ILR 04 – Progress Review 3

Shivam Gautam

Team Daedalus

Members: Pranav Maheshwari, Richa Varma, Mohak Bhardwaj, and Dorothy Kirlew

November 13th , 2015

1. Individual Progress

I undertook the following tasks:

1. Implementing the Point Cloud Library for cylinder segmentation
2. Processing Point Cloud Data Obtained by the Kinect
3. Design of Proximity Detection PCB board
4. Design communication data packet for collaboration

1. Implementing the Point Cloud Library (PCL) for Cylinder Segmentation

The PCL library has functionality that allows users to identify 3D shapes like cylinders and cuboids. This allows the team to meet its functional requirement of static obstacle detection.

The program implemented (attached in the 'Code' section) implements a random sample consensus segmentation for cylinders. The code loads a point cloud data file and performs the following tasks-

- Estimate surface normal at each PCL
- Filtering Data points greater than a certain range
- Planar model segmentation
- Cylindrical model segmentation

The key functions in this program are as follows-

- **seg.setModelType (pcl::SACMODEL_CYLINDER)** -This sets the type of figure being segmented as a cylinder.
- **seg.setMethodType (pcl::SAC_RANSAC)** – This sets the sampling consensus method as RANSAC (Random Sampling Consensus). RANSAC is a robust outlier rejection technique and is widely used due to its ease of implementation.
- **seg.setNormalDistanceWeight (0.1) -** This sets the weight to the influence surface normal have to 0.1.
- **seg.setMaxIterations (10000); -** This parameter sets the number of iterations in the RANSAC algorithm to 10000.
- **seg.setDistanceThreshold (0.05) –** The distance threshold for each point to be classified as an inlier is set as 5 cm.
- **seg.setRadiusLimits (0, 0.1) –** The radius of the cylinder to be detected is set to 10 cm.

The output for the segmented cylindrical point cloud is depicted in figure 1.

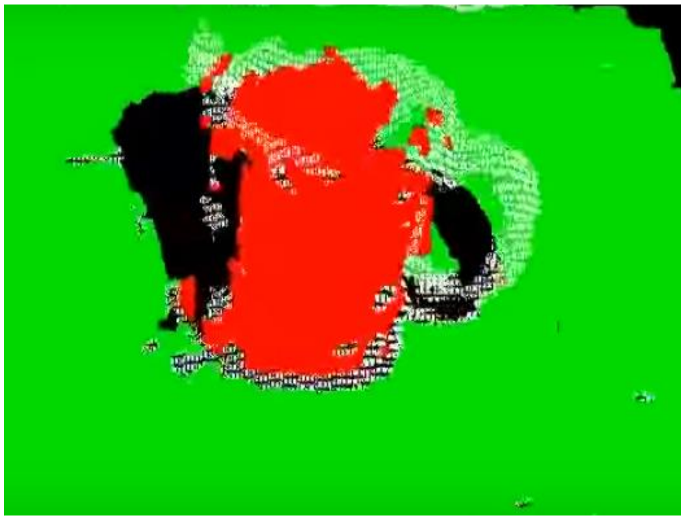
Figure 1- Cylinder segmentation using sample Point Cloud Data

1. Processing Point Cloud Data Obtained by the Kinect

During the previous weeks, we had been able to get point cloud data from the Kinect. Building on that work, we needed to process the data obtained by the Kinect to segment a cylinder. The key characteristics of the data obtained from the Kinect were as follows-

- The point cloud data was dense and therefore processing it was computationally intensive.
- The point cloud data contained points that were irrelevant to us (eg. Point clouds of walls and faraway objects).
- The point cloud data was noisy with spurious detections.

This is aptly depicted in Figure 2 which shows that in addition to the object in front of the Kinect, the shelves and the walls are also detected in the scene.

