Exercise 6: Kinect and PCL

Development of Intelligent Systems

2021

In the scope of this assignment you will learn how to process and use the acquired depth information from the Kinect sensor using the Point Cloud Library.

It is expected that the following assignments are solved in C++, as this is the easiest and fastest way to use the PCL library and process large point clouds. As ROS nodes are meant for long-term continuous operation, be careful to release the memory that you allocate manually or use automatic techniques (e.g. smart pointers). The nodes in the given package mostly consist of implementations of the tutorials on the PCL page in ROS.

1 Using PCL to manipulate a point cloud

The PCL library can perform many operations on point clouds, but to do it, the clouds have to be first retrieved by your node. Note that recent versions of ROS have changed the ways some frequently used messages are handled. For PCL you can now directly operate using PCL structures which are automatically converted to messages. So instead of using classes from the `sensor_msgs` package you can write your most simple node that simply forwards point clouds like this:

```
#include <ros/ros.h>
#include <pcl_conversions/pcl_conversions.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>

ros::Publisher pub;

void callback(const pcl::PCLPointCloud2ConstPtr& cloud_blob) {
  pub.publish(cloud_blob);
}

int main(int argc, char** argv) {
  ros::init (argc, argv, "forwarder");
  ros::NodeHandle nh;
  ros::Subscriber sub = nh.subscribe ("input", 1, callback);
  pub = nh.advertise<pcl::PCLPointCloud2>("output", 1);
  ros::spin ();
}
```

The full source code for this example is available in the package for this exercise under the name `forwarder`.

Next, we will look at a node that takes a full point cloud from the Kinect and makes it more sparse. Sparser point clouds have fewer points and are therefore easier to process (so this operation is suitable as a pre-processing step in computationally intensive operations, but it is not always practical to perform it in a separate node as in this example). A full

example is available in the package for this exercise under the name `voxelgrid` and only differs from the previous example in the callback function and an extra include.

```
#include <pcl/filters/voxel_grid.h>

...

void callback(const pcl::PCLPointCloud2ConstPtr& cloud_blob) {

  pcl::PCLPointCloud2 cloud_filtered;

  pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
  sor.setInputCloud (cloud_blob);
  sor.setLeafSize (0.1, 0.1, 0.1);
  sor.filter (cloud_filtered);

  pub.publish(cloud_filtered);
}
```

2 Determining the ground plane

One of the most useful components of the PCL library are the algorithms for fitting 3D geometric primitives to a point cloud. For an in-door mobile robot an almost ever-present feature of the environment is the ground plane. By detecting the plane we can remove it and process the remaining point cloud in search for objects that stick out from the ground.

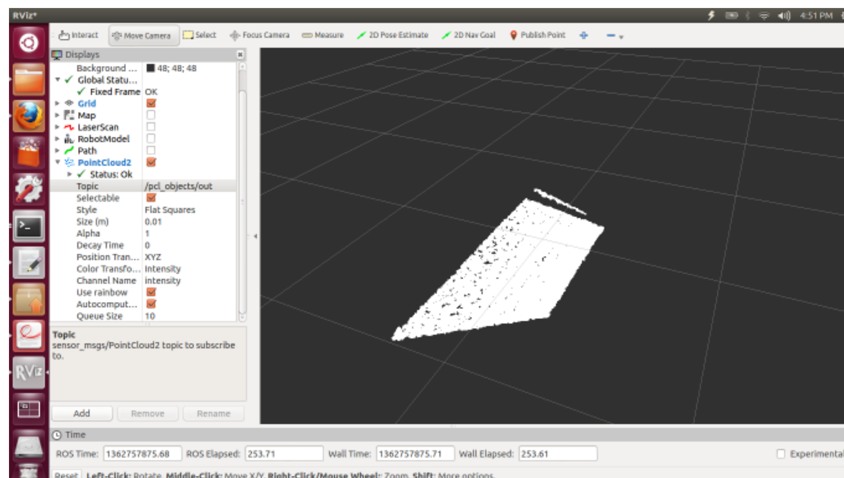
The example available in the package under the name `find_plane` is based on the previous examples and contains a lot more code that demonstrates the main components of the PCL library (e.g. how each algorithm has input and output point clouds).

Note that this example does not always find the ground-plane, but it finds a plane this is a dominant plane in the given point cloud, in most cases this will be floor or wall.

```
#include <pcl/ModelCoefficients.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/filters/extract_indices.h>

...

  pcl::PCLPointCloud2::Ptr cloud_filtered_blob (new pcl::PCLPointCloud2);
  pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_filtered (new pcl::PointCloud<pcl::↩
      PointXYZRGB>);
  pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_outliers (new pcl::PointCloud<pcl::↩
      PointXYZRGB>);

  // Create the filtering object: downsample the dataset using a leaf size of 1cm
  pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
  sor.setInputCloud (cloud_blob);
  sor.setLeafSize (0.01f, 0.01f, 0.01f);
  sor.filter (*cloud_filtered_blob);

  // Convert to the templated PointCloud
  pcl::fromPCLPointCloud2 (*cloud_filtered_blob, *cloud_filtered);

  pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients ());
  pcl::PointIndices::Ptr inliers (new pcl::PointIndices ());

  // Create the segmentation object
  pcl::SACSegmentation<pcl::PointXYZRGB> seg;
```

```
    // Optional
    seg.setOptimizeCoefficients (true);

    // Mandatory
    seg.setModelType (pcl::SACMODEL_PLANE);
    seg.setMethodType (pcl::SAC_RANSAC);
    seg.setMaxIterations (1000);
    seg.setDistanceThreshold (0.01);
    seg.setInputCloud (cloud_filtered);

    seg.segment(*inliers, *coefficients);

    if (inliers->indices.size () == 0) return;

    pcl::ExtractIndices<pcl::PointXYZRGB> extract;

    extract.setInputCloud(cloud_filtered);
    extract.setIndices(inliers);
    extract.setNegative(false);
    extract.filter(*cloud_outliers);

    // Publish the found plane
    pcl::PCLPointCloud2 outcloud;
    pcl::toPCLPointCloud2 (*cloud_outliers, outcloud);

    pub.publish (outcloud);
```

Publish new point cloud on a new topic and visualize it in Rviz.



3 Detecting a cylinder

As part of the final competition in this course you will need to detect the cylinders in the polygon. One way to do this is using the PCL library. In the package for this exercise you have one example how to do this in the `cylinder_segmentation` node. Explore the code. Can you think of ideas to speed up or improve the detections?

4 Homework: 3D object detection

Either using the PCL library or in another way, create a node which detects cylinders and 3D rings. The robot should drive around the polygon and publish a marker with the location of each detected object.