Using ROS to Identify Duplo Bricks of the Same Size

Hieu Nguyen

CSCI-547 Final Project

"Sensing and Planning in Robotics"

Fall 2011

I. Project Description

In the field of robotics, determining the relative sizes of objects is not an easy task. A robot cannot simply “eyeball” a basketball and a golf ball to conclude that one is bigger than the other. Rather, it must make intelligent decisions based on calculated measurements. Using its noisy sensors, the robot must first identify a single object by filtering it out from non-objects in the scene. It then performs a reconstruction to estimate the object's dimensions. The robot must perform this procedure for every object in the scene and run a comparison search to determine relative object sizes. This size information can then be used in recognition, classification, and manipulation applications.

The “Identify Duplos of the Same Size” project is a sensing and perception task that involves using point cloud information to estimate and classify the sizes of Duplo bricks, which are construction toys designed for small children. The assortment of Duplo bricks used in this project consists of three different sizes and four different colors (red, green, blue, and yellow). Data is obtained using a Microsoft Kinect sensor, and the resulting point cloud is processed using ROS (www.ros.org) and PCL (www.pointclouds.org) algorithms. With these tools, one can estimate the sizes of Duplo bricks in a given scene, and determine if the set contains bricks of the same size.

II. Algorithm

A point cloud from the Kinect sensor contains a collection of data points that represent the sensed three-dimensional environment. Each data point contains a set of vertices in the 3D space (represented with XYZ Cartesian coordinates) and a color value (represented in RGB). The general approach for identifying Duplo bricks of the same size is to first segment the bricks from the point cloud, then estimate the size of each Duplo brick, and finally search the group for bricks of similar sizes.

1. Segmentation

Because the Duplo bricks have fixed color values and will be scattered across a white tabletop surface, the bricks can be segmented using the point cloud's position and color information. Given an input point cloud with a table and objects on top of it, we want to segment the individual object point clusters lying on the plane. This can be accomplished by creating a point cloud that has a number of spatially isolated regions to break the cloud down into its constituent parts for independent processing. Each distinct cluster will then represent a single Duplo brick in the scene.

The raw point cloud from the Kinect sensor contains over 300,000 data points. Thus, to improve performance, a pcl::VoxelGrid filter is used to downsample the input cloud. The voxel grid filter assembles a local 3D voxel grid over the input point cloud data. All data points within each voxel are then approximated with their centroid to represent the underlying surface, effectively compressing the point cloud. Moving along in the Duplo segmentation pipeline, the downsampled cloud undergoes a planar segmentation to identify the tabletop surface. The pcl::SACSegmentation algorithm uses the RANSAC method to find all points within a point cloud that support a plane model. This returns the four planar coefficients and all of the inlying points on the largest planar component in the cloud. Using these inlier point indices, pcl::ExtractIndices can project the inliers onto the plane model and create a new point cloud, which represents the objects on the table. The resulting point clouds for planar segmentation and inlier extraction can be visualized in Figure 1.

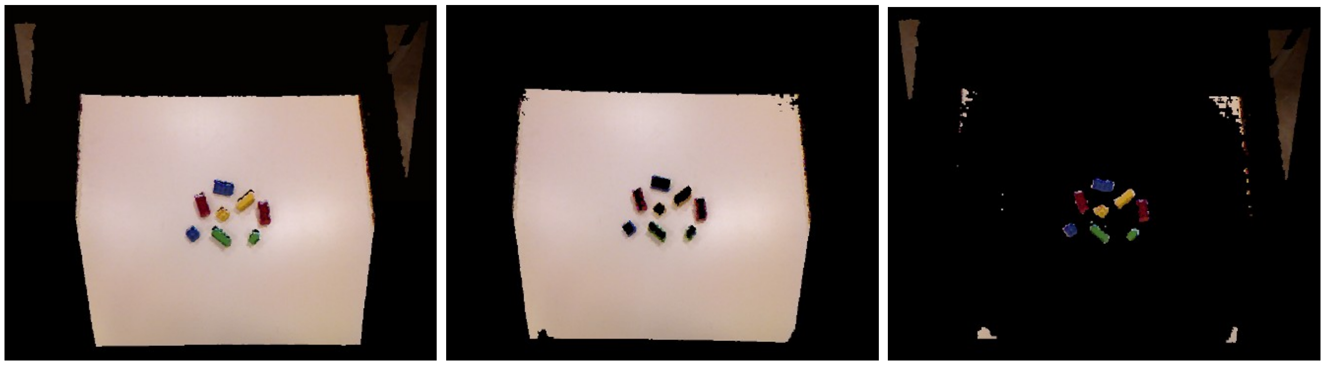


Figure 1. *Duplo bricks on a table. (a) Downsampled cloud, (b) Planar cloud, (c) Objects cloud.*

