

---

# **PCL documentation Documentation**

***Release 0.0.1***

**Arzoo**

**Apr 10, 2020**



<b>1</b>	<b>PCL Walkthrough</b>	<b>3</b>
1.1	Overview . . . . .	4
1.2	Filters . . . . .	6
1.3	Features . . . . .	7
1.4	Keypoints . . . . .	9
1.5	Registration . . . . .	11
1.6	Kd-tree . . . . .	13
1.7	Octree . . . . .	16
1.8	Segmentation . . . . .	18
1.9	Sample Consensus . . . . .	20
1.10	Surface . . . . .	21
1.11	Range Image . . . . .	23
1.12	I/O . . . . .	24
1.13	Visualization . . . . .	25
1.14	Common . . . . .	29
1.15	Search . . . . .	29
1.16	Binaries . . . . .	30
<b>2</b>	<b>Getting Started / Basic Structures</b>	<b>33</b>
2.1	Compiling your first code example . . . . .	34
<b>3</b>	<b>Using PCL in your own project</b>	<b>35</b>
3.1	Prerequisites . . . . .	35
3.2	Project settings . . . . .	35
3.3	The explanation . . . . .	36
3.4	Compiling and running the project . . . . .	37
3.5	Weird installations . . . . .	38
<b>4</b>	<b>Compiling PCL from source on POSIX compliant systems</b>	<b>39</b>
4.1	Stable . . . . .	39
4.2	Experimental . . . . .	40
4.3	Dependencies . . . . .	41
4.4	Troubleshooting . . . . .	42
<b>5</b>	<b>Customizing the PCL build process</b>	<b>43</b>
5.1	Audience . . . . .	43
5.2	Prerequisites . . . . .	43

5.3	PCL basic settings . . . . .	44
5.4	The explanation . . . . .	44
5.5	Tweaking basic settings . . . . .	45
5.6	Tweaking advanced settings . . . . .	45
5.7	Detailed description . . . . .	46
<b>6</b>	<b>Building PCL's dependencies from source on Windows</b>	<b>49</b>
6.1	Requirements . . . . .	50
6.2	Building dependencies . . . . .	50
6.3	Building PCL . . . . .	55
<b>7</b>	<b>Compiling PCL from source on Windows</b>	<b>57</b>
7.1	Requirements . . . . .	58
7.2	Downloading PCL source code . . . . .	59
7.3	Configuring PCL . . . . .	59
7.4	Building PCL . . . . .	66
7.5	Installing PCL . . . . .	68
7.6	Advanced topics . . . . .	69
7.7	Using PCL . . . . .	69
<b>8</b>	<b>Compiling PCL and its dependencies from MacPorts and source on Mac OS X</b>	<b>71</b>
8.1	Prerequisites . . . . .	72
8.2	PCL Dependencies . . . . .	72
8.3	Building, Compiling and Installing PCL Dependencies . . . . .	73
8.4	Building PCL . . . . .	75
8.5	Using PCL . . . . .	76
8.6	Advanced (Developers) . . . . .	76
<b>9</b>	<b>Installing on Mac OS X using Homebrew</b>	<b>77</b>
9.1	Prerequisites . . . . .	78
9.2	Using the formula . . . . .	78
9.3	Using PCL . . . . .	78
<b>10</b>	<b>Using PCL with Eclipse</b>	<b>79</b>
10.1	Prerequisites . . . . .	79
10.2	Creating the eclipse project files . . . . .	80
10.3	Importing into Eclipse . . . . .	83
10.4	Configuring Eclipse . . . . .	83
10.5	Setting the PCL code style in Eclipse . . . . .	83
10.6	Launching the program . . . . .	84
10.7	Where to get more information . . . . .	84
<b>11</b>	<b>Generate a local documentation for PCL</b>	<b>85</b>
11.1	Dependencies . . . . .	85
11.2	Generate the documentation . . . . .	85
11.3	Installing and configuring Apache . . . . .	86
<b>12</b>	<b>Using a matrix to transform a point cloud</b>	<b>87</b>
12.1	The code . . . . .	87
12.2	The explanation . . . . .	87
12.3	Compiling and running the program . . . . .	88
12.4	More about transformations . . . . .	90
<b>13</b>	<b>Adding your own custom <i>PointT</i> type</b>	<b>93</b>
13.1	Why <i>PointT</i> types . . . . .	93

13.2	What <i>PointT</i> types are available in PCL? . . . . .	94
13.3	How are the point types exposed? . . . . .	103
13.4	How to add a new <i>PointT</i> type . . . . .	105
13.5	Example . . . . .	105
<b>14</b>	<b>Writing a new PCL class</b>	<b>107</b>
14.1	Advantages: Why contribute? . . . . .	108
14.2	Example: a bilateral filter . . . . .	108
14.3	Setting up the structure . . . . .	110
14.4	Filling in the class structure . . . . .	112
14.5	Taking advantage of other PCL concepts . . . . .	120
14.6	Testing the new class . . . . .	128
<b>15</b>	<b>How 3D Features work in PCL</b>	<b>129</b>
15.1	Theoretical primer . . . . .	129
15.2	Terminology . . . . .	130
15.3	How to pass the input . . . . .	131
15.4	An example for normal estimation . . . . .	132
<b>16</b>	<b>Indices and tables</b>	<b>135</b>
	<b>Bibliography</b>	<b>137</b>



---

**Note:** This site is under construction :)

---

The Point Cloud Library (PCL) is a standalone, large scale, open project for 2D/3D image and point cloud processing. PCL is released under the terms of the BSD license, and thus free for commercial and research use. We are financially supported by a consortium of commercial companies, with our own non-profit organization, Open Perception. We would also like to thank individual donors and contributors that have been helping the project.



---

## PCL Walkthrough

---

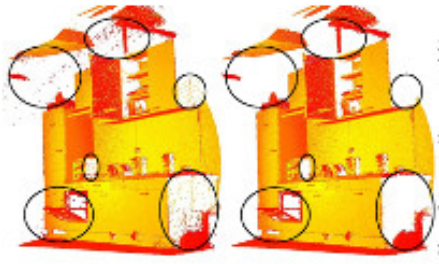
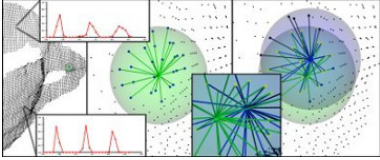
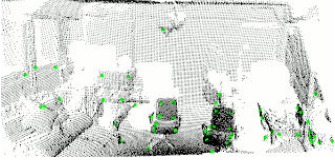
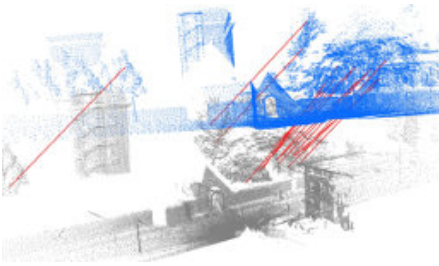
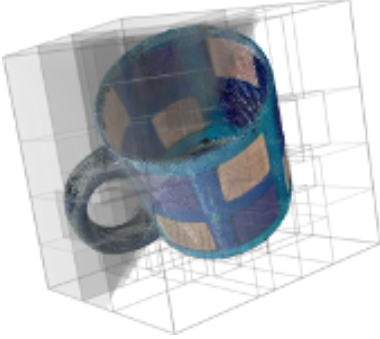
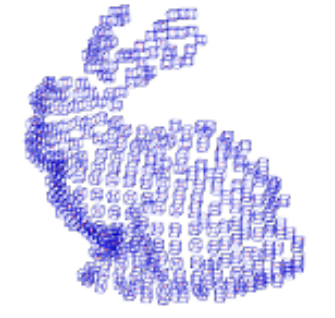


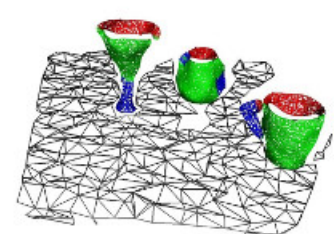
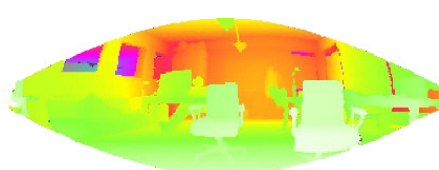




This tutorial will walk you through the components of your PCL installation, providing short descriptions of the modules, indicating where they are located and also listing the interaction between different components.

### Contents

- *PCL Walkthrough*
  - *Overview*
  - *Filters*
  - *Features*
  - *Keypoints*
  - *Registration*
  - *Kd-tree*
  - *Octree*
  - *Segmentation*
  - *Sample Consensus*
  - *Surface*
  - *Range Image*
  - *I/O*
  - *Visualization*
  - *Common*
  - *Search*
  - *Binaries*

## 1.1 Overview

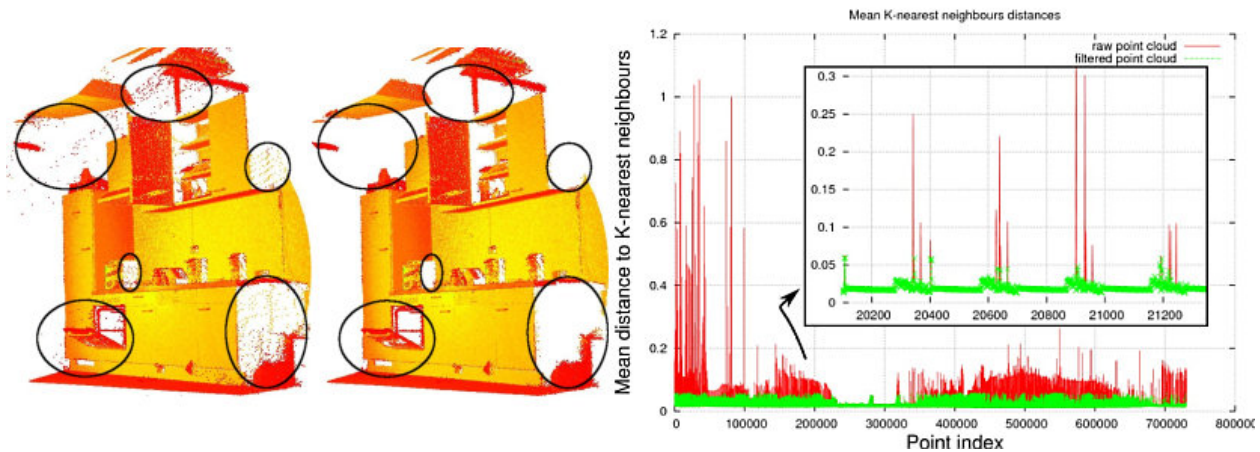
PCL is split in a number of modular libraries. The most important set of released PCL modules is shown below:

<i>Filters</i>	<i>Features</i>	<i>Keypoints</i>
		
<i>Registration</i>	<i>KdTree</i>	<i>OcTree</i>
		
<i>Segmentation</i>	<i>Sample Consensus</i>	<i>Surface</i>
		
<i>Range Image</i>	<i>I/O</i>	<i>Visualization</i>
		
<i>Common</i>	<i>Search</i>	
		

## 1.2 Filters

### Background

An example of noise removal is presented in the figure below. Due to measurement errors, certain datasets present a large number of shadow points. This complicates the estimation of local point cloud 3D features. Some of these outliers can be filtered by performing a statistical analysis on each point's neighborhood, and trimming those that do not meet a certain criteria. The sparse outlier removal implementation in PCL is based on the computation of the distribution of point to neighbor distances in the input dataset. For each point, the mean distance from it to all its neighbors is computed. By assuming that the resulting distribution is Gaussian with a mean and a standard deviation, all points whose mean distances are outside an interval defined by the global distances mean and standard deviation can be considered as outliers and trimmed from the dataset.



