Introduction to Point-Cloud Processing

Wyatt Newman
April, 2015

Introduction: Three-D sensing is increasingly common in robotics. Example sources of 3-D data include tilting or rotating LIDARs, stereo cameras, or structured-light processing, such as the Microsoft Kinect sensor. Each of these sensors has its own virtues and limitations. Rotating LIDAR instruments can require a relatively long acquisition time to achieve reasonable 3-D sampling. The resulting 3-D images can be fairly noisy (on the order of cm, for the Hokuyo). However, such sensors have good range and work outdoors. Stereo vision can be relatively inexpensive, can be high resolution, and can augment range information with color and intensity. However, the range and field of view are restrictive, relative to LIDARs. Further, stereo vision depends on viewing targets that have suitable "texture", which helps to address the left/right camera pixel correspondence problem for generating disparity maps. The useful range of stereo vision depends on the interocular distance; wider separation between the cameras is required for greater depth range, but this also limits the near range (i.e., making the system far-sighted). On the other hand, optics can be designed over huge variations—from telescopes to microscopes.

The Kinect sensor uses a structured-light source of infra-red, projecting a "speckle" pattern. A single camera detects this projected pattern and interprets range from triangulation, equivalently deducing the shape of illuminated scenes from how the speckle pattern is projected over surfaces. The Kinect camera has surprisingly good performance for its low price. The depth points are "colorized" by associating corresponding data from a 2nd (color) camera. The density of points is high. However, the field of view and the range are limited (optimized for gaming applications), and this device does not work in sunlight.

This document will focus on use of the Kinect sensor and interactions with Rviz, but the principles are applicable to 3D images from other sources.

Point Clouds: There are a variety of representations of 3-D images. Many variations are accommodated with the "PointCloud" definition. (see http://pointclouds.org/about/).

A point-cloud has a header, with fields for a time stamp and for a reference frame. A point-cloud also has fields for "height," "width," and "is_dense."

One style of representation is to merely list "points" in a random order, where each point has associated data. This is an "unordered" point cloud. Unordered point clouds have a "height" of 1, and a "width" equal to the number of points. Data that originates from a rotating LIDAR typically results in unordered point clouds, since this data is not acquired in a simple raster pattern. Alternatively, stereo cameras and the Kinect camera produce "ordered" point clouds. Points in an *ordered* point cloud can be accessed by row and column (though index computations are required). The row and column indices of the point-cloud structure correspond to row and column indices of the image plane. At each such [i][j] location, one may find N-dimensional data, e.g. (x,y,z,R,G,B). In contrast to an unordered point cloid, an ordered point cloud can offer computational efficiencies, since points that are close together in [i][j] space are (typically) close together in (x,y,z,R,G,B) space—at least when these points correspond to samples on a smooth patch of surface in the scene.

The "is_dense" field of a point cloud declares whether or not (true/false) every point in the point cloud has valid data. Commonly, sensors provide imperfect results. Stereo vision, for example, typically suffers from inability to resolve the correspondence problem where there is insufficient texture in the scene, and corresponding points lack computable depth. Coordinates for such points may containt "inf" or "NaN" to indicate they are not valid. Such points may be included in the

point-cloud so as to preserve order in an ordered point cloud, or they may be included in a stream if the sensor is not smart or fast enough to remove such points. If the point cloud is not "dense", your processing code must test each value for validity.

A point-cloud may be constructed from a single view (e.g., from a robot standing still, staring at a scene). Alternatively a point-cloud may be the result of multiple sensing devices (but with all points transformed to a common reference frame), or possibly from a single sensor moving around in the environment (but, again, with each sensor-data contribution transformed into a common reference frame). If it is known that a range image is acquired from a single viewpoint, this information can be used to help reduce confusion in perceptual processing from occlusions and shadowing.

Point-cloud processing can be demanding of required memory, computation time and bandwidth. A 640x480 image from a Kinect sensor provides over 300k points, each with (x,y,z,R,G,B). Updated at 30Hz, this consumes approximately 300MB/sec of communication bandwidth (transmitting point clouds as ROS messages). Mere seconds of "bagging" such data can rapidly fill files on the order of gigabytes.

Because point-cloud processing can be demanding, it is important that algorithms analyzing point clouds are particularly efficient. Algorithms within the "Point Cloud Library" (PCL; see http://pointclouds.org) are optimized for efficiency in C++ code. Further, some of the core routines exploit Intel's Streaming SIMD Extensions (SSE), and some of the algorithms exploit multi-core CPU capabilities. Some of the internal computations use the "Eigen" library (see http://eigen.tuxfamily.org), which performs linear-algebra computations efficiently.

Tutorials on how to use the "Point Cloud Library" can be found at: http://pointclouds.org/documentation/tutorials/. Methods within this library are available for a useful variety of point-cloud processing, and all of the source code is openly available.

Some additional useful PCL resources include theses by Rodrigues, http://robotica.unileon.es/mediawiki/index.php/PhD-3D-Object-Tracking, and by Rusu, https://c5155196669c564620ef9c12a25f195f52ea786c.googledrive.com/host/0By7jPChnijtHSlp1QnlYanNhOE0/PCL/tutorials/html/how_features_work.html#rusudissertation. Introductory slides describing pointclouds can be found at http://ais.informatik.uni-freiburg.de/teaching/ws10/robotics2/pdfs/rob2-12-ros-pcl.pdf, and an intro to PCL processing can be found at http://www.jeffdelmerico.com/wp-content/uploads/2014/03/pcl_tutorial.pdf.

Since point clouds can be large, it is important to avoid making copies of these. Instead, point clouds are passed to functions via pointers. This provides a significant speed-up, but at the cost of some confusion regarding pointers and pointer dereferencing. It can be helpful to emulate working examples to construct new point-cloud processing code.