To remove unwanted points that are located in the plane, but off the table, a pcl::PassThrough filter is used. This filter allows data points to pass through to the output cloud, given the points satisfy a specified range along a particular dimension. Here, we set the filter limits to points within (0.8, 1.0) along the z-axis, which helps to segment a specific region of the table where the Duplo bricks are located.

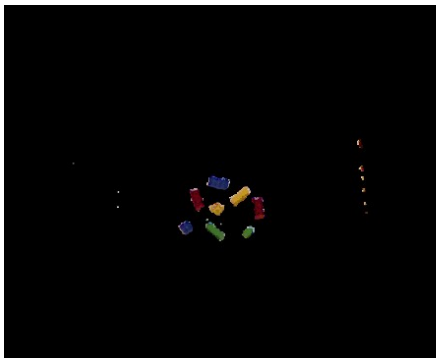


Figure 2. *Output point cloud after a PassThrough filter on the z-axis.*

Next, the filtered point cloud undergoes a color segmentation to separate the red, green, blue, and yellow Duplo bricks from each other. This is done by iterating over all points and pushing those points with specific color characteristics into four new point clouds. Red bricks are characterized by having more R-value than B-value, and twice as much R-value than G-value. Green bricks have a greater G-value than both R- and B-values. Similarly, blue bricks have a greater B-value than both R- and G-values. Yellow bricks have a greater R-value than G-value, and the difference of G-value and B-value is greater than 30.

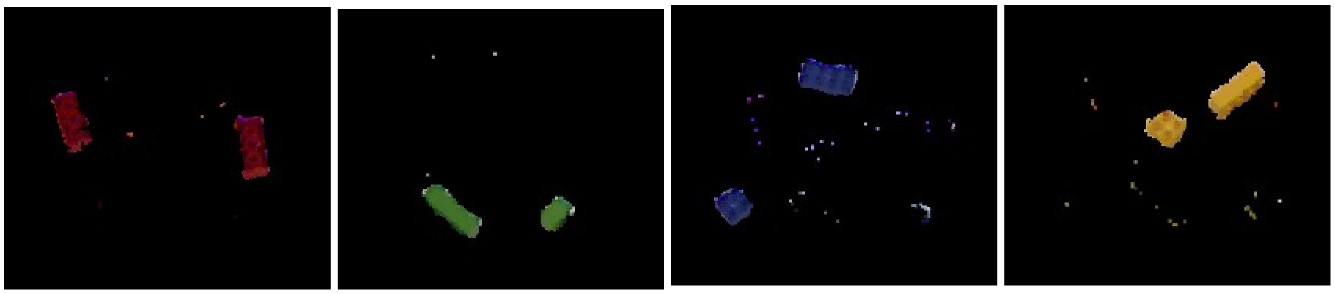


Figure 3. *Color segmentation. (a) Red cloud, (b) Green cloud, (c) Blue cloud, (d) Yellow cloud.*

Each color-segmented point cloud undergoes a clustering algorithm to determine points that belong to the same Duplo brick. The pcl::EuclidianClusterExtraction uses a 3D grid subdivision (octree data structure) for spatial representation, and performs a clustering technique using a nearest neighbors search. A KdTree object is created for the search method of the extraction algorithm to find correspondences between groups of points. The cluster tolerance is set to 0.0075, restricting clusters from having 7.5mm of spacing between points. The minimum cluster size is set to 100 points, which is a function of the downsampling algorithm. Each individual cluster is extracted to its own point cloud and concatenated into a vector of point cloud pointers for further processing.