**Documentation:** [http://docs.pointclouds.org/trunk/group\\_filters.html](http://docs.pointclouds.org/trunk/group_filters.html)

**Tutorials:** <http://pointclouds.org/documentation/tutorials/#filtering-tutorial>

**Interacts with:**

- *Sample Consensus*
- *Kdtree*
- *Octree*

**Location:**

- **MAC OS X (Homebrew installation)**

- Header files: `$(PCL_PREFIX)/pcl-$(PCL_VERSION)/pcl/filters/`
- *Binaries*: `$(PCL_PREFIX)/bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix `CMAKE_INSTALL_PREFIX`, e.g., `/usr/local/`

- **Linux**

- Header files: `$(PCL_PREFIX)/pcl-$(PCL_VERSION)/pcl/filters/`
- *Binaries*: `$(PCL_PREFIX)/bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix `CMAKE_INSTALL_PREFIX`, e.g., `/usr/local/`

- **Windows**

- Header files: `$(PCL_DIRECTORY)/include/pcl-$(PCL_VERSION)/pcl/filters/`
- *Binaries*: `$(PCL_DIRECTORY)/bin/`
- `$(PCL_DIRECTORY)` is the PCL installation directory, e.g., `C:\Program Files\PCL`  
`$(PCL_VERSION)\`

*Top*

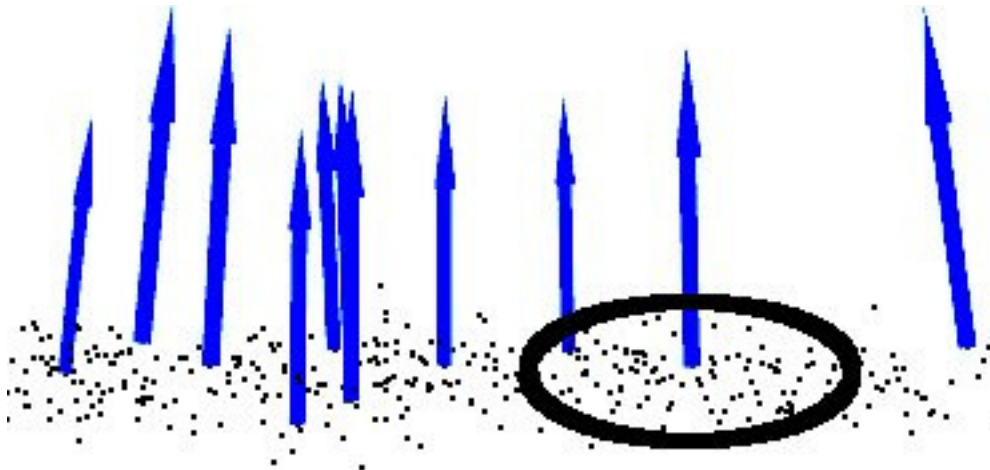
## 1.3 Features

### Background

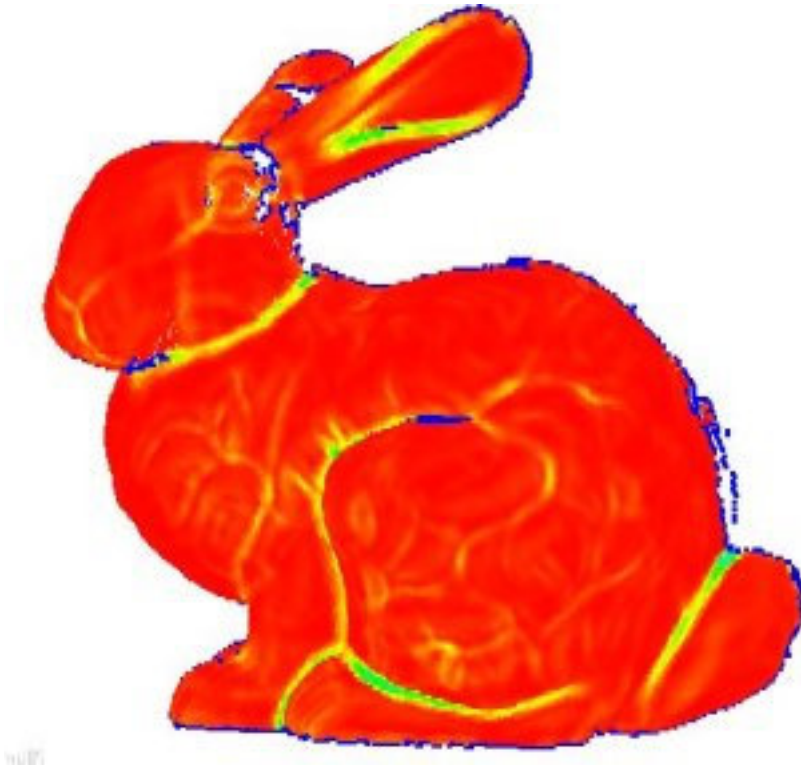
A theoretical primer explaining how features work in PCL can be found in the [3D Features tutorial](#).

The *features* library contains data structures and mechanisms for 3D feature estimation from point cloud data. 3D features are representations at certain 3D points, or positions, in space, which describe geometrical patterns based on the information available around the point. The data space selected around the query point is usually referred to as the *k-neighborhood*.

The following figure shows a simple example of a selected query point, and its selected k-neighborhood.



An example of two of the most widely used geometric point features are the underlying surface's estimated curvature and normal at a query point  $p$ . Both of them are considered local features, as they characterize a point using the information provided by its  $k$  closest point neighbors. For determining these neighbors efficiently, the input dataset is usually split into smaller chunks using spatial decomposition techniques such as octrees or kD-trees, and then closest point searches are performed in that space. Depending on the application one can opt for either determining a fixed number of  $k$  points in the vicinity of  $p$ , or all points which are found inside of a sphere of radius  $r$  centered at  $p$ . Unarguably, one of the easiest methods for estimating the surface normals and curvature changes at a point  $p$  is to perform an eigendecomposition (i.e., compute the eigenvectors and eigenvalues) of the  $k$ -neighborhood point surface patch. Thus, the eigenvector corresponding to the smallest eigenvalue will approximate the surface normal  $n$  at point  $p$ , while the surface curvature change will be estimated from the eigenvalues as  $\frac{\lambda_0}{\lambda_0 + \lambda_1 + \lambda_2}$  with  $\lambda_0 < \lambda_1 < \lambda_2$ .



**Documentation:** [http://docs.pointclouds.org/trunk/group\\_\\_features.html](http://docs.pointclouds.org/trunk/group__features.html)

**Tutorials:** <http://pointclouds.org/documentation/tutorials/#features-tutorial>

**Interacts with:**

- *Common*
- *Search*
- *KdTree*
- *Octree*
- *Range Image*

**Location:**

- **MAC OS X (Homebrew installation)**

- Header files: `$(PCL_PREFIX)/pcl-$(PCL_VERSION)/pcl/features/`
- *Binaries:* `$(PCL_PREFIX)/bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix CMAKE\_INSTALL\_PREFIX, e.g., `/usr/local/`

- **Linux**

- Header files: `$(PCL_PREFIX)/pcl-$(PCL_VERSION)/pcl/filters/`
- *Binaries:* `$(PCL_PREFIX)/bin/`

- `$(PCL_PREFIX)` is the cmake installation prefix `CMAKE_INSTALL_PREFIX`, e.g., `/usr/local/`

- **Windows**

- **Header files:** `$(PCL_DIRECTORY)/include/pcl-$(PCL_VERSION)/pcl/features/`
- *Binaries:* `$(PCL_DIRECTORY)/bin/`
- `$(PCL_DIRECTORY)` is the PCL installation directory, e.g., `C:\Program Files\PCL $(PCL_VERSION)\`

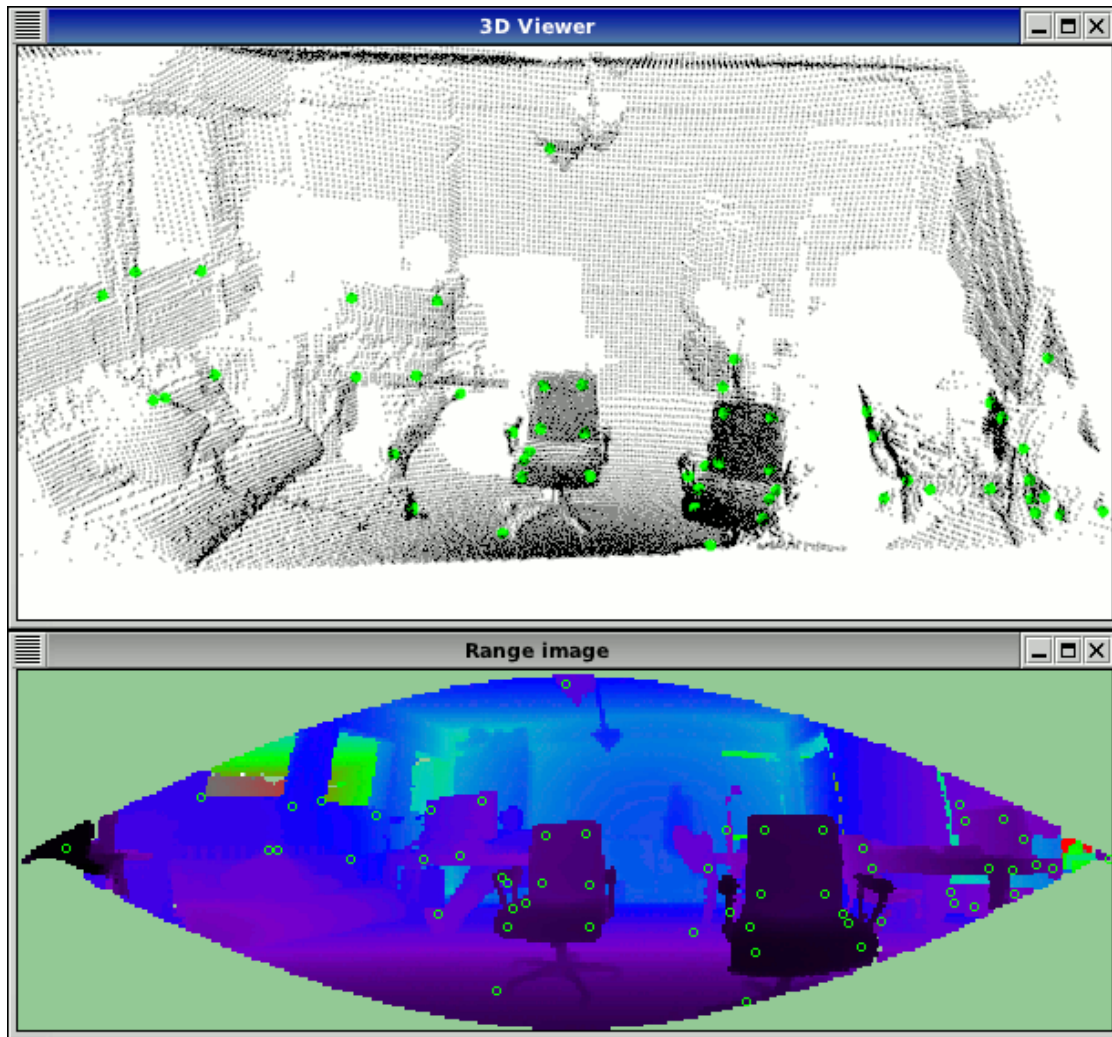
*Top*

## 1.4 Keypoints

### Background

The *keypoints* library contains implementations of two point cloud keypoint detection algorithms. Keypoints (also referred to as *interest points*) are points in an image or point cloud that are stable, distinctive, and can be identified using a well-defined detection criterion. Typically, the number of interest points in a point cloud will be much smaller than the total number of points in the cloud, and when used in combination with local feature descriptors at each keypoint, the keypoints and descriptors can be used to form a compact—yet descriptive—representation of the original data.

The figure below shows the output of NARF keypoints extraction from a range image:



**Documentation:** [http://docs.pointclouds.org/trunk/group\\_\\_keypoints.html](http://docs.pointclouds.org/trunk/group__keypoints.html)

**Tutorials:** <http://pointclouds.org/documentation/tutorials/#keypoints-tutorial>

**Interacts with:**

- *Common*
- *Search*
- *KdTree*
- *Octree*
- *Range Image*
- *Features*
- *Filters*

**Location:**

- **MAC OS X (Homebrew installation)**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/keypoints/`
- *Binaries*: `$(PCL_PREFIX) /bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix `CMAKE_INSTALL_PREFIX`, e.g., `/usr/local/`

- **Linux**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/filters/`
- *Binaries*: `$(PCL_PREFIX) /bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix `CMAKE_INSTALL_PREFIX`, e.g., `/usr/local/`

- **Windows**

- Header files: `$(PCL_DIRECTORY) /include/pcl-$(PCL_VERSION) /pcl/keypoints/`
- *Binaries*: `$(PCL_DIRECTORY) /bin/`
- `$(PCL_DIRECTORY)` is the PCL installation directory, e.g., `C:\Program Files\PCL $(PCL_VERSION) \`

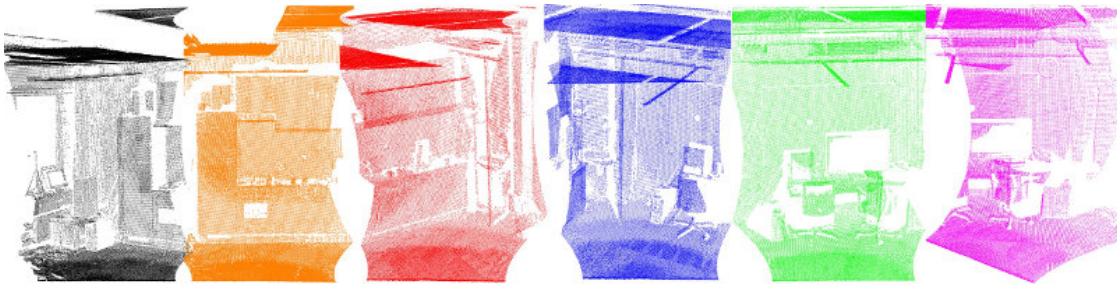
*Top*

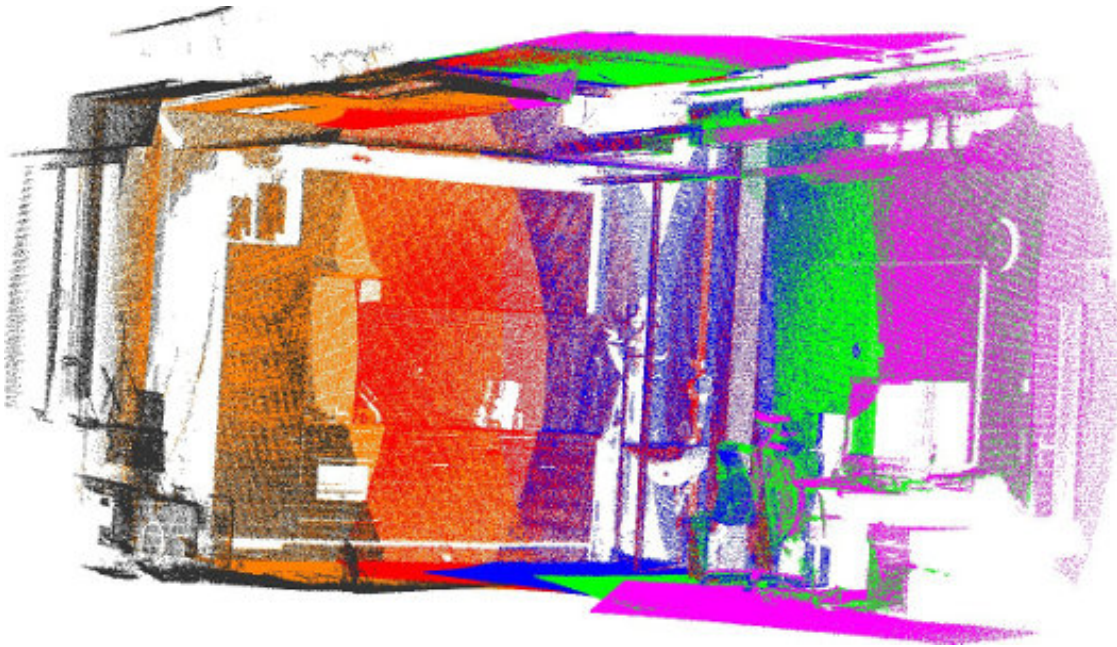
## 1.5 Registration

### Background

Combining several datasets into a global consistent model is usually performed using a technique called registration. The key idea is to identify corresponding points between the data sets and find a transformation that minimizes the distance (alignment error) between corresponding points. This process is repeated, since correspondence search is affected by the relative position and orientation of the data sets. Once the alignment errors fall below a given threshold, the registration is said to be complete.