Snapshot program: In the example_pcl package, the file "pcd_save.cpp" shows how to subscribe to a point-cloud topic and how to save a pointcloud to disk in "PCD" format. The line:

```
ros::Subscriber getPCLPoints = nh.subscribe<sensor_msgs::PointCloud2> ("/kinect/depth/points", 1, kinectCB);
```

sets up a callback function, "kinectCB", to subscribe to the topic "/kinec/depth/points", which carries messages of type "sensor_msgs::PointCloud2". The cwruBot simulation emulates a kinect sensor and transmits data to this topic. A real Kinect sensor will publish similar data, and the example code is applicable to simulated or real data.

The callback function receives a pointcloud as a ROS message, converts it to a pcl::PointCloud datatype, and sets a flag to inform “main” that a pointcloud message has been received. The main program polls this flag, and when the flag is true, “main” saves the received data to disk. The line:

```
pcl::io::savePCDFileASCII ("test_pcd.pcd", *g_pclKinect);
```

saves the pointcloud in “g_pclKinect” to disk in “pcd” format in a file named “test_pcd.pcd”. This example program thus can be used to take a “snapshot” of streaming point-cloud data, saving a single scan to disk. This is useful for off-line data processing.

As an example, cwruBot was set up in front of a table, with a can setting upright on the table top. The emulated Kinect sensor streamed data to topic /kinect/depth/points. A snapshot was taken via:

```
rosrun example_pcd pcd_save
```

This created the file “test_pcd.pcd”. This example file is saved in our repository within the package “example_pcl.” An Rviz view of this snapshot is shown below:

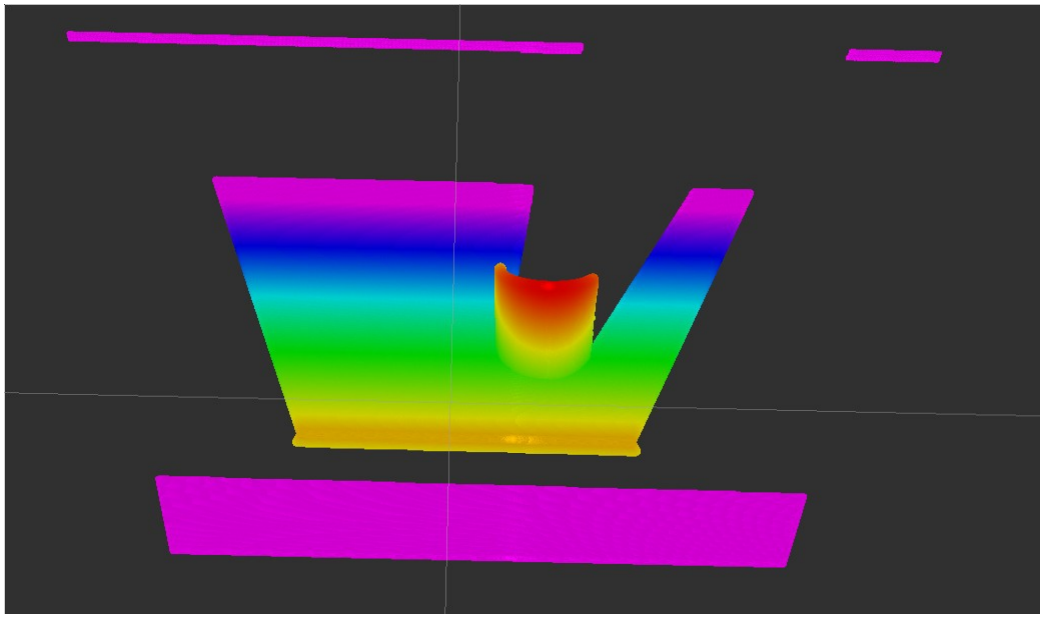

Illustration 1: Rviz display of captured point cloud. Color is set to display z distance from sensor

This view is displayed with color indicating the z-distance from the sensor (where the sensor's z axis is the optical axis, pointing out from the camera). The multi-colored region is the visible part of the table (and the violet strips are visible parts of the floor). The visible part of the can includes red points (points closest to the camera). From the view shown, “shadowing” from the can is observable as missing points behind the can. Only half of the can's cylindrical surface is visible, since depth points are limited to line of sight.

Point-Cloud Processing Overview: The example code “find_can_incomplete.cpp” in the package “example_pcl” illustrates some of the steps towards “registration” of a cylinder with the point-cloud data. The example code uses interactive assistance from an operator via Rviz. The example functions within this node are described separately below.

The main program, run with:

```
rosrun example_pcl find_can_incomplete
```

Only requires “roscore” and “rviz” to be running.

Main sets up a subscriber to the topic “selected_points” (which is published by rviz) and publishers to topics “kinect_pointcloud” and “plane_model.” Rviz should be configured to display “PointCloud2” messages from topics “kinect_pointcloud” and “plane_model.” The fixed frame should be set to “world.”

The main program next reads in the stored data from disk with the line:

```
pcl::io::loadPCDFile<pcl::PointXYZ> ("test_pcd.pcd", *g_cloud_from_disk)
```

Note that this example program must be run from a directory that contains the file “test_pcd.pcd” in order to find this data.

Main also sets up a ROS service, “process_mode”, which expects integer inputs. For example, the terminal command:

```
rosservice call process_mode 0
```

will cause the global variable “g_pcl_process_mode” to get set to zero and the flag “g_trigger” to get set to “true.” On each iteration of main, the g_trigger flag is tested, and if true, a switch/case statement directs main to execute a block of code selected by g_pcl_process_mode. The trigger is reset by main. Also on each iteration, two pointclouds are published:

```
pubPcdCloud.publish(g_cloud_from_disk); //keep displaying the original scene
pubCloud.publish(g_display_cloud); //and also display whatever we choose to put in here
```

Results of point-cloud processing can be viewed in Rviz via these topics.