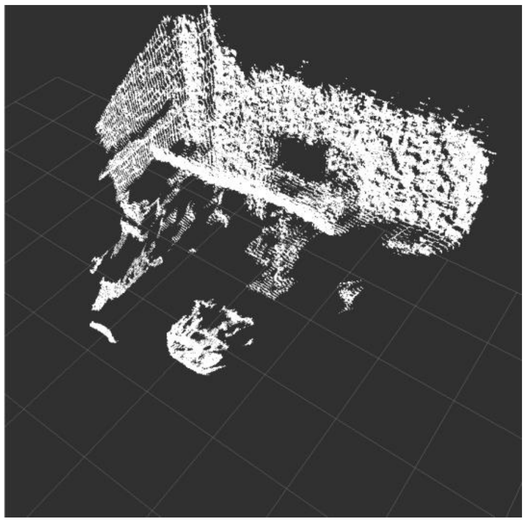
Figure 2- Unfiltered and Dense Point Cloud Data

To mitigate these factors, we decided to write a code using the PCL library to filter and down-sample data. This was done as follows-

- **Down-sampling the PointCloud using a VoxelGrid filter**

The VoxelGrid filter reduces the number of points in the point cloud using a voxelized grid approach. This involves approximating points with their centroids. The results of the down-sampling are depicted in figure 3. The code for this down-sampling is mentioned in the section named 'Code'.

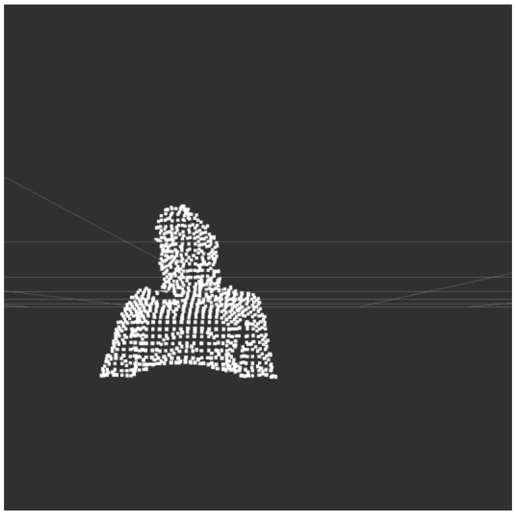

Figure 3- Down-Sampled Data

- **Filtering irrelevant data and noise reduction**

The pass-through filter was used to remove the points in the point cloud that were irrelevant to obstacle detection. Ideally, we intend to identify any obstacles that are within a range of 0.5 m to 1m from the platform. Therefore we used the pass-through filter to remove any points in the point cloud beyond a distance of 1.5m. The results for this are depicted in figure 4 and figure 5 and the code for the same is mentioned in the section named 'Code'.

