 1;
  }

  ros::NodeHandle n;
  ros::ServiceClient client =
  n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_
  beginner_tutorials::AddTwoInts srv;
  srv.request.A = atoll(argv[1]);
  srv.request.B = atoll(argv[2]);
  if (client.call(srv))
  {
    ROS_INFO("Sum: %ld", (long int)srv.response.Sum);
  }
  else
  {
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
  }

  return 0;
```

Server & Client in rospy

```python
#!/usr/bin/env python

from beginner_tutorials.srv import AddTwoInts,AddTwoIntsResponse
import rospy

def handle_add_two_ints(req):
    print("Returning [%s + %s = %s]"%(req.A, req.B, (req.A + req.B)))
    return AddTwoIntsResponse(req.A + req.B)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints
    print("Ready to add two ints.")
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

```python
#!/usr/bin/env python

import sys
import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        req = AddTwoIntsRequest()
        req.A = x
        req.B = y
        resp1 = add_two_ints(req) # (x, y) can be passed directly as
        return resp1.Sum
    except rospy.ServiceException as e:
        print("Service call failed: %s"%e)

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print("usage: add_two_ints_client.py X Y")
        sys.exit(1)
    print("Requesting %s+%s"%(x, y))
    print("%s + %s = %s"%(x, y, add_two_ints_client(x, y)))
```

Parameters in rospy & roscpp

Parameters in rospy

- rospy.get_param('/global_param_name')
- rospy.get_param('param_name')
- rospy.get_param('~private_param_name')
- rospy.set_param('param_name', value)
- rospy.delete_param('param_name')
- rospy.has_param('param_name')
- rospy.delete_param('param_name')
- rospy.resolve_name('param_name')
- rospy.search_param('param_name')

Parameters in roscpp

- bool getParam (const std::string& key, parameter_type& output_value)
- n.param("my_num", i, 42);
- n.setParam("my_param", "hello there");
- n.deleteParam("my_param");
- n.hasParam("my_param")
- n.searchParam('param_name', result))

Private Names in roscpp

Accessing private names with NodeHandle

- When NodeHandles were introduced it created a conundrum when dealing with private names.
- This is since a node handle can be instantiated with is own namespace

```
ros::init(argc, argv, "my_node_name");
ros::NodeHandle nh("/my_node_handle_namespace");
```

- Where should a private name resolve? Some options would be

Accessing private names with NodeHandle

- When NodeHandles were introduced it created a conundrum when dealing with private names.
- This is since a node handle can be instantiated with is own namespace

```
ros::init(argc, argv, "my_node_name");
ros::NodeHandle nh("/my_node_handle_namespace");
```

- Where should a private name resolve? Some options would be
  - `/my_node_handle_namespace/my_node_name/name`
  - `my_node_name/my_node_handle_namespace/name`
  - `/my_node_handle_namespace/name`
  - Something else entirely
- For this reason, NodeHandle does not allow passing private names directly to its methods, or to constructors that take a NodeHandle as an argument.

Accessing private names with NodeHandle

- The solution is to construct a NodeHandle with a private name as its namespace

```
ros::init(argc, argv, "my_node_name");
ros::NodeHandle nh1("~"); // must be in main()
ros::NodeHandle nh2("~foo");
```

- nh1's namespace is `/my_node_name`, and nh2's namespace is `/my_node_name/foo`.

Accessing private names with NodeHandle

- The solution is to construct a NodeHandle with a private name as its namespace

```
ros::init(argc, argv, "my_node_name");
ros::NodeHandle nh1("~"); // must be in main()
ros::NodeHandle nh2("~foo");
```

- nh1's namespace is `/my_node_name`, and nh2's namespace is `/my_node_name/foo`.
- So instead of doing this:

```
ros::NodeHandle nh;
nh.getParam("~name", ... );
```

- You do this

```
ros::NodeHandle nh("~");
nh.getParam("name", ... );
```

Object Oriented Programming in roscpp

Using Class Methods as Callbacks

- Let's take a simple Listener class

```
class Listener {
   public:
   void callback(const std_msgs::String::ConstPtr& msg);
};
```

Using Class Methods as Callbacks

- Let's take a simple Listener class

```
class Listener {
   public:
   void callback(const std_msgs::String::ConstPtr& msg);
};
```

- Previously we used to define a subscriber using a nodehandle as

```
ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
```

Using Class Methods as Callbacks

- Let's take a simple Listener class

```
class Listener {
   public:
   void callback(const std_msgs::String::ConstPtr& msg);
};
```

- Previously we used to define a subscriber using a nodehandle as

```
ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
```

- With class method as a subscriber it will look like this

```
Listener listener;
ros::Subscriber sub = n.subscribe("chatter", 1000, &Listener::callback, &listener);
```

- If the subscriber is inside the class Listener, you can replace the last argument with the keyword `this`, which means that the subscriber will refer to the class it is part of.

