CS 378 - Autonomous Vehicles in Traffic I
Week 6b - Velodyne Processing, rosbag, rviz
Announcements

• Programming assignment 2 due tonight
• I will try and discuss some of the common problems faced by students in the first 2 programming assignments sometime next week
• Programming Assignment 3 should be up by tonight. I'll discuss it next week on Monday
• Make sure you have commit permissions and svn commit is working at least a few hours before the deadline.
• Possible car visit on Tuesday (2/28). I will coordinate with you guys on Monday. We will also have the car on campus (Parking lot north of ACE building) for Explore UT on Saturday (3/3). I am looking for 2-3 volunteers, will send email sometime tonight.
Today

• Today we'll see how to use Velodyne data
• We'll take a look at our obstacle detection code
• We'll look at recording data using the `rosbag` tool and playing it back
• We'll take a look at the main visualization software in ROS - `rviz`.
Code reorganization

• Jack has been reorganizing code in the Velodyne stack to create a cleaner interface. Some of these changes are still in progress.
• To get an updated copy of the code, run the `svn update` command.
  ○ `cd ~/svn/utexas-art-ros-pkg`
  ○ `svn up`
  ○ `rosmake art_run`
• You will need to do this to get the examples from today
The *velodyne_driver* package

- The driver gets raw packet data from the Velodyne, and transmits it in raw form inside the ROS ecosystem.
- This data is still needs to be processed to get a pointcloud like the one you saw on Monday.
- We record this raw data, as it uses up far less space than a pointcloud (more about recording data in later slides).

Raw data from the Velodyne

The `velodyne_driver` package

- The driver publishes the information as:
  - `/velodyne_packets` [velodyne_msgs/VelodyneScan]
  - Note the 1206 bytes in the data array - they contain the packet as it is

- To run the driver (at the car)
  - `rosrun velodyne_driver velodyne_node`

- Given the `rotational_position`, `laser_block`, `distance` and `intensity`, we can compute the position and intensity of each point
- With 260 such packets, we can compute one entire revolution (when running at 10Hz)
The velodyne_pointcloud package

- It accepts the raw data and publishes out the information in a well defined point cloud structure that can be understood by ROS nodes.
- The cloud_node node subscribes to:
  - /velodyne_packets [velodyne_msgs/VelodyneScan]
- It publishes:
  - /velodyne_points [sensor_msgs/PointCloud2]
- This is the same message type that was being published by some node during Monday's demo.
- Aside: Why is it called PointCloud2?
  - Nothing is perfect. The earlier format [sensor_msgs/PointCloud] was quite inefficient.
The *cloud_node* node

- Running this node:
  - `rosrun velodyne_pointcloud cloud_node`

- You will have to run this node when playing back recorded data
  - We store `/velodyne_packets`, not `/velodyne_points`

- OK, so we have this node publishing point clouds. How do I use it?
  - Let's take a look at the example code provided for this
    - You will have to update your code to get this example:
      - `roscd art_examples`
      - `svn up`
Writing a subscriber

- I have a very basic subscriber in the `velodyne_example` package
  - `roscd velodyne_example`
  - `gedit src/subscriber.cc`

- Running the subscriber
  - `rosrun velodyne_example subscriber`
subscriber.cc

```c++
#include <ros/ros.h>
#include <pcl_ros/point_cloud.h>
#include <pcl/point_types.h>
#include <boost/foreach.hpp>

typedef pcl::PointXYZI VPoint;
typedef pcl::PointCloud<VPoint> VPointCloud;

void callback(const VPointCloud::ConstPtr& msg) {
  ROS_INFO("Cloud: width = %d, height = %d", msg->width, msg->height);
  /** A good way to access all the points */
  //BOOST_FOREACH(const VPoint& point, msg->points) {
    /** Access point values using */
    // point.x
    // point.y
    // point.z
    // point.intensity
  //}
}

int main(int argc, char** argv) {
  ros::init(argc, argv, "velodyne_example_sub");
  ros::NodeHandle nh;
  ros::Subscriber sub =
    nh.subscribe<VPointCloud>("/velodyne_points", 1, callback);
  ros::spin();
}
```
Hold on!

• There was a bit of trickery in the last few slides
  ◦ The code in the last slide was subscribing to \texttt{pcl::PointCloud<pcl::PointXYZI>}
  ◦ In earlier slides I said I was publishing a message type called \texttt{sensor_msgs::PointCloud2}
  ◦ ?!??!