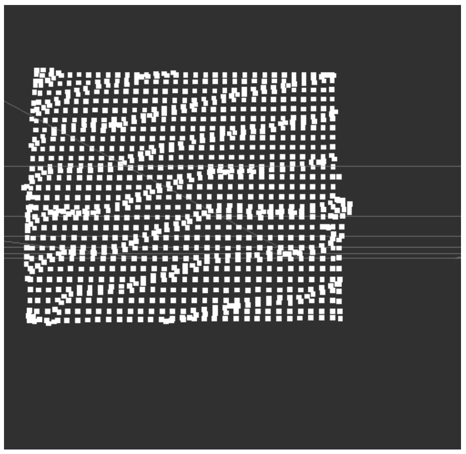

Figure 4- Down-sampled Point Cloud of a board

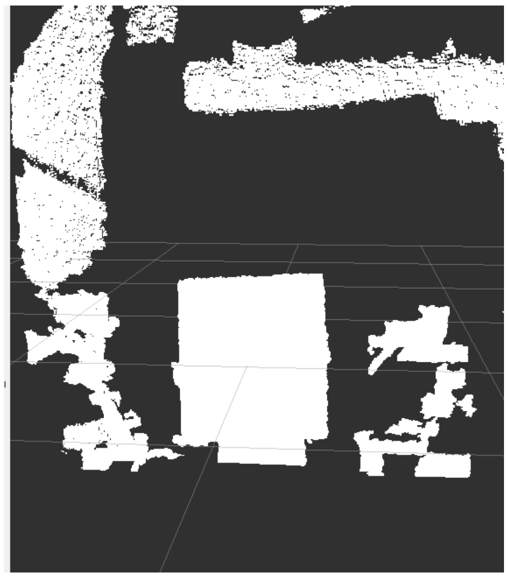

Figure 5- Dense Point Cloud of a board

Figure 6- Point Cloud of a person holding a cup

2. Design of Proximity Detection PCB board

Initially, the team decided to go ahead with a generic power distribution board for the mobile platform. This seemed reasonable as a platform had not been finalized then. After the team decided to go ahead with the mobile platform, the need for a power distribution board did not seem to arise as the platform came with its own power distribution board.

The team decided to design a self-contained proximity detection unit which would house proximity sensors and a microcontroller. If the sensors detect some obstacle within a range of 20cm, it would raise an interrupt and inform the ROS Master about an obstacle.

The PCB contains an Arduino Nano board that snaps onto the board. The board also has a voltage regulator, power-indicator LEDs and connectors for the proximity sensors.

The old and the new PCB board are depicted in figures 7 and 8.

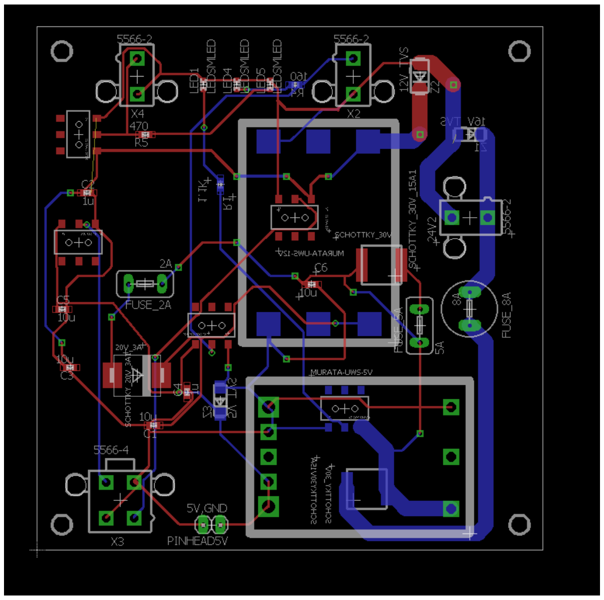
Figure 6 - Power Distribution Board

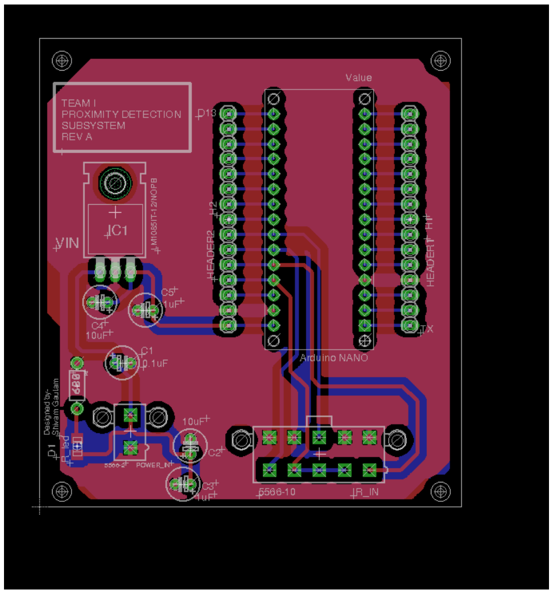
Figure 7- Proximity Detection Subsystem

3. Design communication data packet for collaboration

For the communication system, we needed to come up with a suitable packet of information that would be transmitted across all cars. This packet needed to contain the following with respect to the transmitting vehicle-