Listener Class with Callbacks

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"


class Listener
{
public:
  void callback(const std_msgs::String::ConstPtr& msg);
};

void Listener::callback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}
int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener_class");
  ros::NodeHandle n;

  Listener listener;
  ros::Subscriber sub = n.subscribe("chatter", 1000, &Listener::callback, &listener);

  ros::spin();

  return 0;
}
```

Listener Class with Subscriber inside

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"


class Listener
{
public:
  Listener();

  ros::NodeHandle n;
  ros::Subscriber sub;

  void callback(const std_msgs::String::ConstPtr& msg);
};

Listener::Listener()
{
  sub = n.subscribe("chatter", 1000, &Listener::callback, this);
}

void Listener::callback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener_class");
  Listener listener;

  ros::spin();

  return 0;
}
```

Exercise

- re-write the signal generator node as a cpp class and publish the generated signal on a new topic

ROS Actions

ROS actionlib

- In any large ROS based system, there are cases when someone would like to send a request to a node to perform some task, and also receive a reply to the request.
  - This can currently be achieved via ROS services.

ROS actionlib

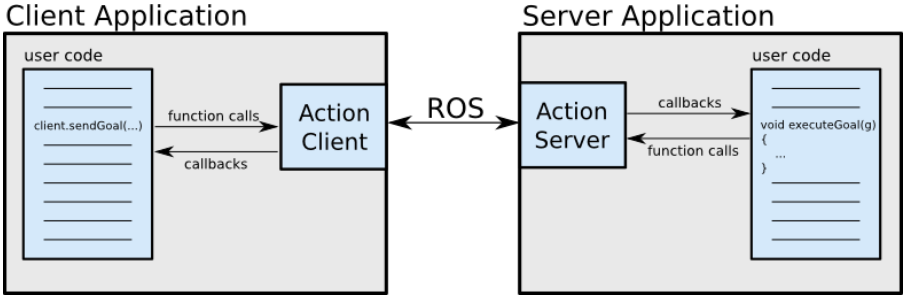
- In any large ROS based system, there are cases when someone would like to send a request to a node to perform some task, and also receive a reply to the request.
  - This can currently be achieved via ROS services.
- In some cases, if the service takes a long time to execute, the user might want the ability to cancel the request during execution or get periodic feedback about how the request is progressing.

ROS actionlib

- In any large ROS based system, there are cases when someone would like to send a request to a node to perform some task, and also receive a reply to the request.
  - This can currently be achieved via ROS services.
- In some cases, if the service takes a long time to execute, the user might want the ability to cancel the request during execution or get periodic feedback about how the request is progressing.
- The actionlib package provides tools to create servers that execute long-running goals that can be preempted.

Client-Server Interaction

- The ActionClient and ActionServer communicate via a ”ROS Action Protocol”, which is built on top of ROS messages.
- The client and server then provide a simple API for users to request goals (on the client side) or to execute goals (on the server side)



Goal, Feedback, & Result

Goal

To accomplish tasks using actions, a goal can be sent to an *ActionServer* by an *ActionClient*

- Example is controlling a tilting laser scanner, the goal would contain the scan parameters (min angle, max angle, speed, etc).

Goal, Feedback, & Result

Goal

To accomplish tasks using actions, a goal can be sent to an *ActionServer* by an *ActionClient*

- Example is controlling a tilting laser scanner, the goal would contain the scan parameters (min angle, max angle, speed, etc).

Feedback

Provides a way to tell an ActionClient about the incremental progress of a goal

- Example: the time left until the scan completes

Goal, Feedback, & Result

Goal

To accomplish tasks using actions, a goal can be sent to an *ActionServer* by an *ActionClient*

- Example is controlling a tilting laser scanner, the goal would contain the scan parameters (min angle, max angle, speed, etc).

Feedback

Provides a way to tell an ActionClient about the incremental progress of a goal

- Example: the time left until the scan completes

Result

Is sent from the ActionServer to the ActionClient upon completion of the goal.

- Example: a point cloud generated from the requested scan.