• The truth is that using \texttt{sensor_msgs::PointCloud2} is an efficient data structure (good for transport) but not so user friendly. It is also ROS aware.
• \texttt{pcl::PointCloud} is an easy to use data structure that is used by the PCL. It is independent of ROS.
• So how were we able to subscribe to the PCL datatype?
Enter `pcl_ros`
Using the *pcl_ros* bridge

- To use the bridge, include the *pcl_ros* header
  - `#include <pcl_ros/point_cloud.h>`
- If you are working independent of ROS, the following header is sufficient
  - `#include <pcl/point_cloud.h>`
- OK, so this was using *pcl_ros* while subscribing. Does it work for publishing as well?
  - Required for programming assignment 3
Height-Difference Map

• Height-difference maps identify vertical surfaces in the environment.
  ◦ Without the need for computationally intensive algorithms for 3D, real-time modeling
• At each cycle (i.e. every complete set of 360° data) we create a 2D (x, y) grid map from the 3D point cloud.
  ◦ Record the maximum and minimum z (vertical) values seen in each grid cell.
• Next a simulated lidar scan is produced from the 2D grid.
  ◦ Cast rays from the sensor origin, an obstacle is detected whenever the difference between the max and min z values is above a threshold
• The result is a 360° 2D simulated lidar scan, which looks like and can be further processed in a manner similar to the data outputed by SICK lidar devices
Algorithm 1 Height-Difference Map

1: for all \( p \in \text{pointcloud} \) do
2: \( x, y \leftarrow \text{grid.index}(p.x, p.y) \)
3: \( \text{grid}[x][y].z_{\text{min}} \leftarrow \min(p.z, \text{grid}[x][y].z_{\text{min}}) \)
4: \( \text{grid}[x][y].z_{\text{max}} \leftarrow \max(p.z, \text{grid}[x][y].z_{\text{max}}) \)
5: end for
6: for all \( \text{angle} \in (0, 360) \) do
7: \( \text{lidar.scan}[\text{angle}] = \text{MAX.DISTANCE} \)
8: for all \( \text{cell} \in \text{raytrace(\text{angle})} \) do
9: if \( (\text{cell.z}_{\max} - \text{cell.z}_{\min}) > \text{THRESHOLD} \) then
10: \( \text{lidar.scan}[\text{angle}] = \text{cell.distance} \)
11: break
12: end if
13: end for
14: end for
Height-Difference Map

(z_{max} - z_{min}) > \text{THRESHOLD}

Uninterrupted simulated lidar scan

(0,0,0)
Height-Difference Map
The *rosbag* tool

- As some of you may have realized by now, writing a lot of code at the car is not feasible.
- Lucky for us, we have a way of collecting data easily.
- You can record topics using the rosbag record command.
- Examples:
  - `rosbag help record`
  - `rosbag record /velodyne_packets`
  - `rosbag record /velodyne_packets /odom`
  - `rosbag record -a`
- Recording a bag will output a `.bag` file
Playing back recorded data

- rosbag provides a playback tool
- Examples:
  - `rosbag play <bag_file>`
  - `rosbag play <bag_file> /velodyne_packets`
  - `rosbag play *_05.bag`
  - `rosbag play *_05.bag -l --clock` (Recommended)
- This is *almost* as good as having all these messages being produced live
Putting it all together

• To run the entire chain, follow these steps:
  ○ Get a bag file
  ○ Start `roscore`
  ○ Play the bag file
    ▪ `rosbag play -l --clock velodyne_2012-02-15-15-06-05.bag`
  ○ Start `cloud_node` (for bags before 2/22)
    ▪ `rosrun velodyne_pointcloud cloud_node velodyne_packets:=velodyne/packets`
  ○ (For bags after 2/22)
    ▪ `rosrun velodyne_pointcloud cloud_node`
  ○ Visualizing in `rviz`
    ▪ `rosrun rviz rviz`
  ○ Build the subscriber
    ▪ `rosmake velodyne_example`
  ○ Running the subscriber
    ▪ `rosrun velodyne_example subscriber`
rviz - The ROS visualization tool

- We briefly saw rviz on Monday.
- An excellent tutorial for navigating rviz is available here
- I will run a live demo of rviz in class today.
- There are a couple of things (like tf) that we'll explain later in the semester.
  - For the time being, set the following values when playing back a bag file:
    - Fixed frame: /velodyne
    - Target frame: <Fixed frame>