The *registration* library implements a plethora of point cloud registration algorithms for both organized and unorganized (general purpose) datasets. For instance, PCL contains a set of powerful algorithms that allow the estimation of multiple sets of correspondences, as well as methods for rejecting bad correspondences, and estimating transformations in a robust manner.





**Documentation:** [http://docs.pointclouds.org/trunk/group\\_\\_registration.html](http://docs.pointclouds.org/trunk/group__registration.html)

**Tutorials:** <http://pointclouds.org/documentation/tutorials/#registration-tutorial>

**Interacts with:**

- *Common*
- *KdTree*
- *Sample Consensus*
- *Features*

**Location:**

- **MAC OS X (Homebrew installation)**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/registration/`
- *Binaries:* `$(PCL_PREFIX) /bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix `CMAKE_INSTALL_PREFIX`, e.g., `/usr/local/`

- **Linux**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/filters/`
- *Binaries:* `$(PCL_PREFIX) /bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix `CMAKE_INSTALL_PREFIX`, e.g., `/usr/local/`

- **Windows**

- Header files: `$(PCL_DIRECTORY)/include/pcl-$(PCL_VERSION)/pcl/registration/`
- *Binaries*: `$(PCL_DIRECTORY)/bin/`
- `$(PCL_DIRECTORY)` is the PCL installation directory, e.g., `C:\Program Files\PCL\$(PCL_VERSION)\`

*Top*

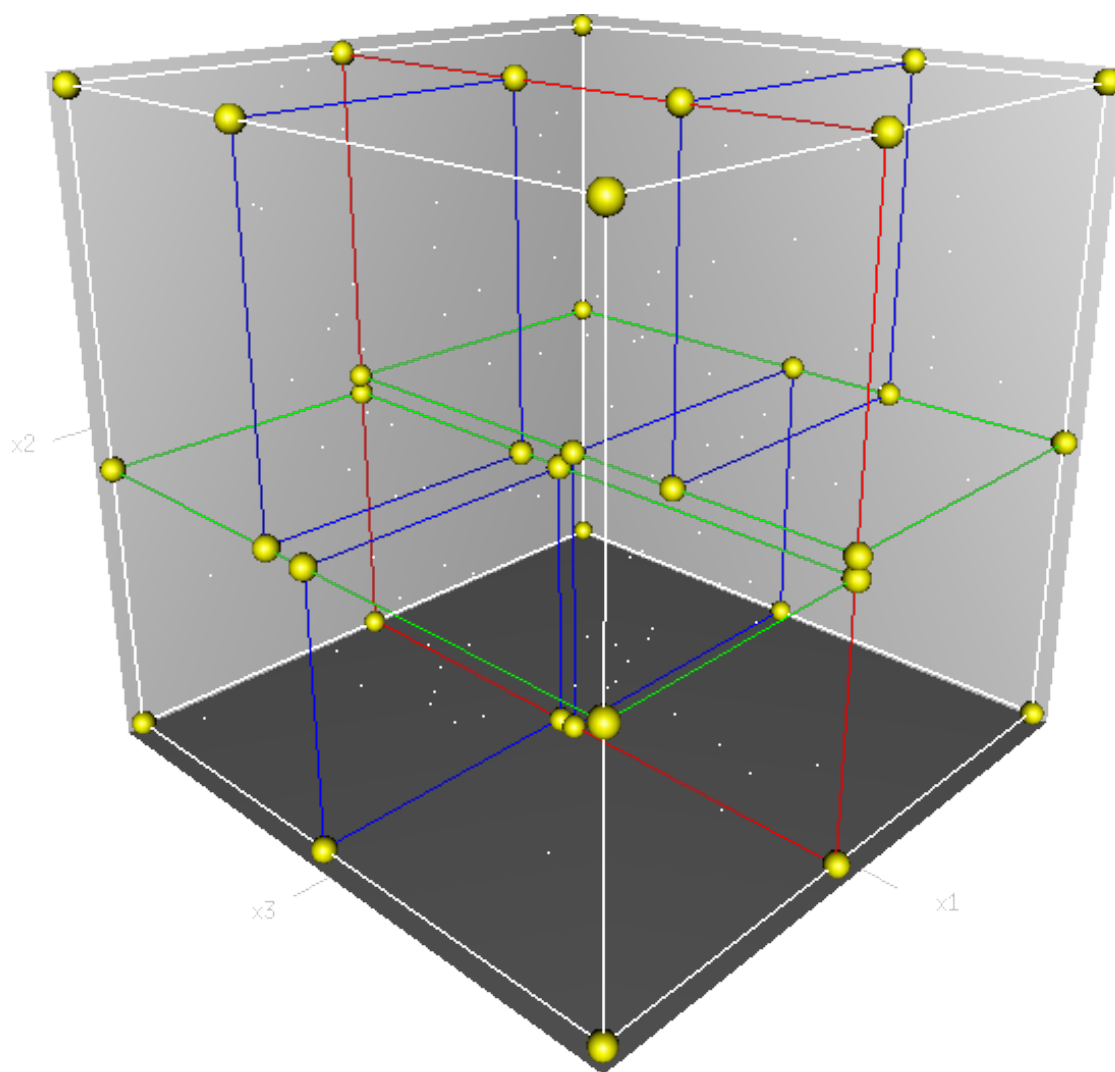
## 1.6 Kd-tree

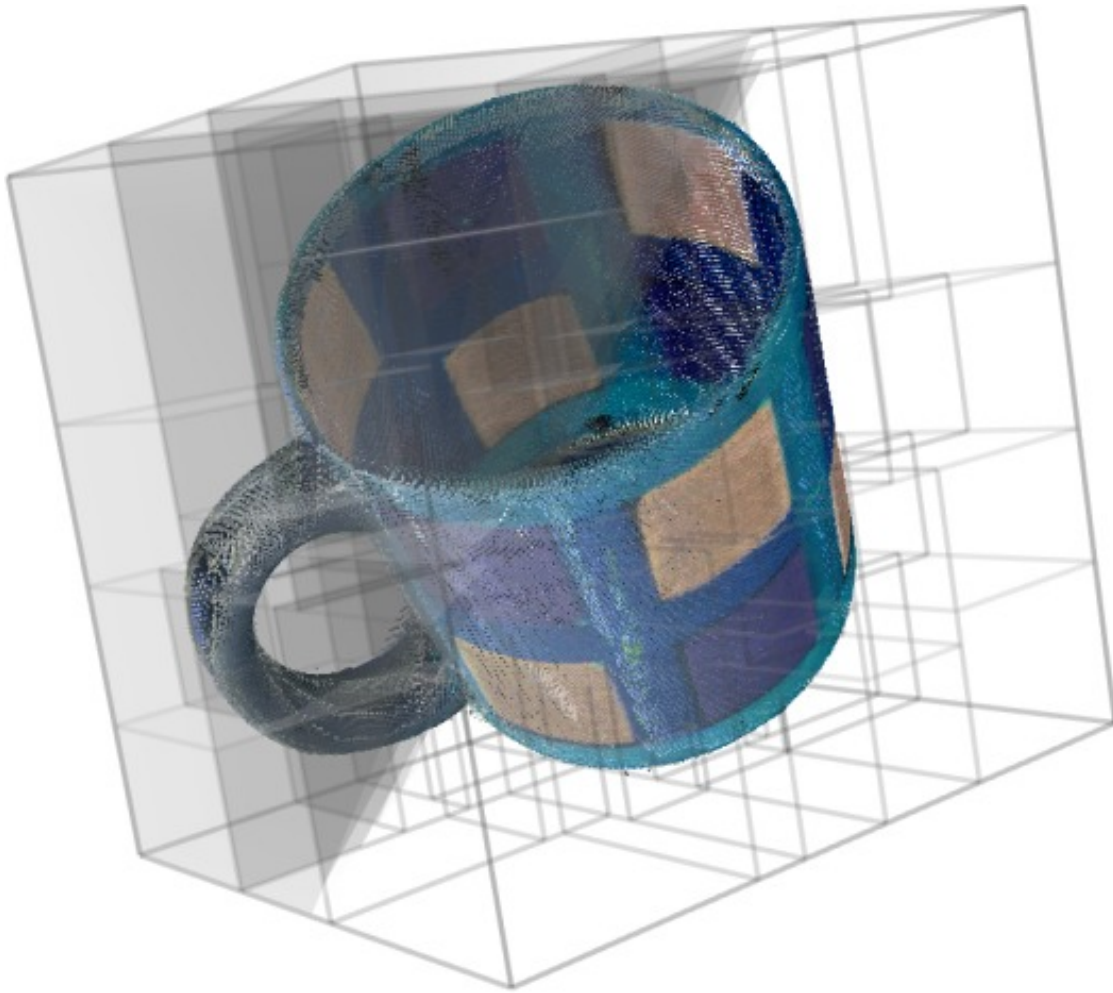
### Background

A theoretical primer explaining how Kd-trees work can be found in the [Kd-tree tutorial](#).

The *kdtree* library provides the kd-tree data-structure, using [FLANN](#), that allows for fast [nearest neighbor searches](#).

A [Kd-tree](#) (k-dimensional tree) is a space-partitioning data structure that stores a set of k-dimensional points in a tree structure that enables efficient range searches and nearest neighbor searches. Nearest neighbor searches are a core operation when working with point cloud data and can be used to find correspondences between groups of points or feature descriptors or to define the local neighborhood around a point or points.





**Documentation:** [http://docs.pointclouds.org/trunk/group\\_\\_kdtree.html](http://docs.pointclouds.org/trunk/group__kdtree.html)

**Tutorials:** <http://pointclouds.org/documentation/tutorials/#kdtree-tutorial>

**Interacts with:** *Common*

**Location:**

- **MAC OS X (Homebrew installation)**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/kdtree/`
- *Binaries*: `$(PCL_PREFIX) /bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix `CMAKE_INSTALL_PREFIX`, e.g., `/usr/local/`

- **Linux**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/filters/`
- *Binaries*: `$(PCL_PREFIX) /bin/`

- `$(PCL_PREFIX)` is the cmake installation prefix `CMAKE_INSTALL_PREFIX`, e.g., `/usr/local/`

- **Windows**

- **Header files:** `$(PCL_DIRECTORY)/include/pcl-$(PCL_VERSION)/pcl/kdtree/`
- *Binaries:* `$(PCL_DIRECTORY)/bin/`
- `$(PCL_DIRECTORY)` is the PCL installation directory, e.g., `C:\Program Files\PCL $(PCL_VERSION)\`

*Top*

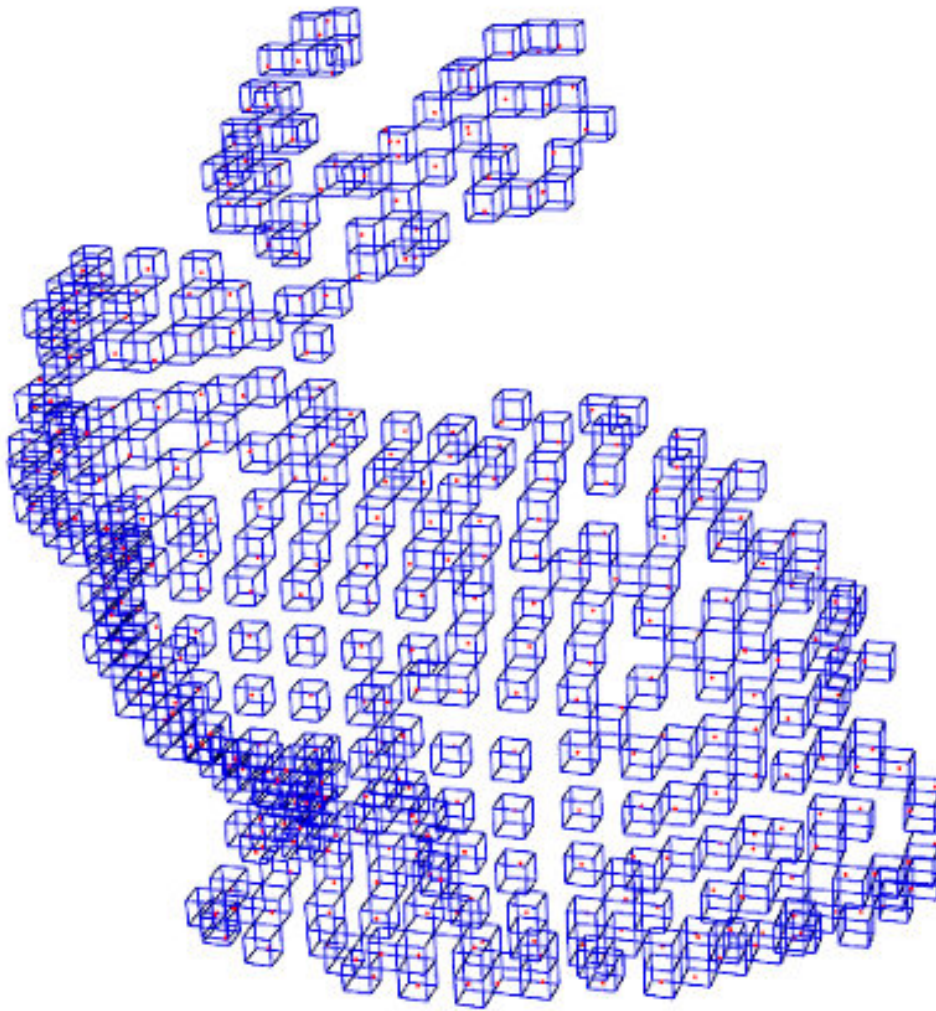
## 1.7 Octree

### Background

The *octree* library provides efficient methods for creating a hierarchical tree data structure from point cloud data. This enables spatial partitioning, downsampling and search operations on the point data set. Each octree node has either eight children or no children. The root node describes a cubic bounding box which encapsulates all points. At every tree level, this space becomes subdivided by a factor of 2 which results in an increased voxel resolution.