The .action file

- The action specification is defined using a .action file.
- action files are stored under `action/DoDishes.action`

```
# Define the goal
uint32 dishwasher_id # Specify which dishwasher we want to use
---
# Define the result
uint32 total_dishes_cleaned
---
# Define a feedback message
float32 percent_complete
```

- 6 messages are automatically generated in order for the client and server to communicate

Building .action files

- CMakeLists.txt file before catkin_package() `action/DoDishes.action`

```
find_package(catkin REQUIRED genmsg actionlib_msgs)
add_action_files(DIRECTORY action FILES DoDishes.action)
generate_messages(DEPENDENCIES actionlib_msgs)
```

- package.xml

```
<build_depend>actionlib</build_depend>
<build_depend>actionlib_msgs</build_depend>
<exec_depend>actionlib</exec_depend>
<exec_depend>actionlib_msgs</exec_depend>
```

Action Server

```
1 #! /usr/bin/env python
2
3 import roslib
4 roslib.load_manifest('beginner_tutorials')
5 import rospy
6 import actionlib
7
8 from beginner_tutorials.msg import DoDishesAction, DoDishesFeedback, DoDishesResult
9
10 class DoDishesServer:
11   def __init__(self):
12     self.server = actionlib.SimpleActionServer('do_dishes', DoDishesAction,
13         self.execute, False)
14     self._feedback = DoDishesFeedback()
15     self._result = DoDishesResult()
16     self.server.start()
17
18   def execute(self, goal):
19     # Do lots of awesome groundbreaking robot stuff here
20     r = rospy.Rate(2)
21     n_dishes = 10
22     for i in range(1,n_dishes+1):
23         # publish the feedback
24         self._feedback.percent_complete = i/n_dishes*100.0
25         self.server.publish_feedback(self._feedback)
26
27         self._result.total_dishes_cleaned = i
28         r.sleep()
29     self.server.set_succeeded(self._result)
30
31
32 if __name__ == '__main__':
33   rospy.init_node('do_dishes_server')
34   server = DoDishesServer()
35   rospy.spin()
```

Action Client

```
#! /usr/bin/env python

import roslib
roslib.load_manifest('beginner_tutorials')
import rospy
import actionlib

from beginner_tutorials.msg import DoDishesAction, DoDishesGoal

def get_feedback(msg):
    print(msg)

if __name__ == '__main__':
    rospy.init_node('do_dishes_client')
    client = actionlib.SimpleActionClient('do_dishes', DoDishesAction)
    client.wait_for_server()

    goal = DoDishesGoal()
    # Fill in the goal here
    goal.dishwasher_id = 10
    client.send_goal(goal, feedback_cb = get_feedback)

    client.wait_for_result()

    result = client.get_result()
    print(result)
```

Running an Action Client & Server

- add your nodes to the CMakeLists.txt and build the package
- run the server

```
$ rosrun beginner_tutorials do_dishes_server.py
```

- run the client

```
$ rosrun beginner_tutorials do_dishes_client.py
```

```
▶ rosrun beginner_tutorials do_dishes_client.py
percent_complete: 10.0
percent_complete: 20.0
percent_complete: 30.0
percent_complete: 40.0
percent_complete: 50.0
percent_complete: 60.0
percent_complete: 70.0
percent_complete: 80.0
percent_complete: 90.0
percent_complete: 100.0
total_dishes_cleaned: 10
```

More on actionlib

- https://wiki.ros.org/actionlib/DetailedDescription
- https://wiki.ros.org/actionlib_tutorials/Tutorials

Recording and Playing back Data

rosbag

- A bag is a file format in ROS for storing ROS message data.
- The `rosbag` command can record, replay and manipulate bags.
  - `record`: Record a bag file with the contents of specified topics.
  - `play`: Play back the contents of one or more bag files in a time-synchronized fashion.
  - `info`: Summarize the contents of one or more bag files.
  - `compress`: Compress one or more bag files.
  - `decompress`: Decompress one or more bag files.
  - `check`: Determine whether a bag is playable in the current system, or if it can be migrated.
  - `fix`: Repair the messages in a bag file so that it can be played in the current system.
  - `reindex`: Reindexes one or more bag files.

