

# Introduction to Point-Cloud Processing

Wyatt Newman

October, 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 cloud, 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 contain “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.

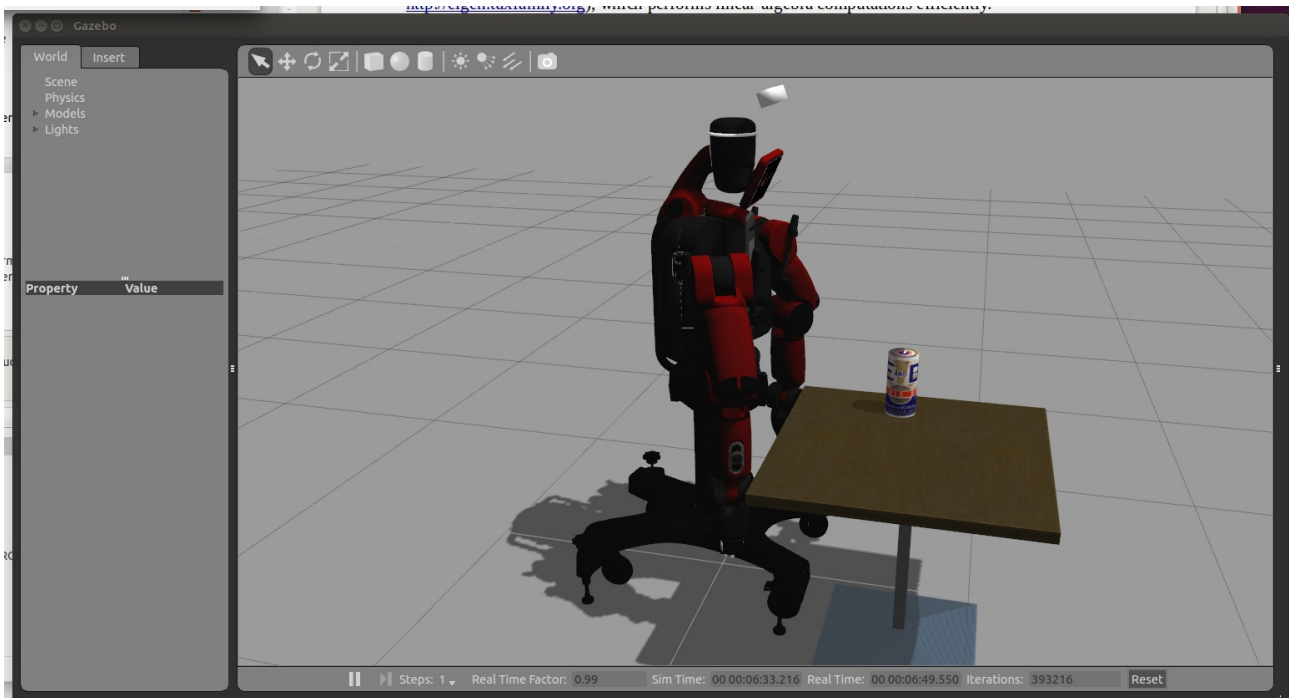
Some additional useful PCL resources include theses by Rodrigues, <http://robotica.unileon.es/mediawiki/index.php/PhD-3D-Object-Tracking>, and by Rusu, [https://c5155196669c564620ef9c12a25f195f52ea786c.googleusercontent.com/host/0By7jPChnjtHSlp1QnLYanNhOE0/PCL/tutorials/html/how\\_features\\_work.html#rusudissertation](https://c5155196669c564620ef9c12a25f195f52ea786c.googleusercontent.com/host/0By7jPChnjtHSlp1QnLYanNhOE0/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](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.

One can visualize a point-cloud display in rviz, given a real or an emulated point-cloud source. We can get a point-cloud stream from the Baxter simulator with the following commands.