The *octree* implementation provides efficient nearest neighbor search routines, such as “Neighbors within Voxel Search”, “K Nearest Neighbor Search” and “Neighbors within Radius Search”. It automatically adjusts its dimension to the point data set. A set of leaf node classes provide additional functionality, such as spacial “occupancy” and “point density per voxel” checks. Functions for serialization and deserialization enable to efficiently encode the octree structure into a binary format. Furthermore, a memory pool implementation reduces expensive memory allocation and deallocation operations in scenarios where octrees needs to be created at high rate.

The following figure illustrates the voxel bounding boxes of an octree nodes at lowest tree level. The octree voxels are surrounding every 3D point from the Stanford bunny’s surface. The red dots represent the point data. This image is created with the *octree\_viewer*.



**Documentation:** [http://docs.pointclouds.org/trunk/group\\_\\_octree.html](http://docs.pointclouds.org/trunk/group__octree.html)

**Tutorials:** <http://pointclouds.org/documentation/tutorials/#octree-tutorial>

**Interacts with:** *Common*

**Location:**

- **MAC OS X (Homebrew installation)**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/octree/`
- *Binaries*: `$(PCL_PREFIX) /bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix CMAKE\_INSTALL\_PREFIX, e.g., `/usr/local/`

- **Linux**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/filters/`
- *Binaries*: `$(PCL_PREFIX) /bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix CMAKE\_INSTALL\_PREFIX, e.g., `/usr/local/`

- **Windows**

- Header files: `$(PCL_DIRECTORY) /include/pcl-$(PCL_VERSION) /pcl/octree/`
- *Binaries*: `$(PCL_DIRECTORY) /bin/`
- `$(PCL_DIRECTORY)` is the PCL installation directory, e.g., `C:\Program Files\PCL $(PCL_VERSION) \`

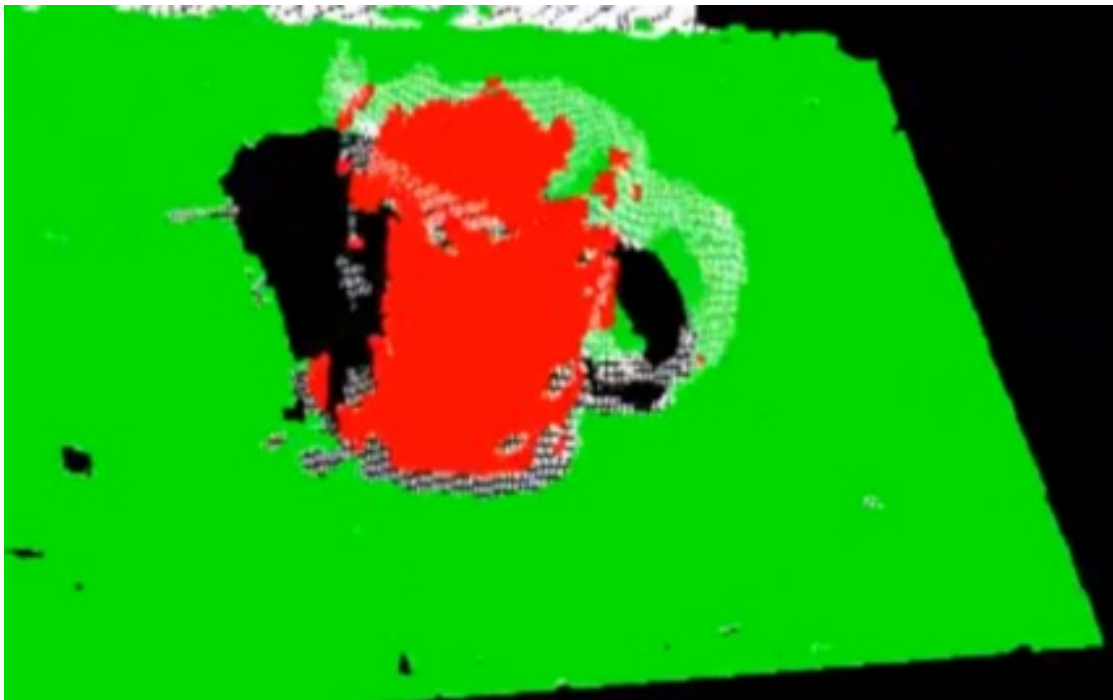
*Top*

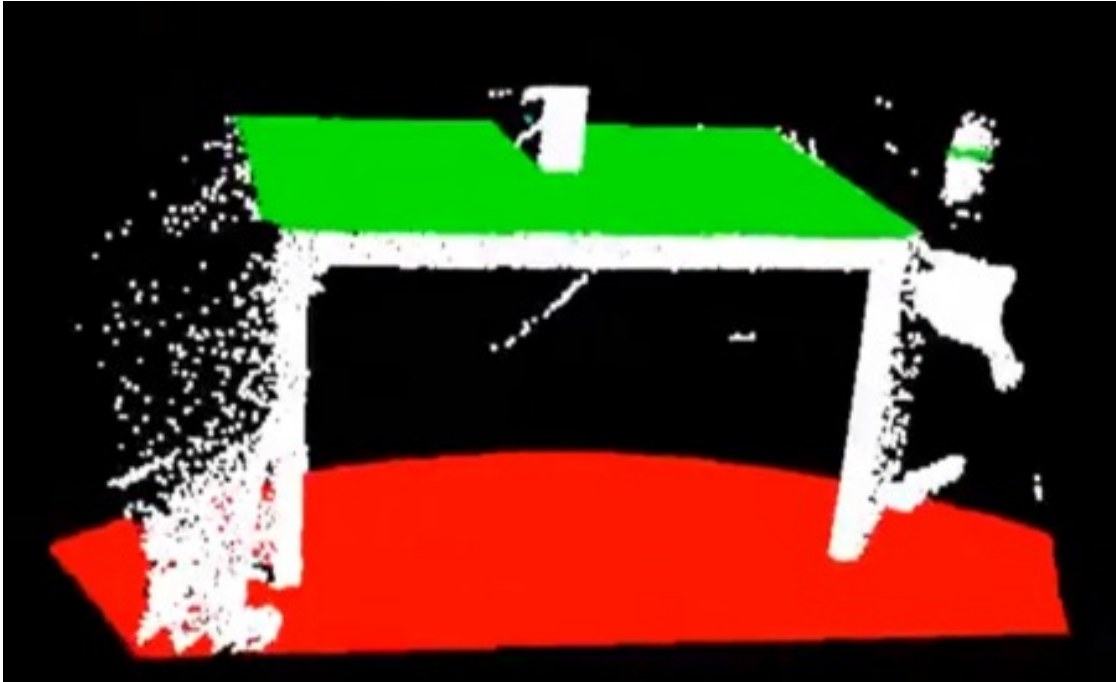
## 1.8 Segmentation

### Background

The *segmentation* library contains algorithms for segmenting a point cloud into distinct clusters. These algorithms are best suited for processing a point cloud that is composed of a number of spatially isolated regions. In such cases, clustering is often used to break the cloud down into its constituent parts, which can then be processed independently.

A theoretical primer explaining how clustering methods work can be found in the [cluster extraction tutorial](#). The two figures illustrate the results of plane model segmentation (left) and cylinder model segmentation (right).





**Documentation:** [http://docs.pointclouds.org/trunk/group\\_\\_segmentation.html](http://docs.pointclouds.org/trunk/group__segmentation.html)

**Tutorials:** <http://pointclouds.org/documentation/tutorials/#segmentation-tutorial>

**Interacts with:**

- *Common*
- *Search*
- *Sample Consensus*
- *KdTree*
- *Octree*

**Location:**

- **MAC OS X (Homebrew installation)**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/segmentation/`
- *Binaries*: `$(PCL_PREFIX) /bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix `CMAKE_INSTALL_PREFIX`, e.g., `/usr/local/`

- **Linux**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/filters/`
- *Binaries*: `$(PCL_PREFIX) /bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix `CMAKE_INSTALL_PREFIX`, e.g., `/usr/local/`

- Windows

- Header files: `$(PCL_DIRECTORY)/include/pcl-$(PCL_VERSION)/pcl/segmentation/`
- Binaries: `$(PCL_DIRECTORY)/bin/`
- `$(PCL_DIRECTORY)` is the PCL installation directory, e.g., `C:\Program Files\PCL $(PCL_VERSION) \`

*Top*

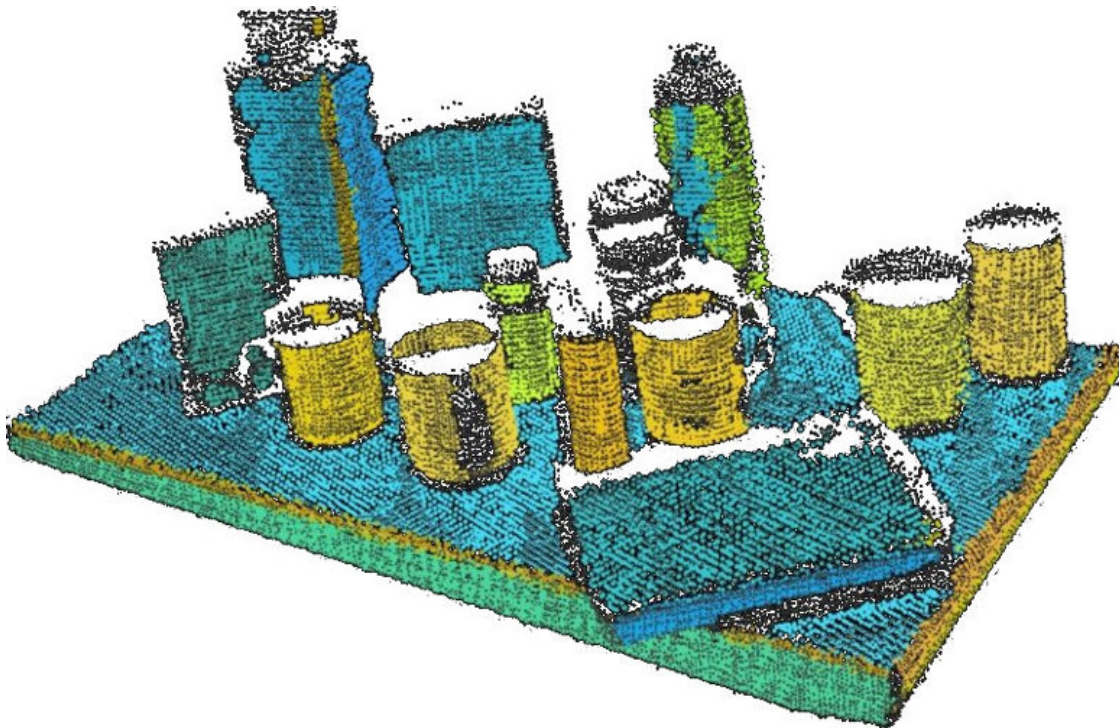
## 1.9 Sample Consensus

### Background

The *sample\_consensus* library holds SAmple Consensus (SAC) methods like RANSAC and models like planes and cylinders. These can be combined freely in order to detect specific models and their parameters in point clouds.

A theoretical primer explaining how sample consensus algorithms work can be found in the [Random Sample Consensus tutorial](#)

Some of the models implemented in this library include: lines, planes, cylinders, and spheres. Plane fitting is often applied to the task of detecting common indoor surfaces, such as walls, floors, and table tops. Other models can be used to detect and segment objects with common geometric structures (e.g., fitting a cylinder model to a mug).