Creating bag files

- Start the following commands
  - **Terminal 1:** `$ roscore`
  - **Terminal 2:** `$ rosrun turtlesim turtlesim_node`
  - **Terminal 3:** `$ rosrun turtlesim turtle_teleop_key`
  - **Terminal 4:** `$ rostopic list -v`

```
▶ rostopic list -v

Published topics:
 * /rosout_agg [rosgraph_msgs/Log] 1 publisher
 * /rosout [rosgraph_msgs/Log] 2 publishers
 * /turtle1/pose [turtlesim/Pose] 1 publisher
 * /turtle1/color_sensor [turtlesim/Color] 1 publisher
 * /turtle1/cmd_vel [geometry_msgs/Twist] 1 publisher

Subscribed topics:
 * /rosout [rosgraph_msgs/Log] 1 subscriber
 * /turtle1/cmd_vel [geometry_msgs/Twist] 1 subscriber
```

Creating bag files

- now create a new directory called bagfiles

```
$ mkdir bagfiles
$ cd bagfiles
```

- now record the topics being published using

```
$ rosbag record -a
```

- `-a` indicates that all the topics should be accumulated in that bag file.
- next select the terminal you started `turtle_teleop_key` in and move around the turtle while the bag file is being recorded

Inspecting bag files

- Stop recording by simply pressing Crtl-C, then display the details of the recorded bagfile using

```
$ rosbag info <your bagfile>
```

```
▶ rosbag info 2021-10-03-15-55-15.bag
path:        2021-10-03-15-55-15.bag
version:     2.0
duration:    27.5s
start:       Oct 03 2021 15:55:15.95 (1633269315.95)
end:         Oct 03 2021 15:55:43.43 (1633269343.43)
size:        245.6 KB
messages:    3447
compression: none [1/1 chunks]
types:       geometry_msgs/Twist [9f195f881246fdfa2798d1d3eebca84a]
             rosgraph_msgs/Log   [acffd30cd6b6de30f120938c17c593fb]
             turtlesim/Color     [353891e354491c51aabe32df673fb446]
             turtlesim/Pose      [863b248d5016ca62ea2e895ae5265cf9]
topics:      /rosout                3 msgs    : rosgraph_msgs/Log
             /turtle1/cmd_vel          38 msgs    : geometry_msgs/Twist
             /turtle1/color_sensor   1703 msgs    : turtlesim/Color
             /turtle1/pose           1703 msgs    : turtlesim/Pose
```

Replaying bag files

- Stop the running nodes and keep roscore running
- You can replay the bagfile simply using

```
$ rosbag play <your bagfile>
```

```
▶ rosbag play 2021-10-03-15-55-15.bag
[ INFO] [1633269608.479104817]: Opening 2021-10-03-15-55-15.bag

Waiting 0.2 seconds after advertising topics... done.

Hit space to toggle paused, or 's' to step.
 [RUNNING]  Bag Time: 1633269315.947732   Duration: 0.000000 / 27.484337
 [RUNNING]  Bag Time: 1633269315.948517   Duration: 0.000785 / 27.484337
 [RUNNING]  Bag Time: 1633269316.049132   Duration: 0.101400 / 27.484337
 [RUNNING]  Bag Time: 1633269316.149411   Duration: 0.201679 / 27.484337
 [RUNNING]  Bag Time: 1633269316.200076   Duration: 0.252344 / 27.484337
 [RUNNING]  Bag Time: 1633269316.216548   Duration: 0.268816 / 27.484337
 [RUNNING]  Bag Time: 1633269316.231919   Duration: 0.284187 / 27.484337
 [RUNNING]  Bag Time: 1633269316.248009   Duration: 0.300278 / 27.484337
 [RUNNING]  Bag Time: 1633269316.264830   Duration: 0.317098 / 27.484337
 [RUNNING]  Bag Time: 1633269316.280106   Duration: 0.332374 / 27.484337
 [RUNNING]  Bag Time: 1633269316.296433   Duration: 0.348701 / 27.484337
 [RUNNING]  Bag Time: 1633269316.311635   Duration: 0.363903 / 27.484337
 [RUNNING]  Bag Time: 1633269316.327941   Duration: 0.380209 / 27.484337
```

Replaying bag files

- Playing a bag file will start publishig data to the recorded topic
- You can inspect the list of topics and echo a certain topic to check the data being replayed

```
▶ rostopic echo /turtle1/cmd_vel
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

- start the `turtlesim_node` again alone without `turtle_teleop_key`, replay the bag file then observe the turtlesim

Recording a subset of the data

- Logging only topics of a certain node

```
$ rosbag record <node name>
```

- Logging topics by name

```
$ rosbag record <topic name>
```

- Logging selected topics

```
$ rosbag record -O subset <topic 1> <topic 2> ...
```

Converting bag file into human-readable

- Download this demo bag file from the following

```
$ wget https://open-source-webviz-ui.s3.amazonaws.com/demo.bag
```

- Inspect the topics logged in this bag file

```
$ rosbag info demo.bag
```

- You can convert the data on a certain topic by using