In one terminal start up the Baxter simulator (modified to include a Kinet camera) with:  
`roslaunch cwru_baxter_sim baxter_world.launch`

The Gazebo view will look like the following:



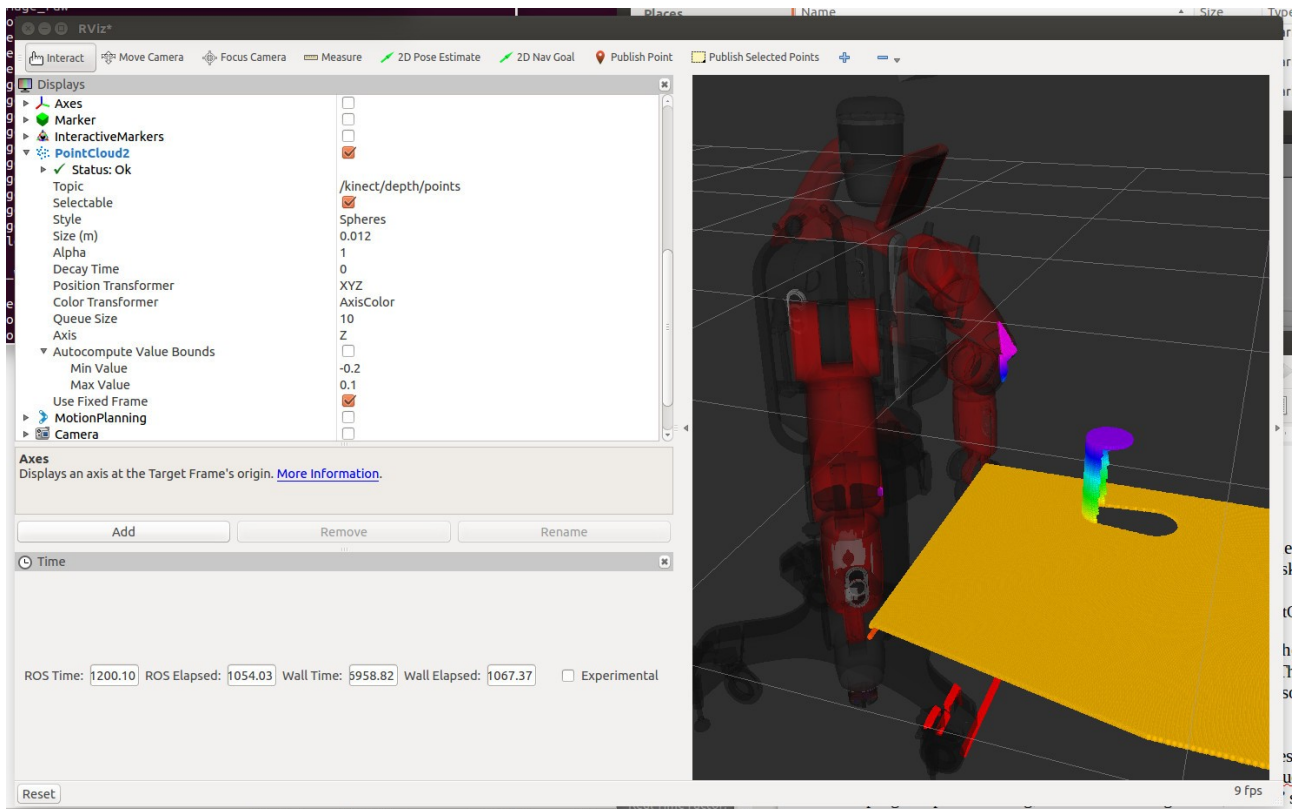
This view shows all of the models contained in the virtual world, including the robot, the ground plane, a cafe table and a beer can. The sensor element within the Kinect link has a transform with respect to the Kinect-link frame, and this transform is not part of the URDF. We need to publish this relationship on the “tf” topic. This is accomplished using the (ROS-provided) node `tf::static_transform_publisher`. This node can be started up conveniently (along with the necessary transform coordinates) via a launch file. From a second terminal, run:

```
roslaunch cwru_baxter_sim kinect_xform.launch
```

With this running, there is enough information (from the robot model on the parameter server, plus all of the transforms being published on tf, plus the emulated Kinect data being published on topic “kinect/depth/points”) for Rviz to receive the Kinect's 3-D point data, transform it to world coordinates, and display the results in 3-D. To do so, start rviz:

```
roslaunch rviz rviz
```

Which produces the result below.