**Documentation:** [http://docs.pointclouds.org/trunk/group\\_\\_sample\\_\\_consensus.html](http://docs.pointclouds.org/trunk/group__sample__consensus.html)

**Tutorials:** <http://pointclouds.org/documentation/tutorials/#sample-consensus>

**Interacts with:** *Common*

**Location:**

- **MAC OS X (Homebrew installation)**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/sample_consensus/`
- *Binaries:* `$(PCL_PREFIX) /bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix CMAKE\_INSTALL\_PREFIX, e.g., `/usr/local/`

- **Linux**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/filters/`
- *Binaries:* `$(PCL_PREFIX) /bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix CMAKE\_INSTALL\_PREFIX, e.g., `/usr/local/`

- **Windows**

- Header files: `$(PCL_DIRECTORY) /include/pcl-$(PCL_VERSION) /pcl/sample_consensus/`
- *Binaries:* `$(PCL_DIRECTORY) /bin/`
- `$(PCL_DIRECTORY)` is the PCL installation directory, e.g., `C:\Program Files\PCL $(PCL_VERSION) \`

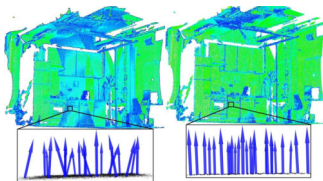
*Top*

## 1.10 Surface

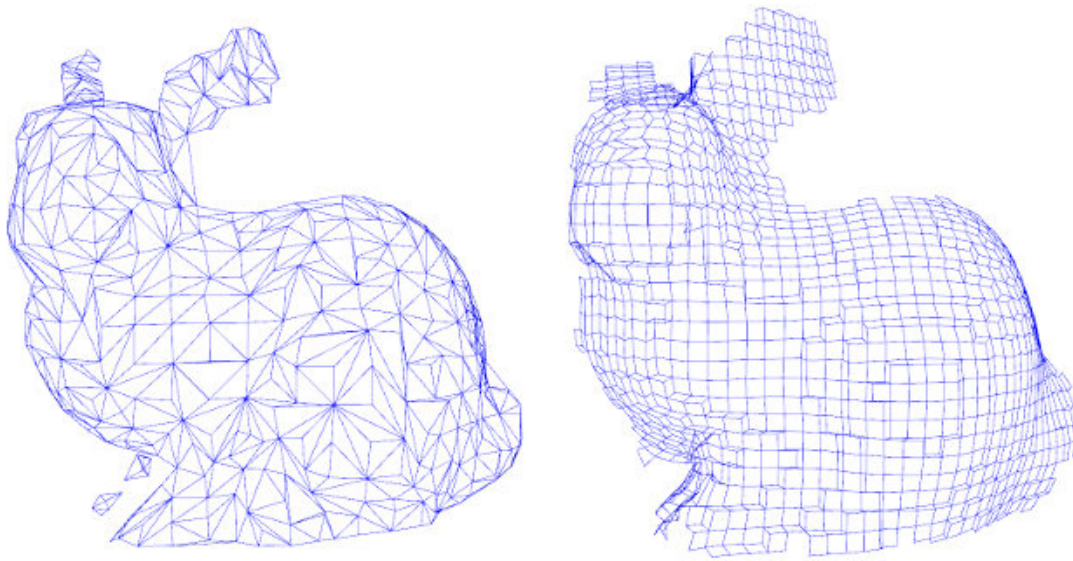
### Background

The *surface* library deals with reconstructing the original surfaces from 3D scans. Depending on the task at hand, this can be for example the hull, a mesh representation or a smoothed/resampled surface with normals.

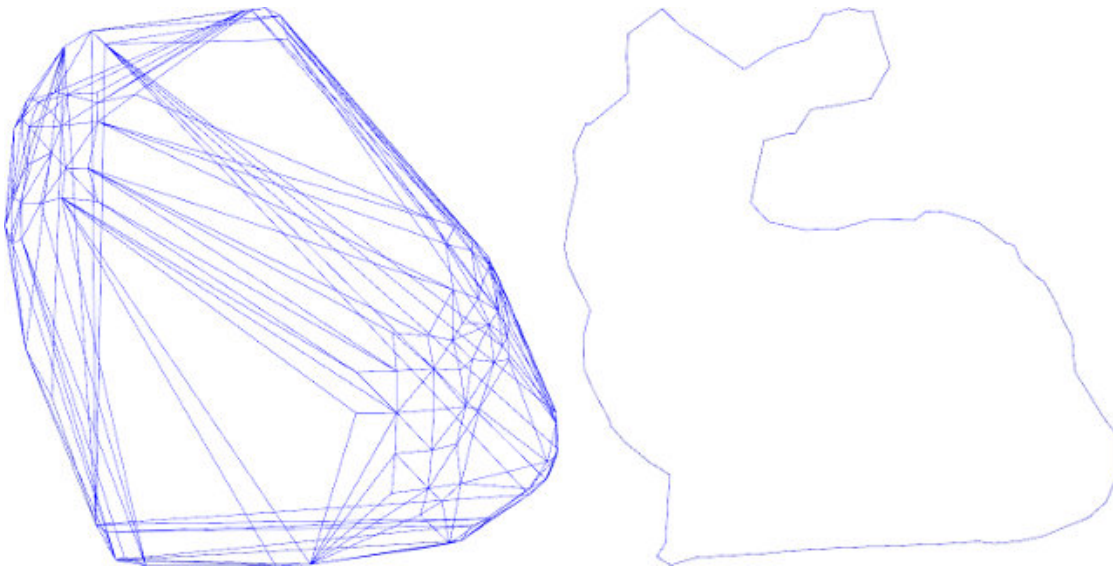
Smoothing and resampling can be important if the cloud is noisy, or if it is composed of multiple scans that are not aligned perfectly. The complexity of the surface estimation can be adjusted, and normals can be estimated in the same step if needed.



Meshing is a general way to create a surface out of points, and currently there are two algorithms provided: a very fast triangulation of the original points, and a slower meshing that does smoothing and hole filling as well.



Creating a convex or concave hull is useful for example when there is a need for a simplified surface representation or when boundaries need to be extracted.



**Documentation:** [http://docs.pointclouds.org/trunk/group\\_\\_surface.html](http://docs.pointclouds.org/trunk/group__surface.html)

**Tutorials:** <http://pointclouds.org/documentation/tutorials/#surface-tutorial>

**Interacts with:**

- *Common*
- *Search*
- *KdTree*
- *Octree*

**Location:**• **MAC OS X (Homebrew installation)**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/surface/`
- *Binaries*: `$(PCL_PREFIX) /bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix `CMAKE_INSTALL_PREFIX`, e.g., `/usr/local/`

• **Linux**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/filters/`
- *Binaries*: `$(PCL_PREFIX) /bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix `CMAKE_INSTALL_PREFIX`, e.g., `/usr/local/`

• **Windows**

- Header files: `$(PCL_DIRECTORY) /include/pcl-$(PCL_VERSION) /pcl/surface/`
- *Binaries*: `$(PCL_DIRECTORY) /bin/`
- `$(PCL_DIRECTORY)` is the PCL installation directory, e.g., `C:\Program Files\PCL $(PCL_VERSION) \`

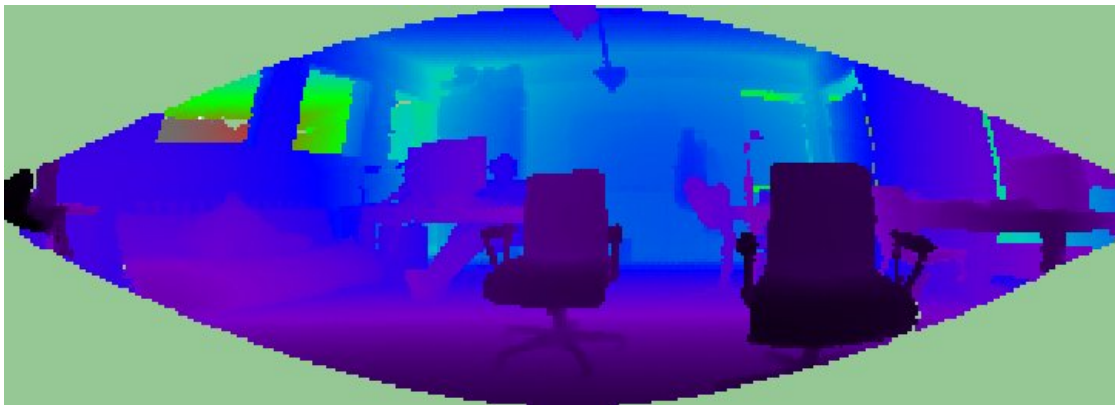
*Top*

## 1.11 Range Image

**Background**

The *range\_image* library contains two classes for representing and working with range images. A range image (or depth map) is an image whose pixel values represent a distance or depth from the sensor's origin. Range images are a common 3D representation and are often generated by stereo or time-of-flight cameras. With knowledge of the camera's intrinsic calibration parameters, a range image can be converted into a point cloud.

Note: *range\_image* is now a part of *Common* module.



**Tutorials:** <http://pointclouds.org/documentation/tutorials/#range-images>

**Interacts with:** *Common*

**Location:**

- **MAC OS X (Homebrew installation)**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/range_image/`
- *Binaries*: `$(PCL_PREFIX) /bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix CMAKE\_INSTALL\_PREFIX, e.g., `/usr/local/`

- **Linux**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/filters/`
- *Binaries*: `$(PCL_PREFIX) /bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix CMAKE\_INSTALL\_PREFIX, e.g., `/usr/local/`

- **Windows**

- Header files: `$(PCL_DIRECTORY) /include/pcl-$(PCL_VERSION) /pcl/range_image/`
- *Binaries*: `$(PCL_DIRECTORY) /bin/`
- `$(PCL_DIRECTORY)` is the PCL installation directory, e.g., `C:\Program Files\PCL $(PCL_VERSION) \`

*Top*

## 1.12 I/O

### Background

The *io* library contains classes and functions for reading and writing point cloud data (PCD) files, as well as capturing point clouds from a variety of sensing devices. An introduction to some of these capabilities can be found in the following tutorials:

- The PCD (Point Cloud Data) file format
- Reading PointCloud data from PCD files
- Writing PointCloud data to PCD files
- The OpenNI Grabber Framework in PCL

**Documentation:** [http://docs.pointclouds.org/trunk/group\\_\\_io.html](http://docs.pointclouds.org/trunk/group__io.html)

**Tutorials:** <http://pointclouds.org/documentation/tutorials/#i-o>

**Interacts with:**

- *Common*

- *Octree*
- OpenNI for kinect handling

**Location:**

- **MAC OS X (Homebrew installation)**

- Header files: `$(PCL_PREFIX)/pcl-$(PCL_VERSION)/pcl/io/`
- *Binaries*: `$(PCL_PREFIX)/bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix CMAKE\_INSTALL\_PREFIX, e.g., `/usr/local/`

- **Linux**

- Header files: `$(PCL_PREFIX)/pcl-$(PCL_VERSION)/pcl/filters/`
- *Binaries*: `$(PCL_PREFIX)/bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix CMAKE\_INSTALL\_PREFIX, e.g., `/usr/local/`

- **Windows**

- Header files: `$(PCL_DIRECTORY)/include/pcl-$(PCL_VERSION)/pcl/io/`
- *Binaries*: `$(PCL_DIRECTORY)/bin/`
- `$(PCL_DIRECTORY)` is the PCL installation directory, e.g., `C:\Program Files\PCL $(PCL_VERSION) \`

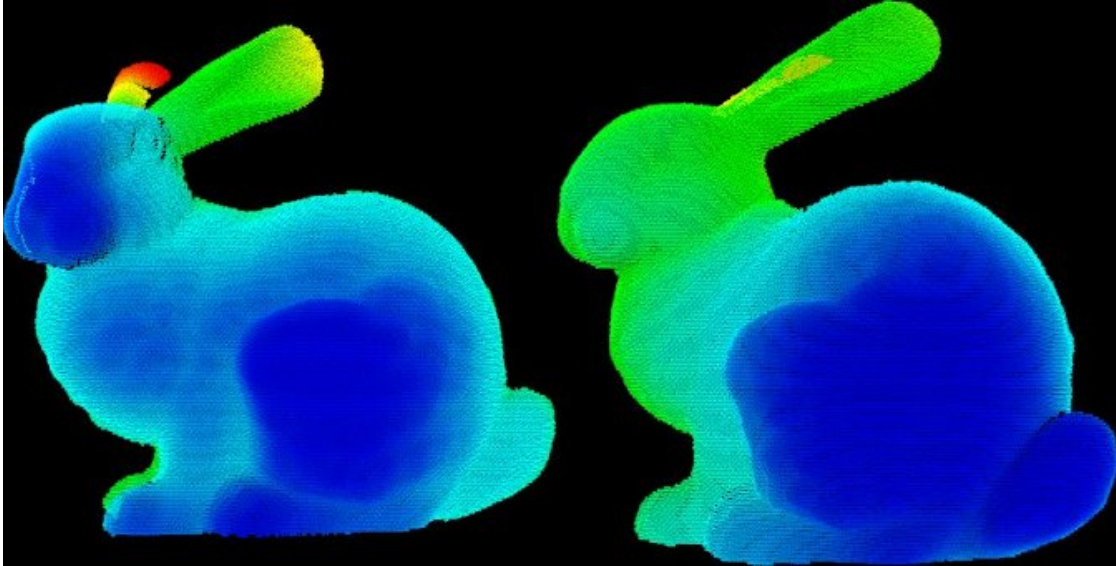
*Top*

## 1.13 Visualization

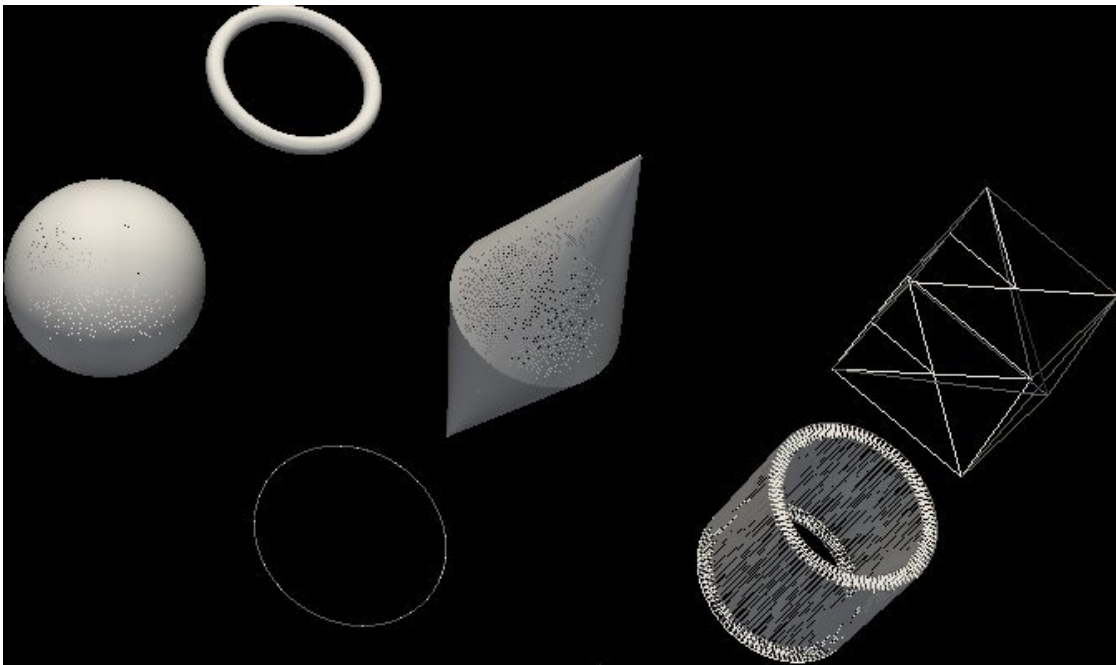
### Background

The *visualization* library was built for the purpose of being able to quickly prototype and visualize the results of algorithms operating on 3D point cloud data. Similar to OpenCV's *highgui* routines for displaying 2D images and for drawing basic 2D shapes on screen, the library offers:

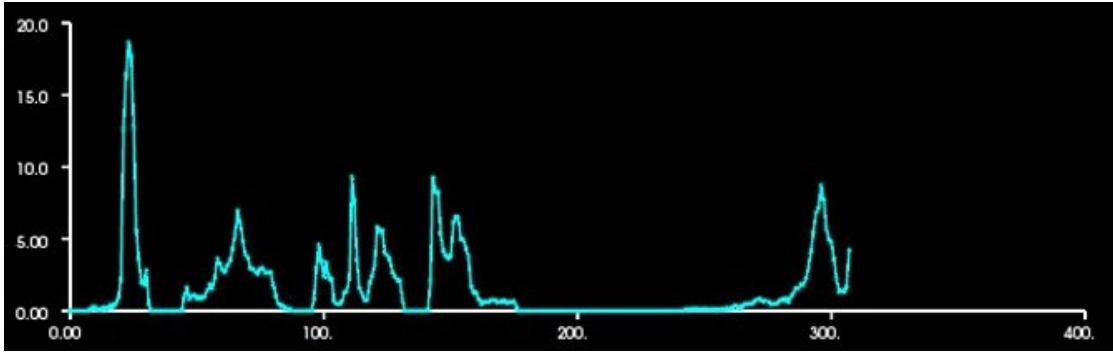
methods for rendering and setting visual properties (colors, point sizes, opacity, etc) for any n-D point cloud datasets in `pcl::PointCloud<T>` format;



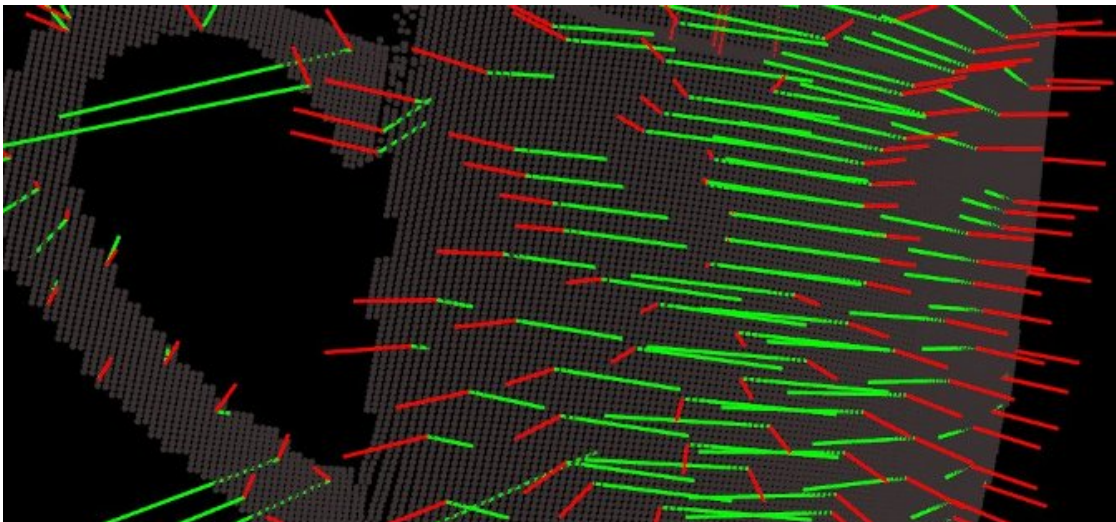
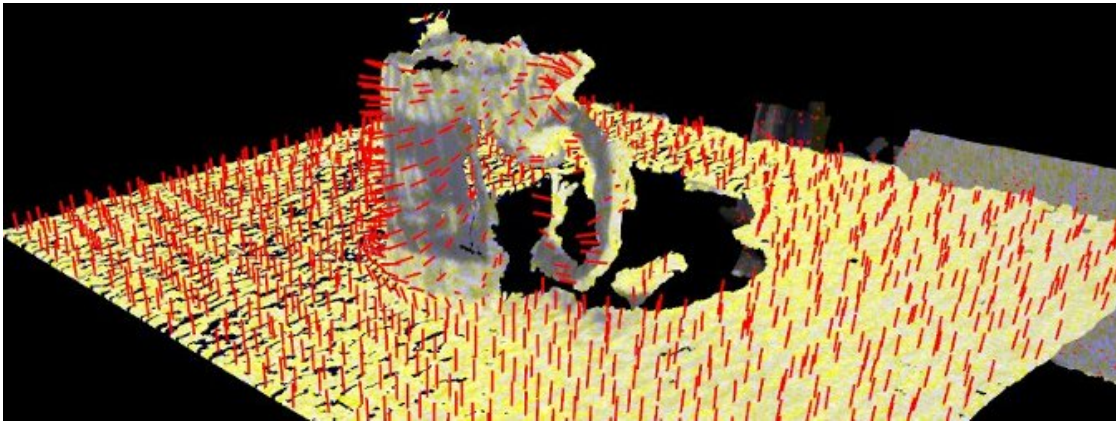
methods for drawing basic 3D shapes on screen (e.g., cylinders, spheres, lines, polygons, etc) either from sets of points or from parametric equations;



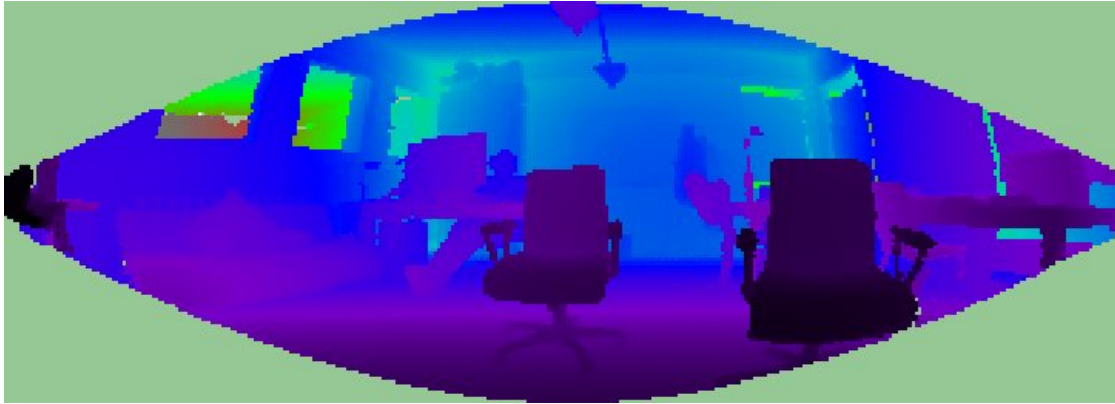
a histogram visualization module (PCLHistogramVisualizer) for 2D plots;



a multitude of Geometry and Color handlers for `pcl::PointCloud<T>` datasets;



a `pcl::RangeImage` visualization module.



The package makes use of the VTK library for 3D rendering for range image and 2D operations.

For implementing your own visualizers, take a look at the tests and examples accompanying the library.

**Documentation:** [http://docs.pointclouds.org/trunk/group\\_\\_visualization.html](http://docs.pointclouds.org/trunk/group__visualization.html)

**Tutorials:** <http://pointclouds.org/documentation/tutorials/#visualization-tutorial>

**Interacts with:**

- *Common*
- *I/O*
- *KdTree*
- *Range Image*
- VTK

**Location:**

- **MAC OS X (Homebrew installation)**

- Header files: `$(PCL_PREFIX)/pcl-$(PCL_VERSION)/pcl/visualization/`
- *Binaries*: `$(PCL_PREFIX)/bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix CMAKE\_INSTALL\_PREFIX, e.g., `/usr/local/`

- **Linux**

- Header files: `$(PCL_PREFIX)/pcl-$(PCL_VERSION)/pcl/filters/`
- *Binaries*: `$(PCL_PREFIX)/bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix CMAKE\_INSTALL\_PREFIX, e.g., `/usr/local/`

- **Windows**

- Header files: `$(PCL_DIRECTORY)/include/pcl-$(PCL_VERSION)/pcl/visualization/`
- *Binaries*: `$(PCL_DIRECTORY)/bin/`

- `$(PCL_DIRECTORY)` is the PCL installation directory, e.g., `C:\Program Files\PCL`  
`$(PCL_VERSION) \`

[Top](#)

## 1.14 Common

### Background

The *common* library contains the common data structures and methods used by the majority of PCL libraries. The core data structures include the `PointCloud` class and a multitude of point types that are used to represent points, surface normals, RGB color values, feature descriptors, etc. It also contains numerous functions for computing distances/norms, means and covariances, angular conversions, geometric transformations, and more.

#### Location:

- **MAC OS X (Homebrew installation)**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/common/`
- *Binaries*: `$(PCL_PREFIX) /bin/`
- `$(PCL_PREFIX)` is the `cmake` installation prefix `CMAKE_INSTALL_PREFIX`, e.g., `/usr/local/`

- **Linux**

- Header files: `$(PCL_PREFIX) /pcl-$(PCL_VERSION) /pcl/common/`
- *Binaries*: `$(PCL_PREFIX) /bin/`
- `$(PCL_PREFIX)` is the `cmake` installation prefix `CMAKE_INSTALL_PREFIX`, e.g., `/usr/local/`

- **Windows**

- Header files: `$(PCL_DIRECTORY) /include/pcl-$(PCL_VERSION) /pcl/common/`
- *Binaries*: `$(PCL_DIRECTORY) /bin/`
- `$(PCL_DIRECTORY)` is the PCL installation directory, e.g., `C:\Program Files\PCL`  
`$(PCL_VERSION) \`

[Top](#)

## 1.15 Search

### Background

The *search* library provides methods for searching for nearest neighbors using different data structures, including:

- *KdTree*
- *Octree*
- brute force
- specialized search for organized datasets

**Interacts with:**

- *Common*
- *Kdtree*
- *Octree*

**Location:**

- **MAC OS X (Homebrew installation)**

- Header files: `$(PCL_PREFIX)/pcl-$(PCL_VERSION)/pcl/search/`
- *Binaries*: `$(PCL_PREFIX)/bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix CMAKE\_INSTALL\_PREFIX, e.g., `/usr/local/`

- **Linux**

- Header files: `$(PCL_PREFIX)/pcl-$(PCL_VERSION)/pcl/search/`
- *Binaries*: `$(PCL_PREFIX)/bin/`
- `$(PCL_PREFIX)` is the cmake installation prefix CMAKE\_INSTALL\_PREFIX, e.g., `/usr/local/`

- **Windows**

- Header files: `$(PCL_DIRECTORY)/include/pcl-$(PCL_VERSION)/pcl/search/`
- *Binaries*: `$(PCL_DIRECTORY)/bin/`
- `$(PCL_DIRECTORY)` is the PCL installation directory, e.g., `C:\Program Files\PCL $(PCL_VERSION) \`

*Top*

## 1.16 Binaries

This section provides a quick reference for some of the common tools in PCL.

- `pcl_viewer`: a quick way for visualizing PCD (Point Cloud Data) files. More information about PCD files can be found in the [PCD file format tutorial](#).

**Syntax is:** `pcl_viewer <file_name 1..N>.<pcd or vtk> <options>`, where options are:

- bc r,g,b = background color
- fc r,g,b = foreground color
- ps X = point size (1..64)
- opaque X = rendered point cloud opacity (0..1)
- ax n = enable on-screen display of XYZ axes and scale them to n
- ax\_pos X,Y,Z = if axes are enabled, set their X,Y,Z position in space (default 0,0,0)
- cam (\*) = use given camera settings as initial view

(\*) [Clipping Range / Focal Point / Position / ViewUp / Distance / Field of View Y / Window Size / Window Pos] or use a <filename.cam> that contains the same information.

-multiview 0/1 = enable/disable auto-multi viewport rendering (default disabled)

-normals 0/X = disable/enable the display of every Xth point's surface normal as lines (default disabled) -normals\_scale X = resize the normal unit vector size to X (default 0.02)

-pc 0/X = disable/enable the display of every Xth point's principal curvatures as lines (default disabled) -pc\_scale X = resize the principal curvatures vectors size to X (default 0.02)

(Note: for multiple .pcd files, provide multiple {-fc,ps,opaque} parameters; they will be automatically assigned to the right file)

#### Usage example:

```
pcl_viewer -multiview 1 data/partial_cup_model.pcd data/
partial_cup_model.pcd data/partial_cup_model.pcd
```

The above will load the partial\_cup\_model.pcd file 3 times, and will create a multi-viewport rendering (-multiview 1).



- `pcd_convert_NaN_nan`: converts “NaN” values to “nan” values. (Note: Starting with PCL version 1.0.1 the string representation for NaN is “nan”.)