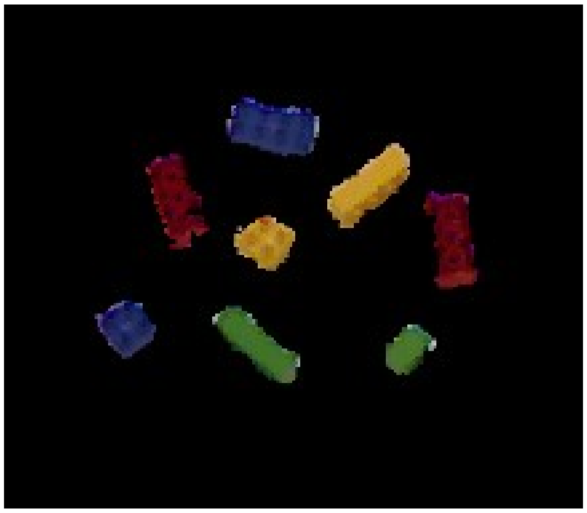


Figure 4. *Concatenated point cloud of all cluster extractions.*

2. Size Estimation

As each cluster is extracted from its color-segmented point cloud, a simple 2D convex hull polygon for the set of points is calculated using a pcl::ConvexHull reconstruction. This convex hull provides an envelope of points representing the minimal convex set. Using these convex hull points, we can estimate the height and width of each cluster. Height is calculated by finding the difference between the maximum and minimum distances from a point in the convex hull reconstruction to the table plane using the planar coefficients obtained in the pcl::SACSegmentation. This height estimation technique was chosen to account for bricks being stacked on one another. The width of each cluster is calculated by finding the maximum distance between two points in the 2D convex hull polygon.

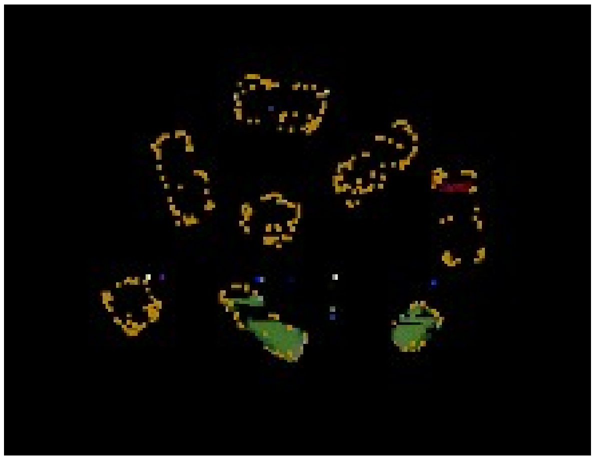


Figure 5. *Concatenated point cloud of all convex hull reconstructions.*

It is possible to get a more accurate size estimation by utilizing pcl::ConvexHull's getTotalArea() function on the convex polygon, which uses the libqhull library to return an area using more complex calculations. Also, running getTotalVolume() on a convex polyhedron reconstruction of the clusters would return volume information. However, this approach would not accurately estimate size because all points of a Duplo brick are not present in the point cloud, causing the 3D reconstruction to be incomplete.

3. Size Classification

Using the height and width calculations, the Duplo bricks are classified by height orientation and brick type. The height classification speculates how the brick is situated on the table. It can determine if the brick is standing flat, standing on its side, or standing long ways (for larger bricks) by checking if the calculated height falls within the actual measured height, ±5mm for error. Similarly, the width classification speculates the brick type. In this project, only three different size Duplo bricks are used: 1x1, 1x2, and 1x4. The width classification thresholds were determined by measuring the minimum 2D brick length and maximum brick length along the 3D diagonal, ±5mm for error. Bricks with calculated heights and widths outside the classification thresholds are ID'ed as "unclassified."

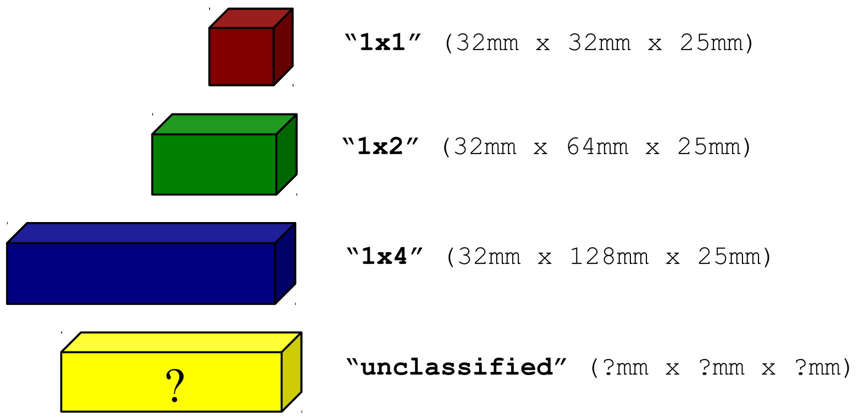

"1x1" (32mm x 32mm x 25mm)
"1x2" (32mm x 64mm x 25mm)
"1x4" (32mm x 128mm x 25mm)
?
"unclassified" (?mm x ?mm x ?mm)