Operation of the example code proceeds as follows. Once the program and rviz are running, the user should select a “patch” of points in rviz corresponding to the table top, e.g. as shown below:

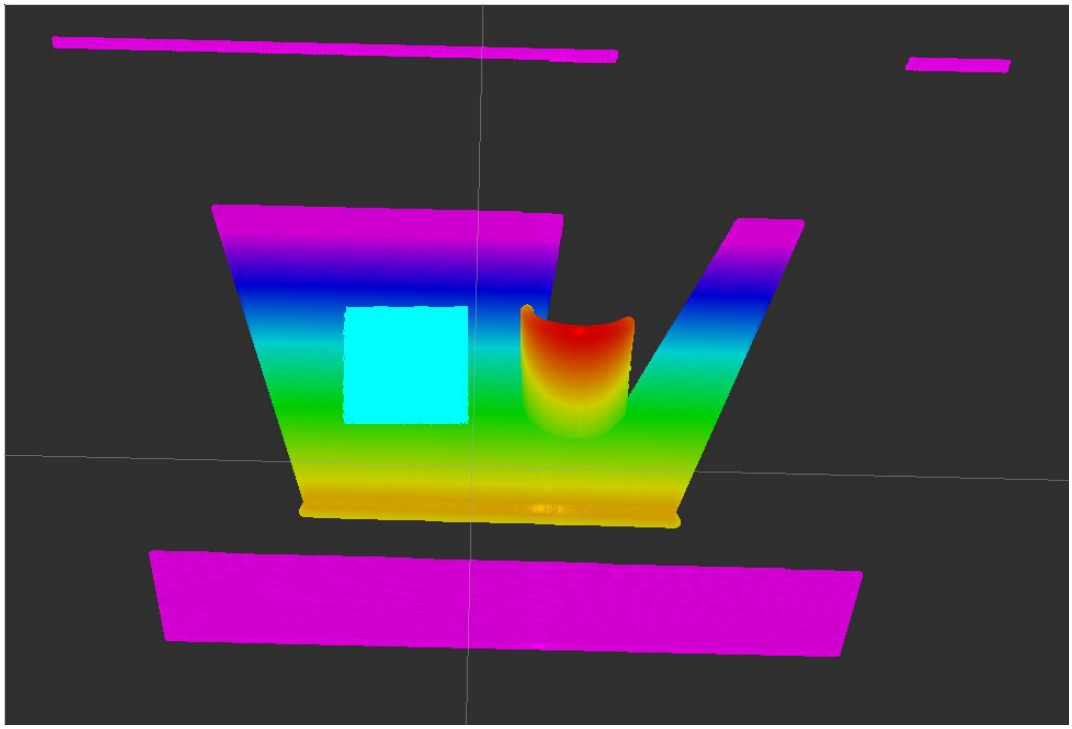

Illustration 2: light blue patch shows selection of points on the table top via the "publish selected points" tool in rviz

The points selected in rviz via the “publish selected points” tool are published to topic “selected_points.” This invokes execution of the callback function “selectCB().” Each time a patch is selected, the selected points are filtered to remove outliers, then the centroid and surface normal of the filtered points are computed. These values are saved in global variables g_patch_centroid

and g_plane_params. These results are used differently, depending on the processing mode.

After a table-top patch is selected, the first phase of processing is invoked with the command:
rosservice call process_mode 0
This sets the mode to IDENTIFY_PLANE, which invokes the function:
find_plane(g_plane_params, g_indices_of_plane);
This function uses the plane parameters to search the original point cloud to find all points that are within a tolerance Z_EPS of lying on the same plane as the selected patch. The result is displayed via the reduced pointcloud, g_display_cloud, which contains only the points determined to lie on the user-implied plane. The figure below shows the result of this operation:

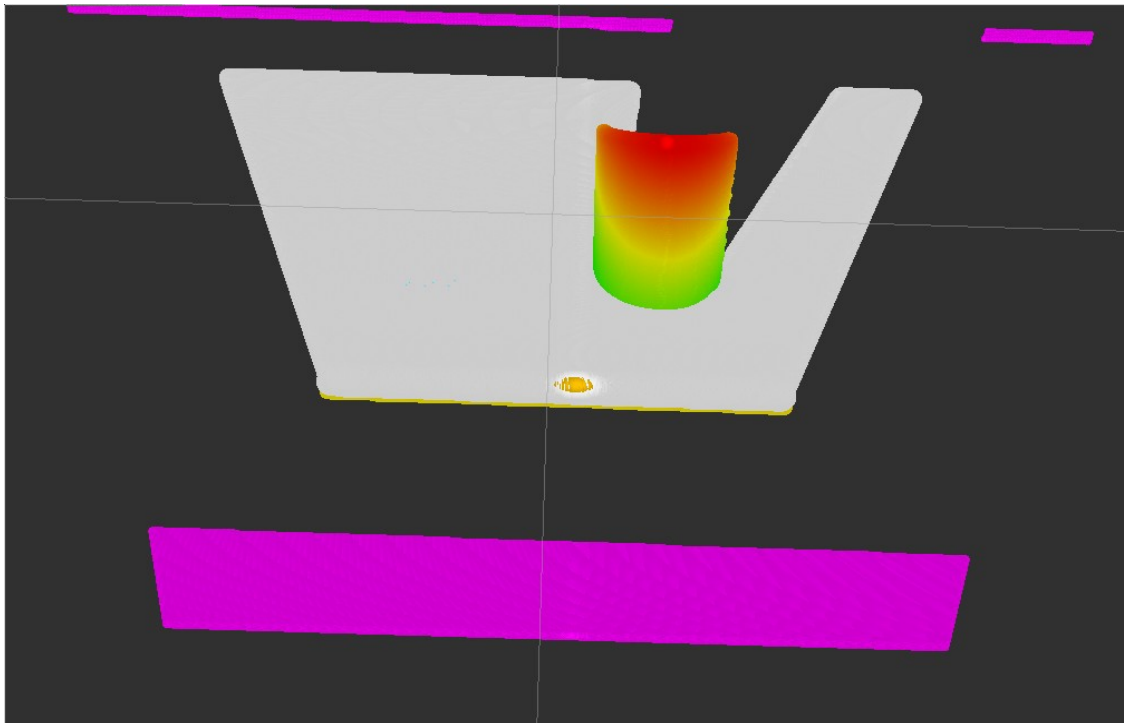

Illustration 3: display of points determined to lie on indicated plane, shown in gray

The gray region corresponds to the pointcloud determined to lie on the implied plane.