In the above view, the point-cloud data has been colorized by choosing options for the display. The rviz “Displays” panel has added a topic “PointCloud2.” Expanding the menu options for this item, the “Topic” is set to “kinect/depth/points”, which is the name of the topic to which the Kinect gazebo emulation publishes. The “Style” of points was chosen to be “Spheres” of 0.012m diameter. The “Color Transformer” was set to “AxisColor” with the axis chosen to be the z-axis. Point colors range from red (lowest z value) to violet (highest z value). The range of z values applied to variable color can be automatically applied, or it can be manually specified, as in the present case, having set z-min to -0.2m and z-max to 0.1m. With this “zoom” on color, the beer can is yellow at the surface of the table, and the color varies up to purple at the top of the can. This visualization is similar to that which we introduced earlier, using the minimal robot. But we can do more with the point-cloud data than display it. We can operate on the data to try to make sense of it. This will be explored in the “cwru\_baxter” repository within the package “pcl\_perception\_node.”

**A Snapshot Program:** In the pcl\_perception\_node 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. Line 61, in main(), is:

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

which sets up a callback function, “kinectCB”, to subscribe to the topic “/kinect/depth/points”. This topic carries messages of type “sensor\_msgs::PointCloud2.” The Baxter simulation emulates a kinect sensor and transmits data to this topic. A real Kinect sensor would publish similar data, and the example code is applicable to simulated or real data (provided the topic names are consistent).

The callback function responds to an incoming pointcloud ROS message. The callback function converts this message type to a pcl::PointCloud datatype, thus making it appropriate for use with the Point Cloud Library. However, the only PCL function used in this callback is:

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

which saves the pointcloud to disk under the name “snapshot.pcd” (in the current directory of the

terminal that executed `pcd_save`). After saving the data, the callback function sets the global flag “`g_got_cloud`”, which informs “`main`” that a pointcloud has been received and saved. `Main()` then returns, ending the program.

This example program thus can be used to take a “snapshot” of streaming point-cloud data, saving a single scan to disk. Such snapshots can be useful for off-line data processing.

Saving the pointcloud as ASCII is larger than the alternative binary storage. However, it does allow one to manually view the data. The initial lines of “`snapshot.pcd`” are:

```
# .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 307200
```

```
HEIGHT 1
```

```
VIEWPOINT 0 0 0 1 0 0 0
```

```
POINTS 307200
```

```
DATA ascii
```

```
-1.3971394 -1.047308 2.4242566
```

```
-1.3927666 -1.047308 2.4242566
```

```
-1.3883936 -1.047308 2.4242566
```

```
-1.3840208 -1.047308 2.4242566
```

```
-1.3796479 -1.047308 2.4242566
```

```
-1.375275 -1.047308 2.4242566
```

```
-1.3709021 -1.047308 2.4242566
```

```
-1.3665292 -1.047308 2.4242566
```

```
-1.3621563 -1.047308 2.4242566
```

```
-1.3577834 -1.047308 2.4242566
```

```
-1.3534105 -1.047308 2.4242566
```

```
-1.3490376 -1.047308 2.4242566
```

```
-1.3446647 -1.047308 2.4242566
```

```
-1.3402919 -1.047308 2.4242566
```

```
...
```

This output shows that the header describes how the rest of the file is organized, which is 3 fields of single-precision (4-byte) values for the x, y and z sensed points. These are organized as one long list of 307,200 points. The dimensions will be in meters. From the viewpoint of the sensor, the z-axis points straight out from the camera, along the optical axis.

The data thus acquired and saved is somewhat hard to interpret, since it is in the (tilted) sensor frame. We will often be interested in surfaces that are horizontal (e.g. floors, tables) or vertical (e.g., walls, doors) with respect to the world frame. It is thus useful to transform the acquired data into a frame within which it will be easier to interpret the points.

**Transforming Point-Cloud Data to Alternative Frames:** The program

“`pcd_transform_and_save`”, also within the package “`pcl_perception_node`”, is related to `pcd_save`, except it also performs a transformation to express the sensor data in the “torso” frame of the robot. This will be useful for hand/eye coordination, since we will be able to express the hand frame with respect to the torso, and we can compare this to the coordinates of an object of interest, as perceived by the Kinect but also expressed with respect to the torso frame.