Figure 6. *Duplo brick classification and dimensions.*

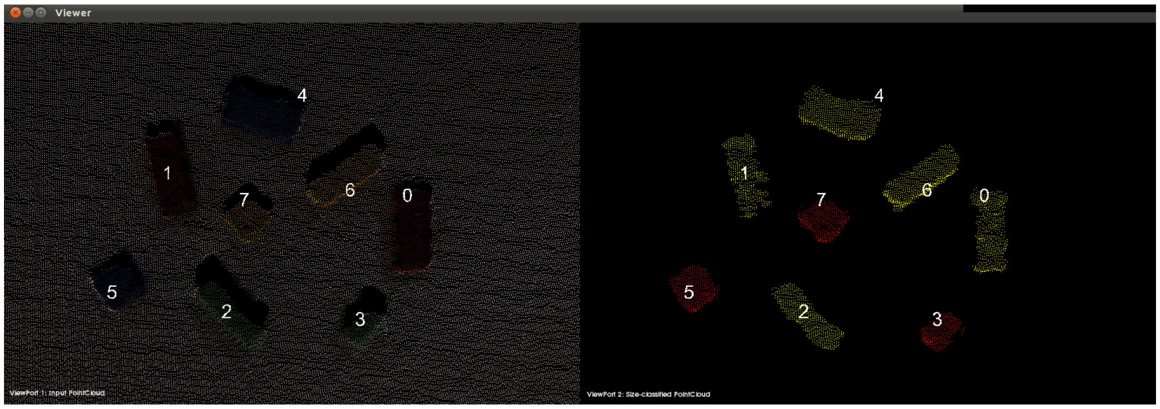
After Duplo brick segmentation, size estimation, and size classification, the ROS node will output information to the terminal window and publish its results to a pcl::PCLVisualizer viewer. The terminal output contains quantitative information on the number of color clusters, the total number of clusters, and the number of bricks classified for each brick type. It also lists the cluster indices for each brick type. The PCLVisualizer uses two viewports to visualize the (downsampled) input point cloud and the size-classified output point cloud in a side-by-side fashion. The clusters are labeled with their cluster indices, and are color-coded based on their brick type; the input point cloud retains its original RGB color values. Brick type 1x1 is colored RED, type 1x2 is colored YELLOW, type 1x4 is colored BLUE, and type "unclassified" is colored WHITE. The PCLVisualizer makes it easy to view (and debug) the results of the overall algorithm.

III. Results

The TA for this class provided example Point Cloud Data (.PCD) files for testing our algorithms. For this project, a couple assumptions are made on the content of each point cloud. First, the Duplo bricks only occur in four possible colors (red, green, blue, yellow) and three possible sizes (1x1, 1x2, 1x4). Also, Duplo bricks of the same color do not come into contact with one another.

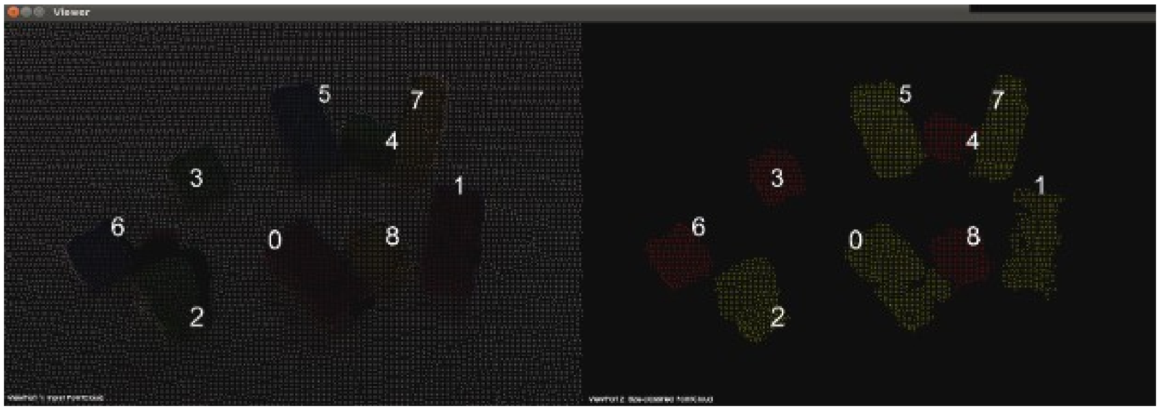
The ROS node subscribes to the "cloud_pcd" topic, which is published by the pcd_to_pointcloud node. The following are the PCLVisualizer viewers and terminal outputs:

group2_1.pcd


Viewer
4
1
7
6
0
5
2
3
ViewPort 1: Input PointCloud
4
1
7
6
0
5
2
3
ViewPort 2: Size-classified PointCloud

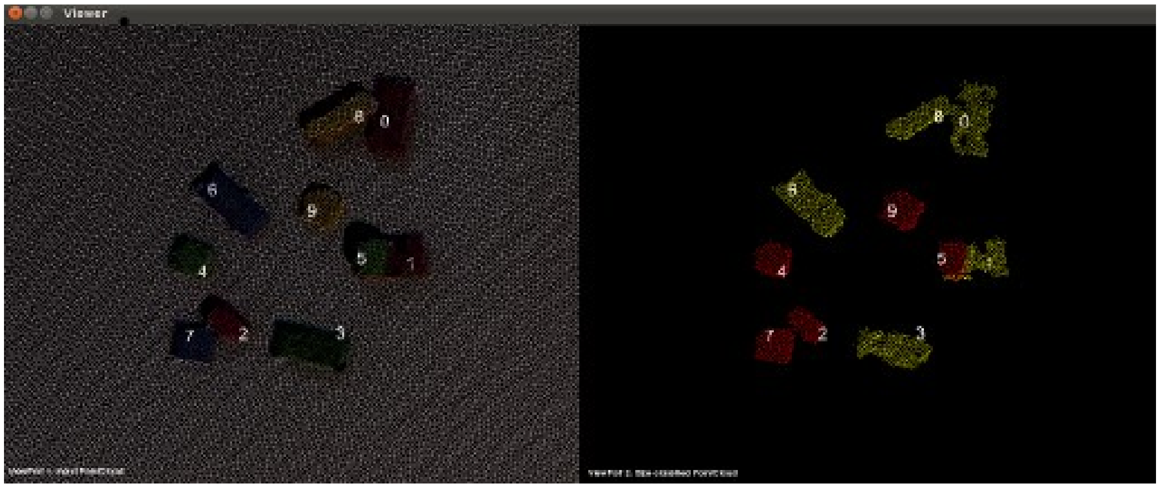
```
Number of RED clusters: 2
Number of GREEN clusters: 2
Number of BLUE clusters: 2
Number of YELLOW clusters: 2
TOTAL number of clusters: 8
There are 3 bricks of size 1x1 (cluster index: 3, 5, 7, )
There are 5 bricks of size 1x2 (cluster index: 0, 1, 2, 4, 6, )
There are 0 bricks of size 1x4
There are 0 unclassified bricks
```

group2_2.pcd



```
Number of RED clusters: 2
Number of GREEN clusters: 3
Number of BLUE clusters: 2
Number of YELLOW clusters: 2
TOTAL number of clusters: 9
There are 4 bricks of size 1x1 (cluster index: 3, 4, 6, 8, )
There are 5 bricks of size 1x2 (cluster index: 0, 1, 2, 5, 7, )
There are 0 bricks of size 1x4
There are 0 unclassified bricks
```

group2_3.pcd



```
Number of RED clusters: 3
Number of GREEN clusters: 3
Number of BLUE clusters: 2
Number of YELLOW clusters: 2
TOTAL number of clusters: 10
There are 5 bricks of size 1x1 (cluster index: 2, 4, 5, 7, 9, )
There are 5 bricks of size 1x2 (cluster index: 0, 1, 3, 6, 8, )
There are 0 bricks of size 1x4
There are 0 unclassified bricks
```

Looking at all three results, the algorithm succeeds in segmenting and classifying the Duplo bricks. The total number of clusters matches the total number of bricks, and there are 0 unclassified bricks in each trial. Further inspection reveals that the algorithm arrived at the correct results, as signified by the color-coded point clouds in the viewer windows.