1. Unique ID
2. Current Status
3. Current Position
4. Destination
5. Velocity
6. Obstacles in the parking lot
7. Optimum Spot for the next car
8. Occupancy Map of the parking lot

The occupancy map transmitted needs to incorporate the following information-

1. Free spots in the Lot
2. Current Position of transmitting vehicle
3. Destination of transmitting vehicle
4. Obstacles in the parking lot
5. Optimum Spot for the next car
6. Occupied Spots in the parking lot
7. Free lanes where traversal is allowed

Other information like the Unique ID and the velocity would be present in the packet separately. I intend to formalize this in terms of actual integer values and test this packet size by transmitting it across XBees.

2. Challenges

One of the challenges I faced during working with the Kinect was the fact that the OpenNI library could not detect the Kinect. After spending numerous hours trying to solve this problem, the solution that worked was removing all libraries and drivers related to the Kinect and installing them again. This can be explained by the fact that other conflicting drivers installed on the machine interfere with the Kinect's PrimeSense drivers while OpenNI attempts to access the Kinect. The solution seemed trivial but it was essential to not have conflicting libraries (OpenNI 2, libfreenect etc.) present while trying to work with OpenNI.

The development of the PCB board was a bit challenging as I had to redesign the PCB to work as a proximity detection unit from an initially conceived power distribution system. Also, generation of the GERBER files consumed some time as the Advanced Circuits tool did not detect my '.drd' drill file. This was rectified by selecting the right format- 'Excellon' while generating the drill file.

3. Teamwork

The team made good progress this week with respect to the mobile app, obstacle detection and the communication subsystem. I worked with Mohak on processing the point-cloud data from the Kinect. Working with Richa on the PCB development was a fruitful experience as this helped in coming up with a contingency plan when our initial design had to change. I also worked with Dorothy on the actual implementation of the Arduino as a ROS node for proximity detection.

Pranav and Richa set up the XBee adapters that will be used to communicate between mobile platforms. They also started working on a defining a serial protocol and conducted preliminary testing of the XBee adapters. Dorothy worked on setting up serial communication from an Android phone to a laptop over Bluetooth and made great progress.

4. Plans

The team would be working on accomplishing the following tasks for next week-

1. Mobile Platform-
    a. Assemble the platform and test locomotion through an SBC (Mohak and Pranav)
    b. Integrate electronic components with the platform (Shivam and Richa)
    c. Mount and integrate the IR sensors and Arduino Nano with platform (Shivam and Dorothy)
2. Communication System-
    a. Formulate a formal packet size for data transmission and conduct testing. (Shivam and Richa)
    b. Make two SBCs communicate and share data using DigiMesh XBee adapters (Richa and Pranav)
3. Obstacle Detection-
    a. Detect cylindrical obstacles using data from Kinect (Mohak and Shivam)
4. Android App/Bluetooth Communication Subsystem
    a. Work on bidirectional serial communication via Bluetooth between App and laptop (Dorothy and Mohak)