Next, points "above" the table are found by invoking mode 1, FIND_PNTS_ABOVE_PLANE, via:
rosservice call process_mode 1

The result of this operation is shown below:

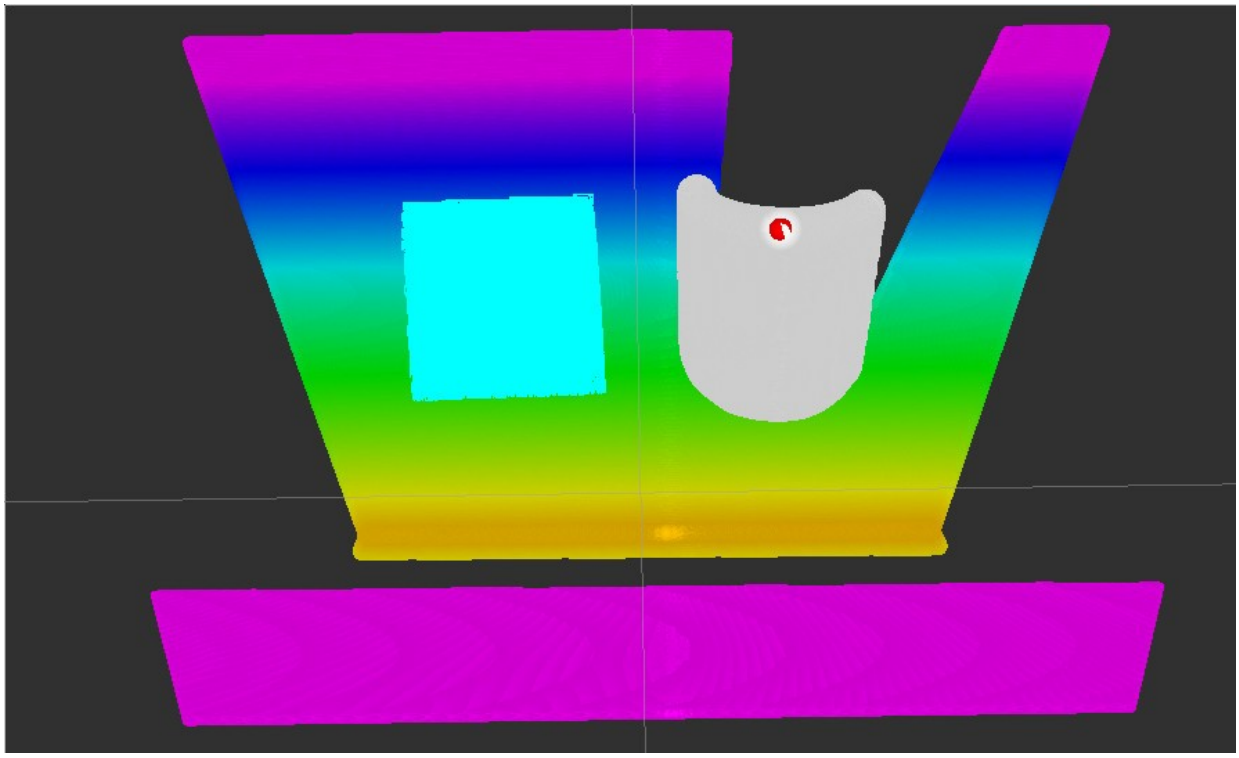

Illustration 4: Result of mode-1 operation. Points determined to be "above" the table are shown in gray.

Next, the user assists processing by selecting a region of points on the can, to help focus a region of interest. (In the present case, there is only one object on the table, so this step could be automated easily). If the patch is chosen to be roughly parallel to the cylinder's axis, the patch information can be used to help estimate the cylinder pose. The figure below shows an example region selection on the can:

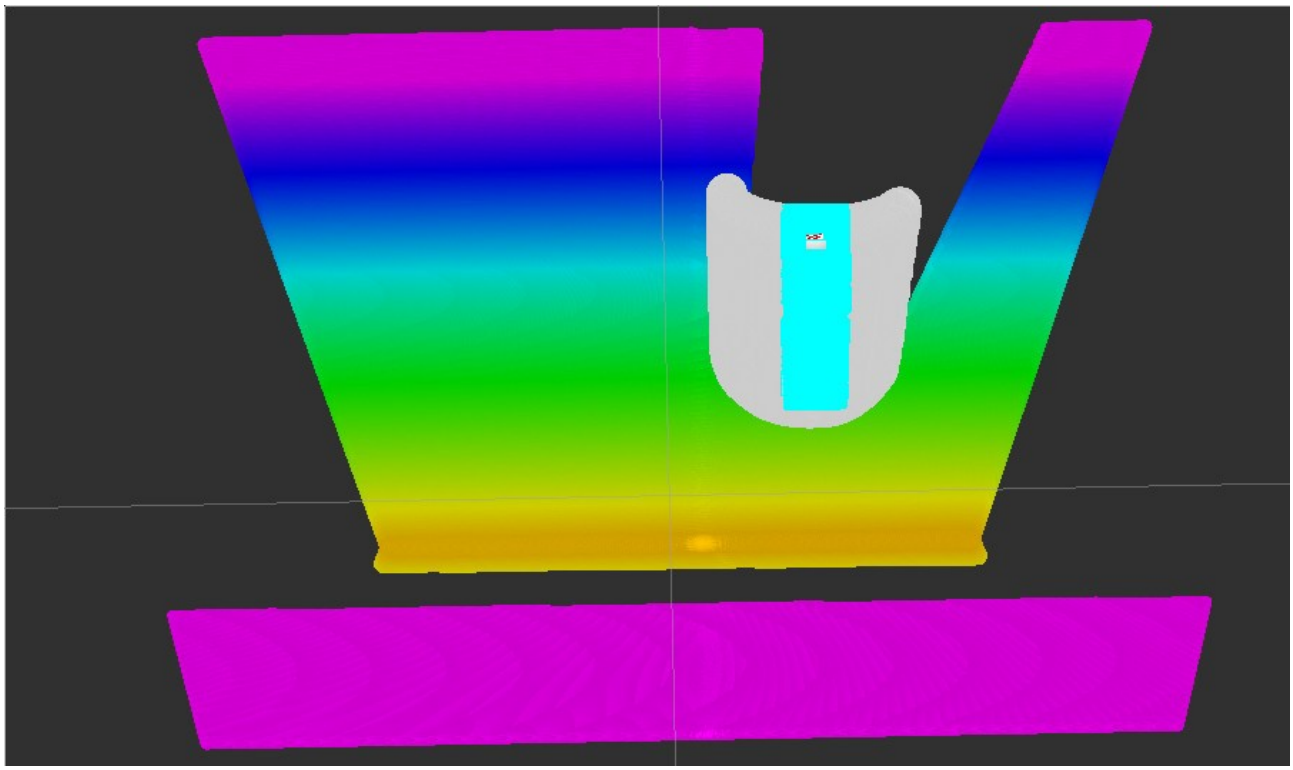

Illustration 5: Light blue region is selection of points on the desired object (cylinder). Selection of points roughly parallel to the cylinder's axis can assist esimation of registration.

After patch selection, invoking processing mode “2”, via: *rosservice call process_mode 2* produces an initial guess for the cylinder pose. A pointcloud is generated corresponding to a model of the expected cylinder, and the initial pose guess for this model is shown superimposed on the scan data. After mode-2 (COMPUTE_CYLINDRICAL_FIT_ERR_INIT) operation, rviz shows:

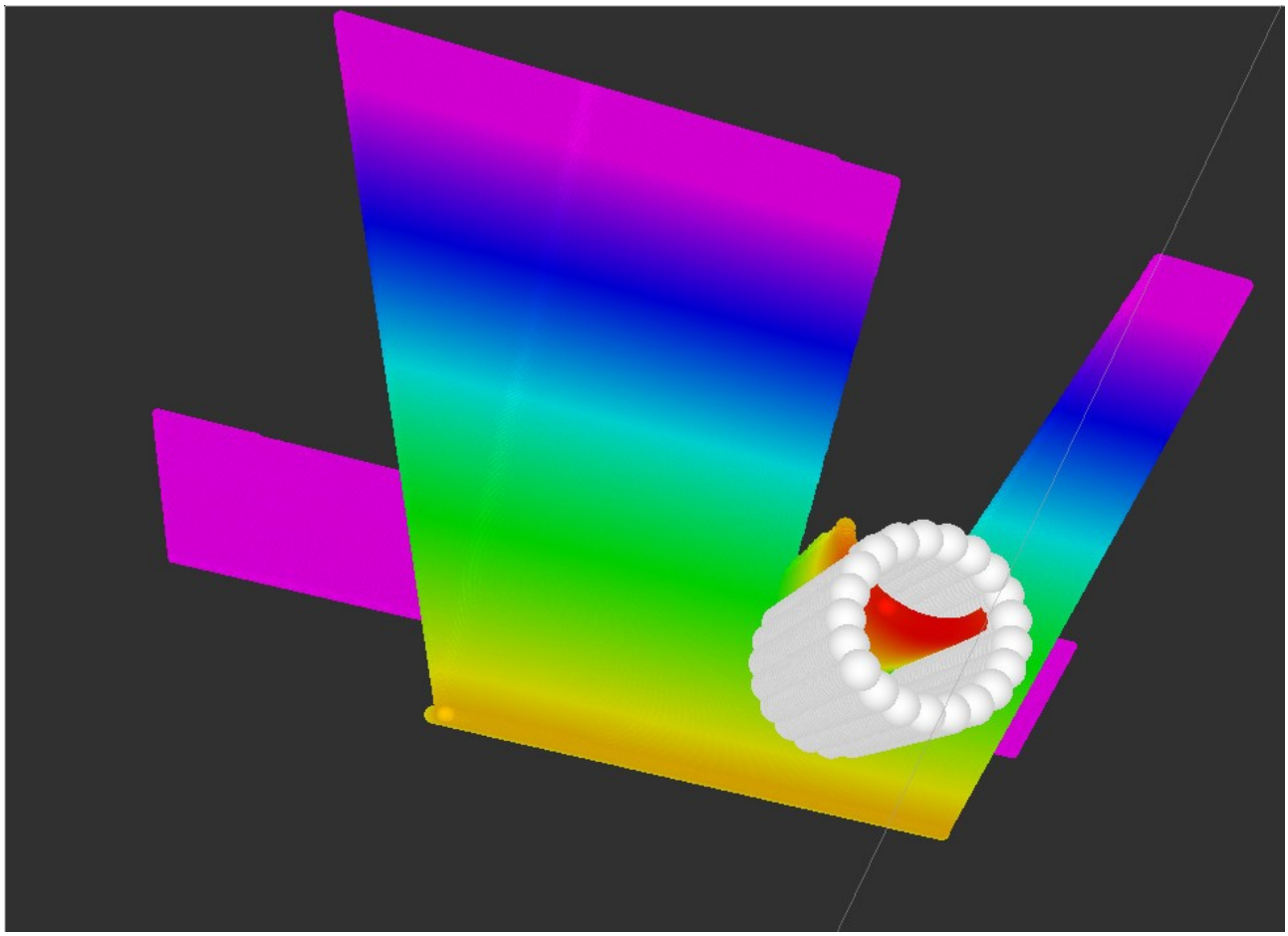

Illustration 6: Model (white points) of cylinder superimposed on original scan with initial estimate of pose. Objective or registration is to align the model with the appropriate subset of scan data.