#### Usage example:

```
pcd_convert_NaN_nan input.pcd output.pcd
```

- `convert_pcd_ascii_binary`: converts PCD (Point Cloud Data) files from ASCII to binary and viceversa.

#### Usage example:

```
convert_pcd_ascii_binary <file_in.pcd> <file_out.pcd> 0/1/2
(ascii/binary/binary_compressed) [precision (ASCII)]
```

- `concatenate_points_pcd`: concatenates the points of two or more PCD (Point Cloud Data) files into a single PCD file.

#### Usage example:

```
concatenate_points_pcd <filename 1..N.pcd>
```

(Note: the resulting PCD file will be “output.pcd”)

- `pcd2vtk`: converts PCD (Point Cloud Data) files to the [VTK format](#).

**Usage example:**

```
pcd2vtk input.pcd output.vtk
```

- `pcd2ply`: converts PCD (Point Cloud Data) files to the [PLY format](#).

**Usage example:**

```
pcd2ply input.pcd output.ply
```

- `mesh2pcd`: convert a CAD model to a PCD (Point Cloud Data) file, using ray tracing operations.

**Syntax is:** `mesh2pcd input.{ply,obj} output.pcd <options>`, where options are:

-level X = tessellated sphere level (default: 2)

-resolution X = the sphere resolution in angle increments (default: 100 deg)

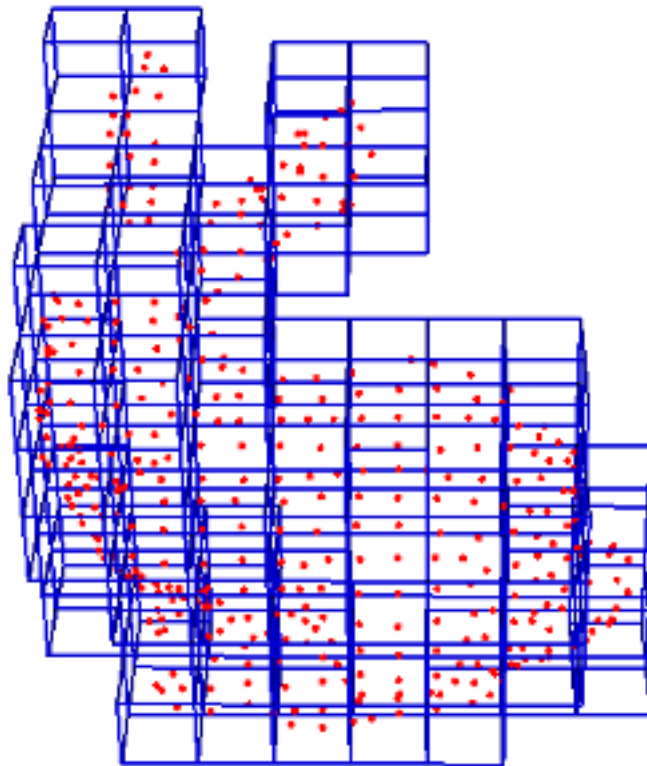
-leaf\_size X = the XYZ leaf size for the VoxelGrid – for data reduction (default: 0.010000 m)

- `octree_viewer`: allows the visualization of *octrees*

**Syntax is:** `octree_viewer <file_name.pcd> <octree resolution>`

**Usage example:**

Example: `./octree_viewer ../../test/bunny.pcd 0.02`



*Top*

---

## Getting Started / Basic Structures

---

The basic data type in PCL 1.x is a `:pcl::PointCloud<pcl::PointXYZ>`. A PointCloud is a C++ class which contains the following data fields:

- `:pcl::PointCloud::width` (int)

Specifies the width of the point cloud dataset in the number of points. *width* has two meanings:

- it can specify the total number of points in the cloud (equal with the number of elements in **points** – see below) for unorganized datasets;
- it can specify the width (total number of points in a row) of an organized point cloud dataset.

---

**Note:** An **organized point cloud** dataset is the name given to point clouds that resemble an organized image (or matrix) like structure, where the data is split into rows and columns. Examples of such point clouds include data coming from stereo cameras or Time Of Flight cameras. The advantages of an organized dataset is that by knowing the relationship between adjacent points (e.g. pixels), nearest neighbor operations are much more efficient, thus speeding up the computation and lowering the costs of certain algorithms in PCL.

---



---

**Note:** An **projectable point cloud** dataset is the name given to point clouds that have a correlation according to a pinhole camera model between the (u,v) index of a point in the organized point cloud and the actual 3D values. This correlation can be expressed in it's easiest form as:  $u = f \cdot x / z$  and  $v = f \cdot y / z$

---

Examples:

```
cloud.width = 640; // there are 640 points per line
```

- `:pcl::PointCloud::height` (int)

Specifies the height of the point cloud dataset in the number of points. *height* has two meanings:

- it can specify the height (total number of rows) of an organized point cloud dataset;
- it is set to **1** for unorganized datasets (*thus used to check whether a dataset is organized or not*).

Example:

```
cloud.width = 640; // Image-like organized structure, with 480 rows and
↳640 columns,
cloud.height = 480; // thus 640*480=307200 points total in the dataset
```

Example:

```
cloud.width = 307200;
cloud.height = 1; // unorganized point cloud dataset with 307200 points
```

- **:pcl::points<pcl::PointCloud::points>** (std::vector<PointT>)

Contains the data array where all the points of type **PointT** are stored. For example, for a cloud containing XYZ data, **points** contains a vector of *pcl::PointXYZ* elements:

```
pcl::PointCloud<pcl::PointXYZ> cloud;
std::vector<pcl::PointXYZ> data = cloud.points;
```

- **:pcl::is\_dense<pcl::PointCloud::is\_dense>** (bool)

Specifies if all the data in **points** is finite (true), or whether the XYZ values of certain points might contain Inf/NaN values (false).

- **:pcl::sensor\_origin\_<pcl::PointCloud::sensor\_origin\_>** (Eigen::Vector4f)

Specifies the sensor acquisition pose (origin/translation). This member is usually optional, and not used by the majority of the algorithms in PCL.

- **:pcl::sensor\_orientation\_<pcl::PointCloud::sensor\_orientation\_>** (Eigen::Quaternionf)

Specifies the sensor acquisition pose (orientation). This member is usually optional, and not used by the majority of the algorithms in PCL.

To simplify development, the **:pcl::PointCloud<pcl::PointCloud>** class contains a number of helper member functions. For example, users don't have to check if **height** equals 1 or not in their code in order to see if a dataset is organized or not, but instead use **:pcl::PointCloud<pcl::PointCloud::isOrganized>**:

```
if (!cloud.isOrganized ())
...
```

The **PointT** type is the primary point data type and describes what each individual element of **:pcl::points<pcl::PointCloud::points>** holds. PCL comes with a large variety of different point types, most explained in the [Adding your own custom PointT type](#) tutorial.

## 2.1 Compiling your first code example

Until we find the right minimal code example, please take a look at the [Using PCL in your own project](#) and [Writing a new PCL class](#) tutorials to see how to compile and write code for or using PCL.

---

### Using PCL in your own project

---

This tutorial explains how to use PCL in your own projects.

#### Contents

- *Using PCL in your own project*
  - *Prerequisites*
  - *Project settings*
  - *The explanation*
  - *Compiling and running the project*
    - \* *Using command line CMake*
    - \* *Using CMake gui (e.g. Windows)*
  - *Weird installations*

### 3.1 Prerequisites

We assume you have downloaded, compiled and installed PCL on your machine.

### 3.2 Project settings

Let us say the project is placed under `/PATH/TO/MY/GRAND/PROJECT` that contains a lonely `cpp` file name `pcd_write.cpp` (copy it from the `writing_pcd` tutorial). In the same folder, create a file named `CMakeLists.txt` that contains:

```
cmake_minimum_required(VERSION 2.6 FATAL_ERROR)
project(MY_GRAND_PROJECT)
find_package(PCL 1.3 REQUIRED COMPONENTS common io)
include_directories(${PCL_INCLUDE_DIRS})
link_directories(${PCL_LIBRARY_DIRS})
add_definitions(${PCL_DEFINITIONS})
add_executable(pcd_write_test pcd_write.cpp)
target_link_libraries(pcd_write_test ${PCL_LIBRARIES})
```

### 3.3 The explanation

Now, let's see what we did.

```
cmake_minimum_required(VERSION 2.6 FATAL_ERROR)
```

This is mandatory for cmake, and since we are making very basic project we don't need features from cmake 2.8 or higher.

```
project(MY_GRAND_PROJECT)
```

This line names your project and sets some useful cmake variables such as those to refer to the source directory (MY\_GRAND\_PROJECT\_SOURCE\_DIR) and the directory from which you are invoking cmake (MY\_GRAND\_PROJECT\_BINARY\_DIR).

```
find_package(PCL 1.3 REQUIRED COMPONENTS common io)
```

We are requesting to find the PCL package at minimum version 1.3. We also says that it is REQUIRED meaning that cmake will fail gracefully if it can't be found. As PCL is modular one can request:

- only one component: `find_package(PCL 1.3 REQUIRED COMPONENTS io)`
- several: `find_package(PCL 1.3 REQUIRED COMPONENTS io common)`
- all existing: `find_package(PCL 1.3 REQUIRED)`

```
include_directories(${PCL_INCLUDE_DIRS})
link_directories(${PCL_LIBRARY_DIRS})
add_definitions(${PCL_DEFINITIONS})
```

When PCL is found, several related variables are set:

- *PCL\_FOUND*: set to 1 if PCL is found, otherwise unset
- *PCL\_INCLUDE\_DIRS*: set to the paths to PCL installed headers and the dependency headers
- *PCL\_LIBRARIES*: set to the file names of the built and installed PCL libraries
- *PCL\_LIBRARY\_DIRS*: set to the paths to where PCL libraries and 3rd party dependencies reside
- *PCL\_VERSION*: the version of the found PCL
- *PCL\_COMPONENTS*: lists all available components
- *PCL\_DEFINITIONS*: lists the needed preprocessor definitions and compiler flags

To let cmake know about external headers you include in your project, one needs to use `include_directories()` macro. In our case *PCL\_INCLUDE\_DIRS*, contains exactly what we need, thus we ask cmake to search the paths it contains for a header potentially included.

```
add_executable(pcd_write_test pcd_write.cpp)
```

Here, we tell cmake that we are trying to make an executable file named `pcd_write_test` from one single source file `pcd_write.cpp`. CMake will take care of the suffix (`.exe` on Windows platform and blank on UNIX) and the permissions.

```
target_link_libraries(pcd_write_test ${PCL_LIBRARIES})
```

The executable we are building makes call to PCL functions. So far, we have only included the PCL headers so the compilers knows about the methods we are calling. We need also to make the linker knows about the libraries we are linking against. As said before the, PCL found libraries are referred to using `PCL_LIBRARIES` variable, all that remains is to trigger the link operation which we do calling `target_link_libraries()` macro. `PCLConfig.cmake` uses a CMake special feature named *EXPORT* which allows for using others' projects targets as if you built them yourself. When you are using such targets they are called *imported targets* and acts just like any other target.

## 3.4 Compiling and running the project

### 3.4.1 Using command line CMake

Make a directory called `build`, in which the compilation will be done. Do:

```
$ cd /PATH/TO/MY/GRAND/PROJECT
$ mkdir build
$ cd build
$ cmake ..
```

You will see something similar to:

```
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Found PCL_IO: /usr/local/lib/libpcl_io.so
-- Found PCL: /usr/local/lib/libpcl_io.so (Required is at least version "1.0")
-- Configuring done
-- Generating done
-- Build files have been written to: /PATH/TO/MY/GRAND/PROJECT/build
```

If you want to see what is written on the CMake cache:

```
CMAKE_BUILD_TYPE
CMAKE_INSTALL_PREFIX      /usr/local
PCL_DIR                   /usr/local/share/pcl
```

Now, we can build up our project, simply typing:

```
$ make
```

The result should be as follow:

```
Scanning dependencies of target pcd_write_test
[100%] Building CXX object
CMakeFiles/pcd_write_test.dir/pcd_write.cpp.o
Linking CXX executable pcd_write_test
[100%] Built target pcd_write_test
```

The project is now compiled, linked and ready to test:

```
$ ./pcd_write_test
```

Which leads to this:

```
Saved 5 data points to test_pcd.pcd.
0.352222 -0.151883 -0.106395
-0.397406 -0.473106 0.292602
-0.731898 0.667105 0.441304
-0.734766 0.854581 -0.0361733
-0.4607 -0.277468 -0.916762
```

### 3.4.2 Using CMake gui (e.g. Windows)

Run CMake GUI, and fill these fields :

- Where is the source code : this is the folder containing the CMakeLists.txt file and the sources.
- Where to build the binaries : this is where the Visual Studio project files will be generated

Then, click Configure. You will be prompted for a generator/compiler. Then click the Generate button. If there is no errors, the project files will be generated into the Where to build the binaries folder.

Open the sln file, and build your project!

## 3.5 Weird installations

CMake has a list of default searchable paths where it seeks for FindXXX.cmake or XXXConfig.cmake. If you happen to install in some non obvious repository (let us say in *Documents* for evils) then you can help cmake find PCLConfig.cmake adding this line:

```
set(PCL_DIR "/path/to/PCLConfig.cmake")
```

before this one:

```
find_package(PCL 1.3 REQUIRED COMPONENTS common io)
...
```

---

## Compiling PCL from source on POSIX compliant systems

---

Though not a dependency per se, don't forget that you also need the [CMake build system](#), at least version 3.5.0. Additional help on how to use the CMake build system is available [here](#).

Please note that the following installation instructions are only valid for POSIX systems (e.g., Linux, MacOS) with an already installed make/gnu toolchain. For instructions on how to download and compile PCL in Windows (which uses a slightly different process), please visit [our tutorials page](#).