To transform the Kinect data, we bring in a “transform listener” to obtain the transforms between the sensor frame and the torso. We will look up a new transform for every snapshot, since (in a more general use of this example program) we would also be interested in including the influence of pan rotation of Baxter's head, on which we have mounted the Kinect. The transform listener will be aware of such motions and will take this into account, resulting in consistent sensor transforms to the torso frame.

The function `transformTFToEigen()` is similar to a version introduced earlier. It takes a `tf::Transform` as an argument, and it converts this transform into an `Eigen::Affine3f`. The “f” (float) version is used instead of the “d” (double-precision float), since our pointcloud objects use 4-byte floats. This is desirable, since the sensor accuracy does not justify double-precision representation, and the 307,200 points to be saved consume significant memory (which is halved using single-precision representation).

A second helper function is: `transform_cloud()`. This function shows a way to perform coordinate transformation of a pointcloud. This function takes input arguments of a source pointcloud, an `Eigen::Affine3f` transform, and a destination pointcloud to be populated with transformed data. Within this function, we manually copy over the header information from the input pointcloud to the output pointcloud. We then transform each 3-D point from the original sensor frame to the desired torso frame using the line:

```
outputCloud->points[i].getVector3fMap() = A * inputCloud->points[i].getVector3fMap();
```

repeated for all of the points in the pointcloud. This somewhat odd syntax allows one to extract the 3-D pointcloud data into an Eigen-compatible vector (which is then pre-multiplied by the transform “A”). It also allows one to put 3-D data back into a pointcloud, converting from an Eigen-type vector into compatible components of the pointcloud points.

The callback function, which responds to incoming pointcloud data on the Kinect topic, uses these two helper functions to transform the received data into world coordinates.

The transformed pointcloud data is saved to a file named “snapshot\_wrt\_torso.pcd.” The first part of this file appears as:

```
# .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 307200
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS 307200
DATA ascii
2.0545802 1.3971394 -0.9300018
2.0545802 1.3927666 -0.9300018
2.0545802 1.3883936 -0.9300018
2.0545802 1.3840208 -0.9300018
2.0545802 1.3796479 -0.9300018
2.0545802 1.375275 -0.9300018
2.0545802 1.3709021 -0.9300018
2.0545802 1.3665292 -0.9300018
```

The z-value of -0.930 m is repeated for a large number of points. This value corresponds to the height of the floor relative to the torso. For ideal Kinect data (no noise was added to the Kinect emulator), the z-height of the floor is precisely uniform.

Another group of point data appears as:

```
1.171401 -0.53229457 -0.15530008
1.171401 -0.53478777 -0.15530008
1.171401 -0.53728098 -0.15530008
1.171401 -0.53977412 -0.15530008
1.171401 -0.54226732 -0.15530008
1.171401 -0.54476053 -0.15530008
1.171401 -0.54725367 -0.15530008
1.171401 -0.54974687 -0.15530008
1.171401 -0.55224007 -0.15530008
1.171401 -0.55473322 -0.15530008
1.171401 -0.55722642 -0.15530008
1.171401 -0.55971962 -0.15530008
```

The z-value of -0.155 corresponds to the height of the table relative to the torso. A large number of points share this same z value, corresponding to the surface of the table.

A smaller set of points looks like the following:

```
0.68502676 -0.02413924 0.075000226
0.68502682 -0.025927335 0.075000107
0.68502688 -0.027715432 0.074999988
0.68502688 -0.029503522 0.074999988
0.68502694 -0.031291619 0.074999928
0.68502682 -0.033079706 0.075000048
0.68502682 -0.034867797 0.075000048
0.68502682 -0.036655892 0.075000048
0.68502682 -0.038443983 0.075000048
0.68502682 -0.040232074 0.075000048
0.68502682 -0.042020168 0.075000048
0.68502682 -0.043808259 0.075000048
```

The z-height of 0.075 presumably corresponds to points on the top of the beer can.

Within this new representation, it will be much easier to find horizontal or vertical surfaces of interest. This will be explored further with additional Point Cloud Library processing examples.