```
$ rostopic echo /obs1/gps/fix | tee topic1.yaml
```

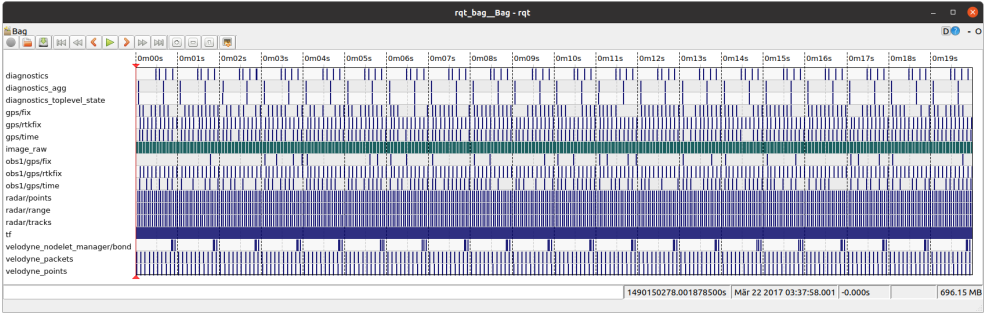
- Then playing the rosbag unsing the tag `--immediate`

```
$ time rosbag play --immediate demo.bag --topics /obs1/gps/fix
```

rqt_bag

- `rqt_bag` is a gui that allows you to visualize a bag file and replay it

```
$ rosrun rqt_bag rqt_bag
```



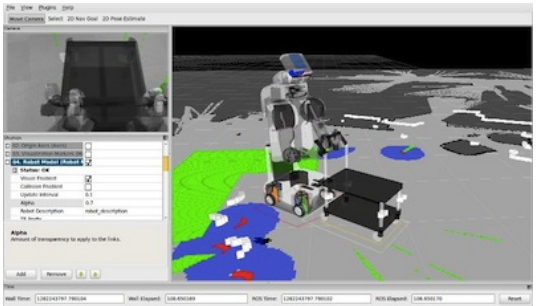
RViz

RViz

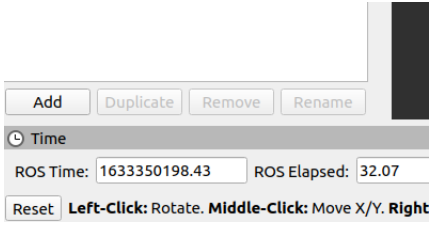
- 3D Visualization tool for ROS
- Subscribes to topics like a node
- Different camera views
- Interactive tools to publish user information
- Extensible with plugins

run using

```
$ rosrun rviz rviz
```

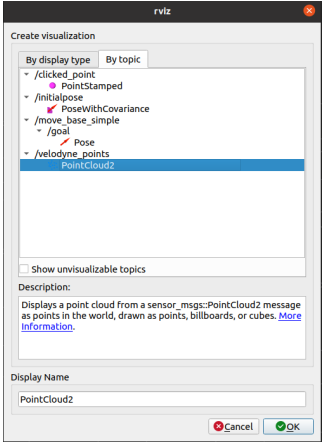


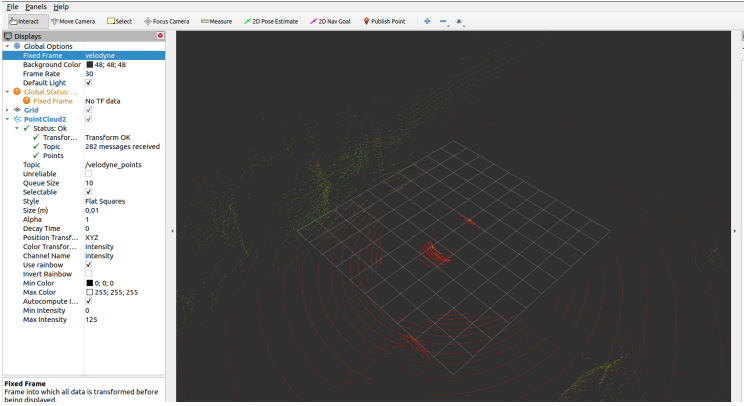
RViz Display plugins



- AccelStamped
- Axes
- Camera
- DepthCloud
- Effort
- FluidPressure
- Grid
- GridCells
- Group
- Illuminance
- Image
- InteractiveMarkers
- LaserScan
- Map
- Marker
- MarkerArray
- Odometry
- Path
- PointCloud
- PointCloud2
- PointStamped
- Polygon
- Pose
- PoseArray
- PoseWithCovariance
- Range
- RelativeHumidity
- RobotModel
- TF
- Temperature
- TwistStamped
- WrenchStamped

RViz Visualize PointCloud





RViz Displays