This initial guess for model alignment uses the table-top plane model to set the height of the can (with can bottom aligned with the table surface) and the orientation of the can (with cylinder's axis parallel to the plane normal). The only unknowns are the x and y coordinates of the can in the frame of the table. The initial guess for these coordinates is based on the centroid of the selected patch on the can. This guess could be improved, e.g. by displacing these values by one radius of the cylinder in the negative direction of the patch selectd on the can's surface.

Mode 3 (COMPUTE_CYLINDRICAL_FIT_ERR_ITERATE) is intended to compute iterative improvements to the estimated cylinder-registration coordinates. This mode has the lines:

```
can_center_wrt_plane[0]+= 0.0;
can_center_wrt_plane[1]+= 0.0;
```

which do nothing. These should be edited to perturb the can's origin (x and y values) to improve the model fit.

An example of successful registration is shown in the rviz views below:

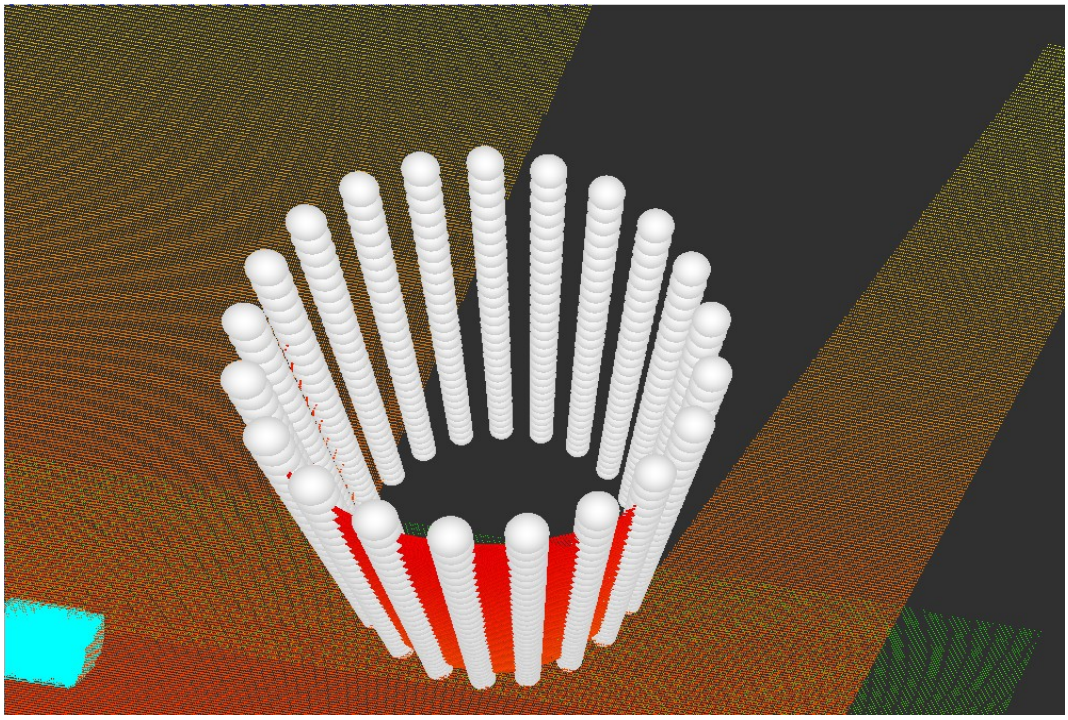

Illustration 7: rviz view of succesful registration of model (white points) to data (red points)