IV. Testing

Additional testing of the algorithm was performed with a Kinect sensor to obtain point cloud data in real-time. The testbed was set up to mimic the conditions of the example .PCD files provided by the TA. A set of Duplo bricks were placed on a white tabletop surface, with the Kinect viewing the scene from a high angle (to mimic the head-mounted Kinect on the PR2). The ROS node now subscribes to the "/camera/rgb/points" topic that is published by the openni_node. As the ROS node receives PointCloud2 messages from the openni_node, the callback function performs the cloud processing to identify Duplo bricks of the same size.

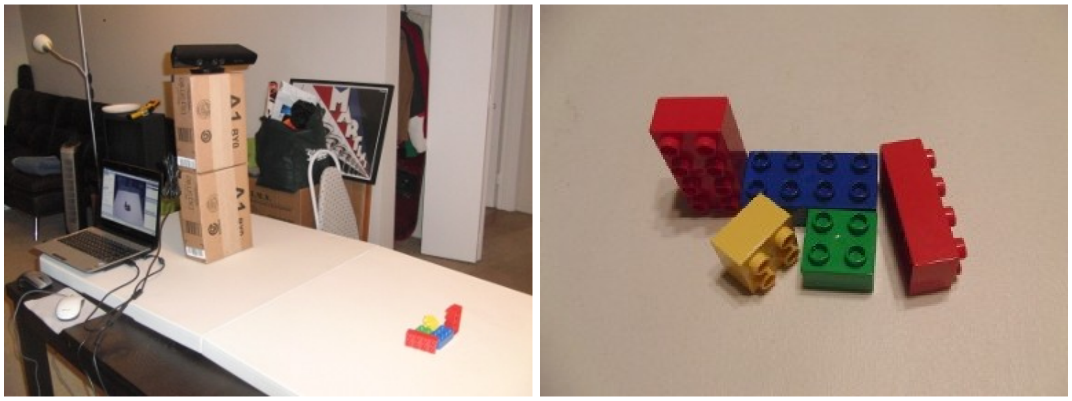


Figure 7. *(a) Home Kinect testbed set up, (b) Duplo brick test configuration.*

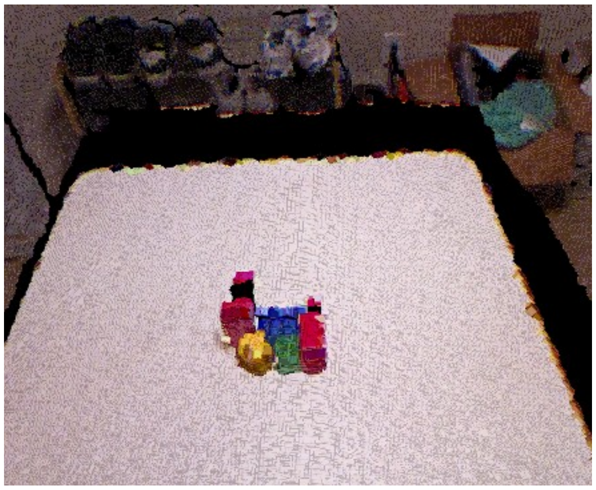
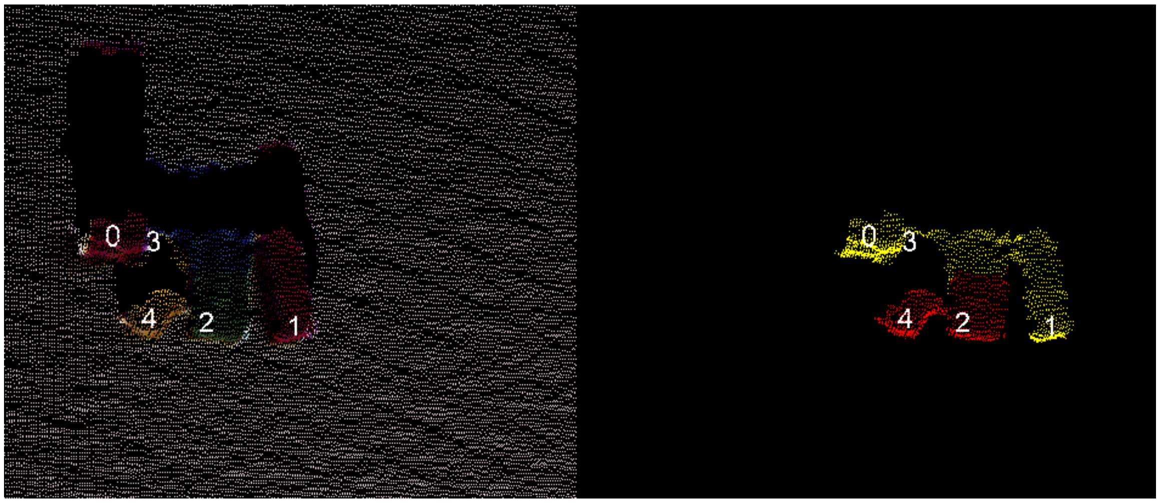