5. CODE

Cylinder Segmentation

```
#include <pcl/ModelCoefficients.h>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/filters/extract_indices.h>
#include <pcl/filters/passthrough.h>
#include <pcl/features/normal_3d.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/segmentation/sac_segmentation.h>

typedef pcl::PointXYZ PointT;

int
main (int argc, char** argv)
{
  // All the objects needed
  pcl::PCDReader reader;
  pcl::PassThrough<PointT> pass;
  pcl::NormalEstimation<PointT, pcl::Normal> ne;
  pcl::SACSegmentationFromNormals<PointT, pcl::Normal> seg;
  pcl::PCDWriter writer;
  pcl::ExtractIndices<PointT> extract;
  pcl::ExtractIndices<pcl::Normal> extract_normals;
  pcl::search::KdTree<PointT>::Ptr tree (new pcl::search::KdTree<PointT> ());

  // Datasets
  pcl::PointCloud<PointT>::Ptr cloud (new pcl::PointCloud<PointT>);
  pcl::PointCloud<PointT>::Ptr cloud_filtered (new pcl::PointCloud<PointT>);
  pcl::PointCloud<pcl::Normal>::Ptr cloud_normals (new pcl::PointCloud<pcl::Normal>);
  pcl::PointCloud<PointT>::Ptr cloud_filtered2 (new pcl::PointCloud<PointT>);
  pcl::PointCloud<pcl::Normal>::Ptr cloud_normals2 (new pcl::PointCloud<pcl::Normal>);
  pcl::ModelCoefficients::Ptr coefficients_plane (new pcl::ModelCoefficients), coefficients_cylinder (new
pcl::ModelCoefficients);
  pcl::PointIndices::Ptr inliers_plane (new pcl::PointIndices), inliers_cylinder (new pcl::PointIndices);

  // Read in the cloud data
  reader.read ("table_scene_mug_stereo_textured.pcd", *cloud);
  std::cerr << "PointCloud has: " << cloud->points.size () << " data points." << std::endl;

  // Build a passthrough filter to remove spurious NaNs
  pass.setInputCloud (cloud);
  pass.setFilterFieldName ("z");
  pass.setFilterLimits (0, 1.5);
  pass.filter (*cloud_filtered);
  std::cerr << "PointCloud after filtering has: " << cloud_filtered->points.size () << " data points." << std::endl;

  // Estimate point normals
  ne.setSearchMethod (tree);
  ne.setInputCloud (cloud_filtered);
  ne.setKSearch (50);
  ne.compute (*cloud_normals);

  // Create the segmentation object for the planar model and set all the parameters
  seg.setOptimizeCoefficients (true);
```

```
 seg.setModelType (pcl::SACMODEL_NORMAL_PLANE);
 seg.setNormalDistanceWeight (0.1);
 seg.setMethodType (pcl::SAC_RANSAC);
 seg.setMaxIterations (100);
 seg.setDistanceThreshold (0.03);
 seg.setInputCloud (cloud_filtered);
 seg.setInputNormals (cloud_normals);
 // Obtain the plane inliers and coefficients
 seg.segment (*inliers_plane, *coefficients_plane);
 std::cerr << "Plane coefficients: " << *coefficients_plane << std::endl;

 // Extract the planar inliers from the input cloud
 extract.setInputCloud (cloud_filtered);
 extract.setIndices (inliers_plane);
 extract.setNegative (false);

 // Write the planar inliers to disk
 pcl::PointCloud<PointT>::Ptr cloud_plane (new pcl::PointCloud<PointT> ());
 extract.filter (*cloud_plane);
 std::cerr << "PointCloud representing the planar component: " << cloud_plane->points.size () << " data points." <<
std::endl;
 writer.write ("table_scene_mug_stereo_textured_plane.pcd", *cloud_plane, false);

 // Remove the planar inliers, extract the rest
 extract.setNegative (true);
 extract.filter (*cloud_filtered2);
 extract_normals.setNegative (true);
 extract_normals.setInputCloud (cloud_normals);
 extract_normals.setIndices (inliers_plane);
 extract_normals.filter (*cloud_normals2);

 // Create the segmentation object for cylinder segmentation and set all the parameters
 seg.setOptimizeCoefficients (true);
 seg.setModelType (pcl::SACMODEL_CYLINDER);
 seg.setMethodType (pcl::SAC_RANSAC);
 seg.setNormalDistanceWeight (0.1);
 seg.setMaxIterations (10000);
 seg.setDistanceThreshold (0.05);
 seg.setRadiusLimits (0, 0.1);
 seg.setInputCloud (cloud_filtered2);
 seg.setInputNormals (cloud_normals2);

 // Obtain the cylinder inliers and coefficients
 seg.segment (*inliers_cylinder, *coefficients_cylinder);
 std::cerr << "Cylinder coefficients: " << *coefficients_cylinder << std::endl;

 // Write the cylinder inliers to disk
 extract.setInputCloud (cloud_filtered2);
 extract.setIndices (inliers_cylinder);
 extract.setNegative (false);
 pcl::PointCloud<PointT>::Ptr cloud_cylinder (new pcl::PointCloud<PointT> ());
 extract.filter (*cloud_cylinder);
 if (cloud_cylinder->points.empty ())
   std::cerr << "Can't find the cylindrical component." << std::endl;
 else
 {
	  std::cerr << "PointCloud representing the cylindrical component: " << cloud_cylinder->points.size () << " data
points." << std::endl;
	  writer.write ("table_scene_mug_stereo_textured_cylinder.pcd", *cloud_cylinder, false);
 }
```

```
  return (0);
}
```