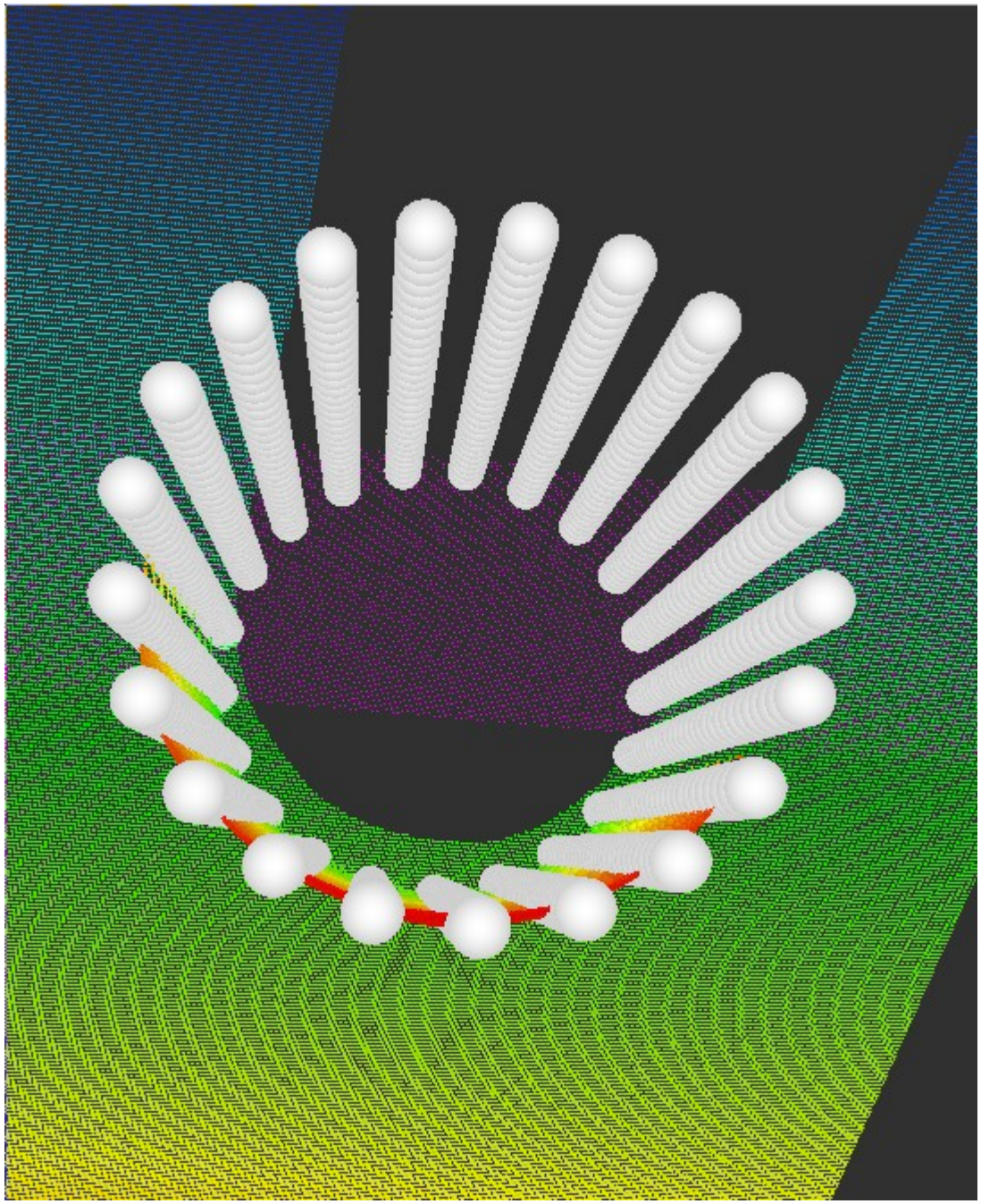

Illustration 8: 2nd view of successful model registration

Details of the functions that perform these steps follow.

Example code: process_patch: The process_patch() function is invoked by selecting a patch of points via rviz. Steps within this function include computeCentroid(), computeRsqd(), and computePointNormal(). Given a pointcloud (selected points), the centroid is computed by adding together all of the 3-D points, then dividing by the number of points. The example centroid function should be improved to exclude NaN or inf values, if the pointcloud is not "dense."

Given a pointcloud and its centroid, it may be necessary to remove outliers (even with NaN and inf values removed). To do so, all points in the pointcloud are evaluated in terms of distance from the centroid. The function computeRsqd() computes the squared distance of each point from the centroid. Based on these values, only points within one standard deviation from the centroid are retained. These points are identified via a list if indices in a vector of integers. After this statistical

filtering, the centroid of the surviving points is re-computed. The surface normal of the surviving points is computed using a PCL object:

```
NormalEstimation<PointXYZ, Normal> n; // object to compute the normal to a set of points
```

which is invoked as:

```
n.computePointNormal(*g_pclSelect, iselect_filtered, plane_params, curvature); // find plane params for filtered patch
```

This function computes the plane parameters, which includes the surface normal (elements 0, 1, 2) and the distance of the identified plane from the sensor-frame origin (element 3 of plane_params). The theory of operation of this function is described here: http://pointclouds.org/documentation/tutorials/normal_estimation.php, which uses the point-cloud centroid, covariance, and eigenvalues/eigenvectors to determine an optimal estimate for the surface normal from a (presumably, nearly flat) pointcloud.

Example code: find_plane(): The find_plane() function uses the results of process_patch in terms of the centroid and plane parameters. A coordinate frame is defined on the desired plane, with origin set at the patch centroid and z-axis set equal to the computed surface normal. Location of the frame origin is somewhat arbitrary, except that it must lie within the desired plane. Additionally, the orientation of the plane frame has some ambiguity, constrained only by the required direction of the z-axis parallel to the surface normal. Arbitrarily, the x-axis of the plane frame is defined to be as close as possible to parallel to the sensor-frame x-axis. The lines:

```
x_dir << 1, 0, 0; // keep x-axis the same...nominally
x_dir = x_dir - g_plane_normal * (g_plane_normal.dot(x_dir)); // force x-dir to be orthogonal to z-dir
x_dir /= x_dir.norm(); // want this to be unit length as well
```

force the plane-frame x-axis to be orthogonal to the plane-frame z-axis and to be of unit length. The y-axis follows from the cross product of z into x. These three axes define the columns of the rotation matrix g_R_transform. Together, the rotation matrix and plane-frame origin define a transformation matrix, g_A_plane. Any point expressed in plane-frame coordinates can be transformed to sensor coordinates by premultiplying the plane-frame point by the transform g_A_plane. Alternatively, sensor-frame points can be transformed to the plane frame by premultiplying by the inverse of g_A_plane. The line:

```
transform_cloud(g_cloud_from_disk, g_A_plane.inverse(), g_cloud_transformed);
```

invokes the transform_cloud() function, which transforms every point in “g_cloud_from_disk” into an output cloud, “g_cloud_transformed”, using the transformation g_A_plane.inverse(). In the resulting cloud, all points coincident with the desired plane should have z-values close to zero. Subsequently, the function:

```
filter_cloud_z(g_cloud_transformed, 0.0, z_eps, indices_z_eps);
```

evaluates all points within the transformed pointcloud, and for each point within tolerance “z_eps” about z=0, the corresponding point index is added to the vector of indices “indices_z_eps.” The function:

```
copy_cloud(g_cloud_from_disk, indices_z_eps, g_display_cloud);
```

populates the cloud “g_display_cloud” be extracting the points named within the vector “indices_z_eps” from the original input cloud. By extracting these points from the original cloud, the points are expressed in the original sensor frame. Thus, g_display_cloud can be displayed in Rviz, overlayed on the original data, thus highlighting the points presumed to be co-planar with the selected patch.