Figure 8. RViz visualization of Kinect point cloud.

Kinect Testbed Results



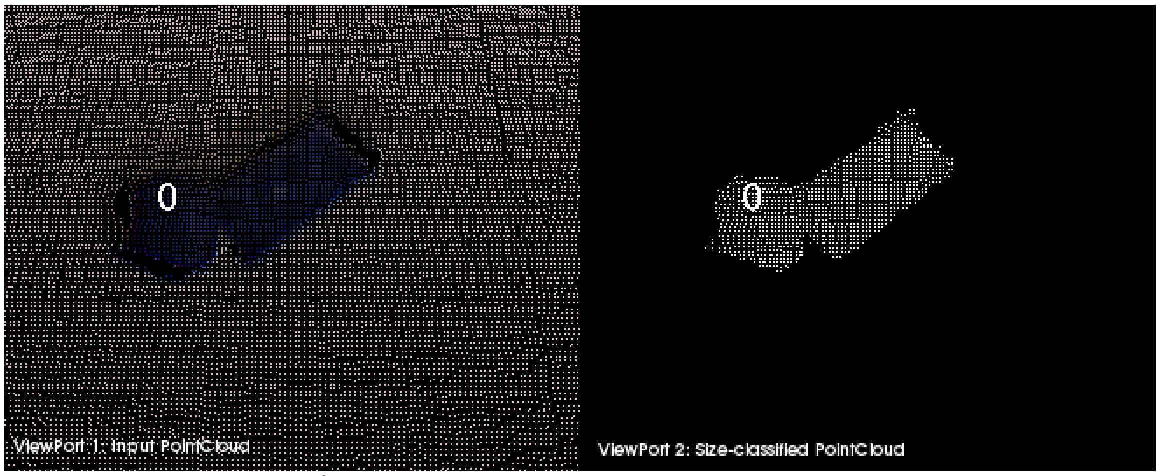
```
Number of RED clusters: 2
Number of GREEN clusters: 1
Number of BLUE clusters: 1
Number of YELLOW clusters: 1
TOTAL number of clusters: 5
There are 2 bricks of size 1x1 (cluster index: 2, 4, )
There are 3 bricks of size 1x2 (cluster index: 0, 1, 3, )
There are 0 bricks of size 1x4
There are 0 unclassified bricks
```

As shown by the results above, the tests with the Kinect sensor were successful. The algorithm correctly segmented and classified all Duplo bricks in the scene. Because the Kinect sensor publishes a new PointCloud2 message every two seconds, the algorithm is running in near-real-time. As the point cloud data changes (when Duplo bricks are added/removed or reoriented), the algorithm reacts accordingly and outputs the correct results.