Point Cloud Filtering

```
#include <ros/ros.h>
// PCL specific includes
#include <sensor_msgs/PointCloud2.h>
#include <pcl_conversions/pcl_conversions.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl/ModelCoefficients.h>
#include <pcl/io/pcd_io.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/filters/passthrough.h>
#include <boost/foreach.hpp>

typedef pcl::PointXYZ PointT;


ros::Publisher pub;

void
cloud_cb (const sensor_msgs::PointCloud2ConstPtr& input)
{
        pcl::PCLPointCloud2::Ptr cloud (new pcl::PCLPointCloud2 ());

        pcl::PCLPointCloud2::Ptr cloud_filtered (new pcl::PCLPointCloud2 ());

        pcl::PCLPointCloud2::Ptr cloud_filtered2 (new pcl::PCLPointCloud2);

        //pcl::PCLPointCloud2::Ptr cloud_noiseFiltered (new pcl::PCLPointCloud2 ());

        pcl_conversions::toPCL(*input, *cloud); // Create a container for the data.


  // Voxel Grid Downsampling

  pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
  sor.setInputCloud (cloud);
  sor.setLeafSize (0.02f, 0.02f, 0.02f);
  sor.filter (*cloud_filtered);

  //~ sensor_msgs::PointCloud2 output;
  //~ output = *input;

  //pcl_conversions::moveToROSMsg(sensor_msgs::PointCloud2 &cloud_filtered, sensor_msgs::PointCloud2 &output)


 pcl::PassThrough< pcl::PCLPointCloud2 > pass;
 pass.setInputCloud(cloud_filtered);
 pass.setFilterFieldName ("z");
 pass.setFilterLimits (0, 1.0);
 //pass.setFilterLimitsNegative (true);
 pass.filter(*cloud_filtered2);



  //~ pcl::PassThrough<pcl::PointXYZ> pass;
  //~ pass.setInputCloud(cloud2);
  //~ pass.setFilterFieldName ("z");
  //~ pass.setFilterLimits (0.0, 1.0);
  //~ //pass.setFilterLimitsNegative (true);
  //~ pass.filter(cloud2);

  sensor_msgs::PointCloud2 output;
  pcl_conversions::fromPCL(*cloud_filtered2, output);

  //~
  // Publish the data.
  pub.publish (output);
}

int
main (int argc, char** argv)
{
  // Initialize ROS
  ros::init (argc, argv, "my_pcl_tutorial");
  ros::NodeHandle nh;

  // Create a ROS subscriber for the input point cloud
  ros::Subscriber sub = nh.subscribe ("input", 1, cloud_cb);

  // Create a ROS publisher for the output point cloud
  pub = nh.advertise<sensor_msgs::PointCloud2> ("output", 1);

  // Spin
  ros::spin ();
}
```

6. References

- http://www.xaxxon.com/xaxxon/malg
- http://pointclouds.org/documentation/tutorials/cylinder_segmentation.php
- http://pointclouds.org/documentation/tutorials/voxel_grid.php
- http://pointclouds.org/documentation/tutorials/passthrough.php