A similar operation is performed by:

```
filter_cloud_above_z(g_cloud_transformed, z_threshold, indices_pts_above_plane);
```

In this function, the transformed pointcloud is evaluated to find all points with z values> z_threshold. The points that satisfy this test are enumerated in the vector indices_pts_above_plane. Referring to the original pointcloud (in the sensor frame), extracting these points and displaying them in rviz shows the subset of points determined to be higher than the table top. These points are candidates for samples of objects sitting on the table.

Example code: make_can_cloud(): The function make_can_cloud() illustrates how one can create a pointcloud by computation, suitable for viewing in rviz. In this simple example, sample points are generated for a cylinder of radius "r" and height "h", with samples computing in cylindrical coordinates. The cylinder is presumed to have its origin at the base and to have its axis of symmetry along the z-axis. The cylinder model can be displayed in rviz as a pointcloud. By rotating and translating this point cloud, one can attempt to align the model with an appropriate subset of the scan data, thus achieving "registration" of a model to data. The transform that succeeds in achieving good registration contains the data necessary to say "where" is the object of interest relative to the sensor frame.

Example code: compute_radial_error(): The function compute_radial_error() operates on the points found to be "above" the table. Since multiple objects can be on the table, this is assisted by the user selecting a patch of points that lie on an object of interest. The centroid of this patch is used to help initialize the pose of the model (can_cloud) in the sensor frame. The model is constrained to be upright (cylinder's axis parallel to the table normal) and sitting on the table (bottom of the cylinder coincident with the height of the table). This is easiest to specify in the plane frame of the table top (constructed as described above). Given the elevation and orientation constraints on the cylinder, the only unknowns for registration are the x and y coordinates of the origin of the cylinder in the plane frame. To find these coordinates, one can attempt to minimize the fit error between the relevant sample scan points and the cylinder model. This is computed as:

```
pt = inputCloud->points[copy_indices[i]].getVector3fMap();
dx = center[0] - pt[0];
dy = center[1] - pt[1];
r_i = sqrt(dx*dx+dy*dy);
r_sqd_err = (r_i - r)*(r_i-r);
```

For each point in the selected cloud (in the plane frame), compute the Euclidean distance from a proposed (x,y) cylinder center to the sample point. This is an equivalent radius, if the point under consideration lies on the surface of a cylinder with center at "center[]." If *all* points considered have radius equal to the model radius, then the registration between model and data is perfect. Otherwise, we can compute the sum-squared error of points relative to the model radius, which is the sum over all points of r_sqd_err, above. A more heuristic measure of the model-fit error is the RMS error, computed as the square root of the mean of the squared error. This is interpretable as an average surface-fit error, in meters.

This function also computes an estimate of dEdCx and dEdCy, which are the expected sensitivities of fit error with respect to perturbations of center coordinates in x and y, respectively. These estimated derivatives can be useful in automating a search for optimal registration.

Based on the current best estimate of the origin of the model fit, expressed in plane coordinates, one can define a transformation for the cylinder model. The rotation is identical to the plane-frame rotation, in order to align the model's z-axis with the plane normal.

```
A_plane_to_sensor.linear() = g_R_transform;
```

The vector from sensor origin to cylinder-model origin, g_cylinder_origin, expressed in the sensor frame, completes specification of the transform:

```
A_plane_to_sensor.translation() = g_cylinder_origin;
```

With this transform, the model of the cylinder can be transformed to show its alignment relative to

the data, as expressed in the sensor frame:

```
transform_cloud(g_canCloud, A_plane_to_sensor, g_display_cloud);
```

The cloud “g_display_cloud” is published by the main program so that it can be viewed in rviz. When the model is well aligned with the (relevant) scan data, the fit error, E, will be small and the model can be seen to be consistent with the data.

Incomplete code: The example code is intentionally incomplete, left as an exercise. Notably, the initial guess for the model origin, g_cylinder_origin = g_patch_centroid, can be improved. Most importantly, the mode “COMPUTE_CYLINDRICAL_FIT_ERR_ITERATE” does nothing. It should adjust the origin of the cylinder model incrementally to improve the registration iteratively. (The derivatives dEdCx and dEdCy can be useful for implementing a gradient-descent algorithm).

Conclusion: This example code illustrates some relatively simple point-cloud processing. The code should be reorganized as a class with methods. Better still, the methods may be added to existing PCL classes. Some of the code is redundant with existing PCL methods. Greater productivity (and simpler, more reliable code) can be achieved through greater use of existing PCL methods.

The examples of plane fitting and cylinder-model registration are simpler than the more general methods in PCL, since these methods assume help from an operator (via point selections in rviz). Fully automated model registration in PCL can be hard to make robust. When user assistance is feasible, such input can be exploited to make registration faster and simpler than the general case. When fitting models, the problem is simplified by context, with explicit declaration of what model is expected to fit in the scene. For multiple objects in a scene, “segmentation” of the point-cloud data can be an important preprocess, helping to reduce the set of points considered to be associated with any one object. PCL functions can be used to perform such segmentation.

For our objectives, we desire to compute the coordinates (pose) of an object of interest with respect to the sensor frame. Subsequently, with hand/eye calibration, we can compute coordinates for grasp with a robot arm.