Programming exercise 4: PCL

Credit:http://pointclouds.org/

1.Read and write PointCloud

Write and read a PointCloud

First follow this tutorial http://pointclouds.org/documentation/tutorials/writing_pcd.php#writing-pcd to create a simple PointCloud and write it into a *.pcd* file. After creating the PointCloud, take a look at the data using *cat* or text editor and you should see the output:

```
# .PCD v0.7 - Point Cloud Data file format
VERSION 0.7
FIELDS x y z
SIZE 4 4 4
TYPE F F F
COUNT 1 1 1
WIDTH 5
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS 5
DATA ascii
0.35222197 -0.15188313 -0.10639524
-0.3974061 -0.47310591 0.29260206
-0.73189831 0.66710472 0.44130373
-0.73476553 0.85458088 -0.036173344
-0.46070004 -0.2774682 -0.91676188
```

Find out the explanation of the header here http://pointclouds.org/documentation/tutorials/pcd_file_format.php#pcd-file-format.

Let's move on to the tutorial on how to read a PointCloud http://pointclouds.org/documentation/tutorials/reading_pcd.php#reading-pcd, and work with the PointCloud you just created.

View a PointCloud

Now let's try to create a more interesting PointCloud and take a look at it. Create source code named *pcd_ellipsoid.cpp* and fill the file with following code. Pay attention to the *pcl::visualization::CloudViewer* part for visualizing the cloud. More details can be found here http://pointclouds.org/documentation/tutorials/cloud_viewer.php#cloud-viewer:

```
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/visualization/cloud_viewer.h>
#include <cmath>
#include <iostream>

int main(int argc, char* argv[])
{
    const float pi = 3.14159265358979f;
    const float two_pi = 2*pi;
```

```
    // The ellipsoid center at (x0,y0,z0)
    const float x0 = 0.0f, y0 = 0.0f, z0 = 0.0f;

    // ellipsoid semi-axes lengths (for sphere, a=b=c)
    const float a = 1.0f, b = 1.0f, c = 1.0f;

    // theta: azimuthal coordinate, 0 to 2*pi
    // psi: polar coordinate, 0 to pi
    const float azimuthal_resolution = pi / 60.0f;
    const float polar_resolution = pi / 60.0f;

    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pcl::PointXYZ>);

    for (float theta=0.0f; theta<=two_pi; theta+=azimuthal_resolution)
    {
      for (float psi=0.0f; psi<=pi; psi+=polar_resolution)
      {
         const float x=x0+a*std::cos(theta)*std::sin(psi);
         const float y=y0+b*std::sin(theta)*std::sin(psi);
         const float z=z0+c*std::cos(psi);
         cloud->push_back(pcl::PointXYZ(x,y,z));
      }
    }

    std::cout << "Saving cloud..." << std::endl;
    pcl::io::savePCDFileASCII ("ellipsoid_pcd.pcd", *cloud);
    std::cout << "Saved " << cloud->points.size () << " data points to ellipsoid_pcd.
        ↪ pcd." << std::endl;

    // See the output
    pcl::visualization::CloudViewer viewer("Simple Cloud Viewer");
    viewer.showCloud(cloud);
    while (!viewer.wasStopped())
    { //no-op until viewer stopped
    }

    return 0;
}
```

You can interact with the *CloudViewer* with the mouse for zooming and rotating.

2. Surface Normals

Surface normals are important properties of a geometric surface and it's frequently used in PCL. Here's an explanation and example `http://pointclouds.org/documentation/tutorials/normal_estimation.php#normal-estimation`. Load the previous ellipsoid cloud and try to compute as well as visualize the surface normals with a more power viewer *PCLVisualizer* `http://pointclouds.org/documentation/tutorials/pcl_visualizer.php`. A snippet code cloud look like this:

```
// ------------------------------------------------------------------
// -----Calculate surface normals with a search radius of 0.05-----
// ------------------------------------------------------------------
pcl::NormalEstimation<pcl::PointXYZRGB, pcl::Normal> ne;
ne.setInputCloud (point_cloud_ptr);
pcl::search::KdTree<pcl::PointXYZRGB>::Ptr tree (new pcl::search::KdTree<pcl::
    ↪ PointXYZRGB> ());
ne.setSearchMethod (tree);
pcl::PointCloud<pcl::Normal>::Ptr cloud_normals1 (new pcl::PointCloud<pcl::Normal>);
ne.setRadiusSearch (0.05);
```

```
ne.compute (*cloud_normals1);
...
boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer (new pcl::visualization::
    ↪ PCLVisualizer ("3D Viewer"));
...
viewer->addPointCloudNormals<pcl::PointXYZRGB, pcl::Normal> (point_cloud_ptr,
    ↪ cloud_normals1, 10, 0.05, "normals");
...
```

3.Filtering a PointCloud

Similar to image data, PointCloud data also contains outlier and noise that we want to remove. *pcl_filters* library provide mechanisms for 3D point cloud data filtering.

PassThrough filter

PassThrough filter performs a simple filtering along a specified dimension. Let's move to the tutorial on PassThrough filter `http://pointclouds.org/documentation/tutorials/passthrough.php#passthrough`. First follow the tutorial example to have a good understanding on *PassThrough* object, then load the previous generated *ellipsoid_pcd.pcd* and try to generate a ring, for example:

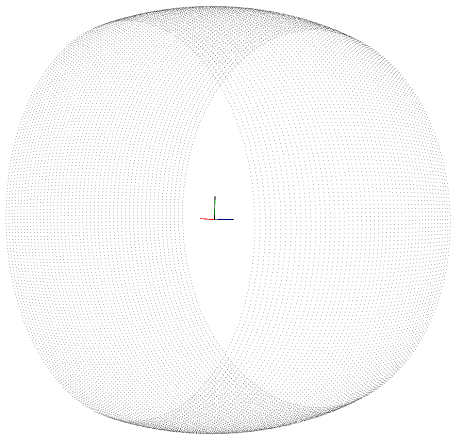

Figure 1: A ring.

Downsampling

A set of PointCloud data can be very large and expensive to process, especially for mobile systems with a on-board computer. In order to reduce computational load and also preserve the data information (color, structure, etc.), downsampling is very often implemented as the very first step of cloud processing.

Follow the tutorial `http://pointclouds.org/documentation/tutorials/voxel_grid.php#voxelgrid` and learn about the *VoxelGrid* filtering object. You can exper-

iment with both the provided *table_scene_lms400.pcd* data and the *ellipsoid_pcd.pcd* data we created before. Try to visualize the downsampled cloud.

4.Segmentation

Plane model segmentation

To get a good understanding of the *SACSegmentation* object, follow the tutorial `http://pointclouds.org/documentation/tutorials/planar_segmentation.php#planar-segmentation`.

After testing the example code, use the provided PointCloud data, extract and display the detected plane with help of this tutorial `http://pointclouds.org/documentation/tutorials/extract_indices.php#extract-indices`. (*Hint:extract the indices of points belong to the detected plane, create a new cloud containing those points and display it.*)

Euclidean cluster extraction

Instead of extracting the planar object, the planar inliers can also be removed and the remaining could be our object of interest. Follow the tutorial here `http://pointclouds.org/documentation/tutorials/cluster_extraction.php#cluster-extraction`, try it on the provided PointCloud data and view the resulted clusters.