V. Limitations

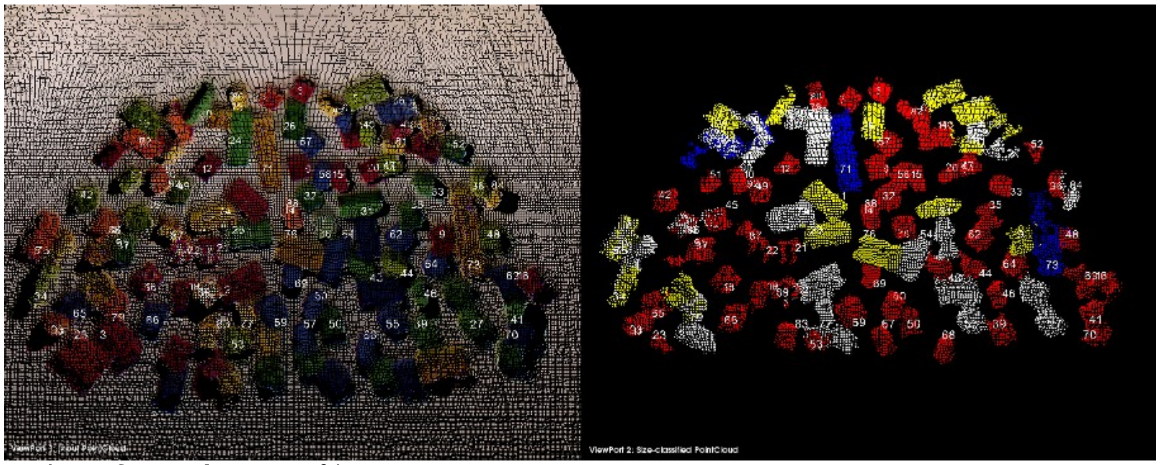
The biggest limitations of the algorithm are segmentation resolution and size classification. The segmentation error occurs when multiple Duplo bricks of the same color come into contact with each other. This causes the Euclidean cluster extraction algorithm to associate both bricks to the same cluster, which propagates error to the size estimation and classification algorithms. The size classification algorithm is also limited to strict situations, and will only work properly if all Duplos are situated on a planar surface. The current sensor model allows ±5mm in the height and width estimation, but this can be improved for robustness.

group12_2.pcd


0
ViewPort 1: Input PointCloud
0
ViewPort 2: Size-classified PointCloud

```
Number of RED clusters: 0
Number of GREEN clusters: 0
Number of BLUE clusters: 1
Number of YELLOW clusters: 0
TOTAL number of clusters: 1
There are 0 bricks of size 1x1
There are 0 bricks of size 1x2
There are 0 bricks of size 1x4
There are 1 unclassified bricks (cluster index: 0, )
```

group8_new1.pcd


ViewPort 1: Input PointCloud
ViewPort 2: Size-classified PointCloud

```
Number of RED clusters: 24
Number of GREEN clusters: 30
Number of BLUE clusters: 17
Number of YELLOW clusters: 24
TOTAL number of clusters: 95
There are 63 bricks of size 1x1 (cluster index: 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
        20, 21, 22, 23, 32, 33, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
        51, 52, 53, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 80, 82, 83, 87, 88,
        89, 90, 91, 92, 93, 94, )
There are 15 bricks of size 1x2 (cluster index: 1, 5, 9, 25, 26, 28, 29, 30, 31, 34, 56, 76, 79,
        81, 85, )
There are 3 bricks of size 1x4 (cluster index: 0, 71, 73, )
There are 14 unclassified bricks (cluster index: 2, 3, 4, 24, 27, 54, 55, 72, 74, 75, 77, 78,
        84, 86, )
```

As shown in the test cases, the algorithm fails to segment and classify Duplo bricks when multiple bricks of the same color are sufficiently close together. The algorithm handles these occurrences by classifying these clusters as "unclassified" and color-coding them as WHITE in the PCLVisualizer viewer.

VI. Conclusion

The "identifying Duplo bricks of the same size" task can be accomplished by using a series of filtering, segmentation, and clustering techniques from the Point Cloud Library to extract individual Duplo bricks from the larger point cloud. In each Duplo cluster, the size is estimated by calculating the height and width of its convex hull reconstruction. Finally, the estimated size is used to classify the cluster into a Duplo brick type, which is used in size comparison with other clusters.

This algorithm works well for point clouds that have its Duplo bricks sufficiently spread out across a planar surface. However, it's performance is limited when these assumptions are not met. To resolve these issues, improvements must be made to increase the resolution of the segmentation algorithm to distinguish between same-color bricks that are spaced closely together. The size estimation algorithm can be improved by utilizing more advanced PCL functions dealing with surface reconstruction, model fitting, and volume estimation. This will become easier as the integration of ROS and PCL develops better, more up-to-date support.

This sensing task of identifying the size of Duplo bricks can be extended to manipulation tasks in robots and robotic simulators. Once the sizes of objects are classified, a robot can perform more intelligent tasks when it understands the physical properties associated with those objects. For example, given an assortment of brick sizes, a robot can construct a pyramid structure using larger bricks at the base and smaller bricks at the top.