### Contents

- *Compiling PCL from source on POSIX compliant systems*
  - *Stable*
  - *Experimental*
  - *Dependencies*
    - \* *Mandatory*
    - \* *Optional*
  - *Troubleshooting*
    - \* *MacOS X*

## 4.1 Stable

For systems for which we do not offer precompiled binaries, you need to compile Point Cloud Library (PCL) from source. Here are the steps that you need to take: Go to [Github](#) and download the version number of your choice. Uncompress the tar-bzip archive, e.g. (replace 1.7.2 with the correct version number):

```
tar xvfj pcl-pcl-1.7.2.tar.gz
```

Change the directory to the `pcl-pcl-1.7.2` (replace 1.7.2 with the correct version number) directory, and create a build directory in there:

```
cd pcl-pcl-1.7.2 && mkdir build && cd build
```

Run the CMake build system using the default options:

```
cmake ..
```

Or change them (uses `cmake-curses-gui`):

```
ccmake ..
```

Please note that `cmake` might default to a debug build. If you want to compile a release build of PCL with enhanced compiler optimizations, you can change the build target to “Release” with “`-DCMAKE_BUILD_TYPE=Release`”:

```
cmake -DCMAKE_BUILD_TYPE=Release ..
```

Finally compile everything (see [compiler\\_optimizations](#)):

```
make -j2
```

And install the result:

```
make -j2 install
```

Or alternatively, if you did not change the variable which declares where PCL should be installed, do:

```
sudo make -j2 install
```

Here’s everything again, in case you want to copy & paste it:

```
cd pcl-pcl-1.7.2 && mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make -j2
sudo make -j2 install
```

Again, for a detailed tutorial on how to compile and install PCL and its dependencies in Microsoft Windows, please visit [our tutorials page](#). Additional information for developers is available at the [Github PCL Wiki](#).

## 4.2 Experimental

If you are eager to try out a certain feature of PCL that is currently under development (or you plan on developing and contributing to PCL), we recommend you try checking out our source repository, as shown below. If you’re just interested in browsing our source code, you can do so by visiting <https://github.com/PointCloudLibrary/pcl>.

Clone the repository:

```
git clone https://github.com/PointCloudLibrary/pcl pcl-trunk
```

Please note that above steps (3-5) are almost identical for compiling the experimental PCL trunk code:

```
cd pcl-trunk && mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo ..
make -j2
sudo make -j2 install
```



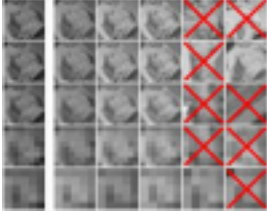

## 4.3 Dependencies

Because PCL is split into a list of code libraries, the list of dependencies differs based on what you need to compile. The difference between mandatory and optional dependencies, is that a mandatory dependency is required in order for that particular PCL library to compile and function, while an optional dependency disables certain functionality within a PCL library but compiles the rest of the library that does not require the dependency.

### 4.3.1 Mandatory




The following code libraries are **required** for the compilation and usage of the PCL libraries shown below:

pcl\_\* denotes all PCL libraries, meaning that the particular dependency is a strict requirement for the usage of anything in PCL.

Logo	Library	Minimum version	Mandatory
	Boost	1.40 (without OpenNI) 1.47 (with OpenNI)	pcl_*
	Eigen	3.0	pcl_*
	FLANN	1.7.1	pcl_*
	VTk	5.6	pcl_visualization

### 4.3.2 Optional

The following code libraries enable certain additional features for the PCL libraries shown below, and are thus **optional**:

Logo	Library	Minimum version	Mandatory
	Qhull	2011.1	pcl_surface
	OpenNI	1.3	pcl_io
	CUDA	4.0	pcl_*

## 4.4 Troubleshooting

In certain situations, the instructions above might fail, either due to custom versions of certain library dependencies installed, or different operating systems than the ones we usually develop on, etc. This section here contains links to discussions held in our community regarding such cases. Please read it before posting new questions on the mailing list, and also **use the search features provided by our forums** - there's no point in starting a new thread if an older one that discusses the same issue already exists.

### 4.4.1 MacOS X

libGL issue when running visualization apps on OSX

---

## Customizing the PCL build process

---

This tutorial explains how to modify the PCL cmake options and tweak your building process to better fit the needs of your project and/or your system's requirements.

### Contents

- *Customizing the PCL build process*
  - *Audience*
  - *Prerequisites*
  - *PCL basic settings*
  - *The explanation*
  - *Tweaking basic settings*
  - *Tweaking advanced settings*
    - \* *Building unit tests*
    - \* *General remarks*
  - *Detailed description*

## 5.1 Audience

This tutorial targets users with a basic knowledge of CMake, C++ compilers, linkers, flags and make.

## 5.2 Prerequisites

We assume you have checked out the last available revision of PCL.

## 5.3 PCL basic settings

Let's say PCL is placed under `/PATH/TO/PCL`, which we will refer to as `PCL_ROOT`:

```
$ cd $PCL_ROOT
$ mkdir build && cd build
$ cmake ..
```

This will cause *cmake* to create a file called `CMakeCache.txt` in the build directory with the default options.

Let's have a look at what *cmake* options got enabled:

```
$ cmake ..
```

You should see something like the following on screen:

```
BUILD_common           ON
BUILD_features         ON
BUILD_filters          ON
BUILD_global_tests     OFF
BUILD_io               ON
BUILD_kdtree           ON
BUILD_keypoints        ON
BUILD_octree           ON
BUILD_range_image      ON
BUILD_registration     ON
BUILD_sample_consensus ON
BUILD_segmentation     ON
BUILD_surface          ON
BUILD_visualization    ON
CMAKE_BUILD_TYPE
CMAKE_INSTALL_PREFIX   /usr/local
PCL_SHARED_LIBS        ON
PCL_VERSION            1.0.0
VTK_DIR                /usr/local/lib/vtk-5.6
```

## 5.4 The explanation

- *BUILD\_common*: option to enable/disable building of common library
- *BUILD\_features*: option to enable/disable building of features library
- *BUILD\_filters*: option to enable/disable building of filters library
- *BUILD\_global\_tests*: option to enable/disable building of global unit tests
- *BUILD\_io*: option to enable/disable building of io library
- *BUILD\_kdtree*: option to enable/disable building of kdtree library
- *BUILD\_keypoints*: option to enable/disable building of keypoints library
- *BUILD\_octree*: option to enable/disable building of octree library
- *BUILD\_range\_image*: option to enable/disable building of range\_image library
- *BUILD\_registration*: option to enable/disable building of registration library
- *BUILD\_sample\_consensus*: option to enable/disable building of sample\_consensus library

- *BUILD\_segmentation*: option to enable/disable building of segmentation library
- *BUILD\_surface*: option to enable/disable building of surface library
- *BUILD\_visualization*: option to enable/disable building of visualization library
- *CMAKE\_BUILD\_TYPE*: here you specify the build type. In CMake, a *CMAKE\_BUILD\_TYPE* corresponds to a set of options and flags passed to the compiler to activate/deactivate a functionality and to constrain the building process.
- *CMAKE\_INSTALL\_PREFIX*: where the headers and the built libraries will be installed
- *PCL\_SHARED\_LIBS*: option to enable building of shared libraries. Default is yes.
- *PCL\_VERSION*: this is the PCL library version. It affects the built libraries names.
- *VTk\_DIR*: directory of VTK library if found

The above are called *cmake* cached variables. At this level we only looked at the basic ones.

## 5.5 Tweaking basic settings

Depending on your project/system, you might want to enable/disable certain options. For example, you can prevent the building of:

- tests: setting *BUILD\_global\_tests* to *OFF*
- a library: setting *BUILD\_LIBRARY\_NAME* to *OFF*

Note that if you disable a XXX library that is required for building YYY then XXX will be built but won't appear in the cache.

You can also change the build type:

- **Debug**: means that no optimization is done and all the debugging symbols are embedded into the libraries file. This is platform and compiler dependent. On Linux with gcc this is equivalent to running gcc with *-O0 -g -ggdb -Wall*
- **Release**: the compiled code is optimized and no debug information will be printed out. This will lead to *-O3* for gcc and *-O5* for clang
- **RelWithDebInfo**: the compiled code is optimized but debugging data is also embedded in the libraries. This is a tradeoff between the two former ones.
- **MinSizeRel**: this, normally, results in the smallest libraries you can build. This is interesting when building for Android or a restricted memory/space system.

A list of available CMAKE\_BUILD\_TYPES can be found typing:

```
$ cmake --help-variable CMAKE_BUILD_TYPE
```

## 5.6 Tweaking advanced settings

Now we are done with all the basic stuff. To turn on advanced cache options hit *t* while in *ccmake*. Advanced options become especially useful when you have dependencies installed in unusual locations and thus *cmake* hangs with *XXX\_NOT\_FOUND* this can even prevent you from building PCL although you have all the dependencies installed. In this section we will discuss each dependency entry so that you can configure/build or update/build PCL according to your system.

### 5.6.1 Building unit tests

If you want to contribute to PCL, or are modifying the code, you need to turn on building of unit tests. This is accomplished by setting the `BUILD_global_tests` option to `ON`, with a few caveats. If you're using `ccmake` and you find that `BUILD_global_tests` is reverting to `OFF` when you configure, you can move the cursor up to the `BUILD_global_tests` line to see the error message.

Two options which will need to be turned ON before `BUILD_global_tests` are `BUILD_outofcore` and `BUILD_people`. Your mileage may vary.

Also required for unit tests is the source code for the Google C++ Testing Framework. That is usually as simple as downloading the source, extracting it, and pointing the `GTEST_SRC_DIR` and `GTEST_INCLUDE_DIR` options to the applicable source locations. On Ubuntu, you can simply run `apt-get install libgtest-dev`.

These steps enable the `tests` make target, so you can use `make tests` to run tests.

### 5.6.2 General remarks

Under `${PCL_ROOT}/cmake/Modules` there is a list of `FindXXX.cmake` files used to locate dependencies and set their related variables. They have a list of default searchable paths where to look for them. In addition, if `pkg-config` is available then it is triggered to get hints on their locations. If all of them fail, then we look for a CMake entry or environment variable named `XXX_ROOT` to find headers and libraries. We recommend setting an environment variable since it is independent from CMake and lasts over the changes you can make to your configuration.

The available ROOTs you can set are as follow:

- **BOOST\_ROOT**: for boost libraries with value `C:/Program Files/boost-1.4.6` for instance
- **CMINPACK\_ROOT**: for cminpack with value `C:/Program Files/CMINPACK 1.1.13` for instance
- **QHULL\_ROOT**: for qhull with value `C:/Program Files/qhull 6.2.0.1373` for instance
- **FLANN\_ROOT**: for flann with value `C:/Program Files/flann 1.6.8` for instance
- **EIGEN\_ROOT**: for eigen with value `C:/Program Files/Eigen 3.0.0` for instance

To ensure that all the dependencies were correctly found, beside the message you get from CMake, you can check or edit each dependency specific variables and give it the value that best fits your needs.

UNIX users generally don't have to bother with debug vs release versions they are fully compliant. You would just loose debug symbols if you use release libraries version instead of debug while you will end up with much more verbose output and slower execution. This said, Windows MSVC users and Apple iCode ones can build debug/release from the same project, thus it will be safer and more coherent to fill them accordingly.

## 5.7 Detailed description

Below, each dependency variable is listed, its meaning is explained then a sample value is given for reference.

- Boost

cache variable	meaning	sample value
Boost_DATE_TIME_LIBRARY	full path to boost_date-time.[so,lib,a]	/usr/local/lib/libboost_date_time.so
Boost_DATE_TIME_LIBRARY_DEBUG	full path to boost_date-time.[so,lib,a] (debug version)	/usr/local/lib/libboost_date_time-gd.so
Boost_DATE_TIME_LIBRARY_RELEASE	full path to boost_date-time.[so,lib,a] (release version)	/usr/local/lib/libboost_date_time.so
Boost_FILESYSTEM_LIBRARY	full path to boost_filesystem.[so,lib,a]	/usr/local/lib/libboost_filesystem.so
Boost_FILESYSTEM_LIBRARY_DEBUG	full path to boost_filesystem.[so,lib,a] (debug version)	/usr/local/lib/libboost_filesystem-gd.so
Boost_FILESYSTEM_LIBRARY_RELEASE	full path to boost_filesystem.[so,lib,a] (release version)	/usr/local/lib/libboost_filesystem.so
Boost_INCLUDE_DIR	path to boost headers directory	/usr/local/include
Boost_LIBRARY_DIRS	path to boost libraries directory	/usr/local/lib
Boost_SYSTEM_LIBRARY	full path to boost_system.[so,lib,a]	/usr/local/lib/libboost_system.so
Boost_SYSTEM_LIBRARY_DEBUG	full path to boost_system.[so,lib,a] (debug version)	/usr/local/lib/libboost_system-gd.so
Boost_SYSTEM_LIBRARY_RELEASE	full path to boost_system.[so,lib,a] (release version)	/usr/local/lib/libboost_system.so

- CMinpack

cache variable	meaning	sample value
CMINPACK_INCLUDE_DIR	path to cminpack headers directory	/usr/local/include/cminpack-1
CMINPACK_LIBRARY	full path to cminpack.[so,lib,a] (release version)	/usr/local/lib/libcminpack.so
CMINPACK_LIBRARY_DEBUG	full path to cminpack.[so,lib,a] (debug version)	/usr/local/lib/libcminpack-gd.so

- FLANN

cache variable	meaning	sample value
FLANN_INCLUDE_DIR	path to flann headers directory	/usr/local/include
FLANN_LIBRARY	full path to libflann_cpp.[so,lib,a] (release version)	/usr/local/lib/libflann_cpp.so
FLANN_LIBRARY_DEBUG	full path to libflann_cpp.[so,lib,a] (debug version)	/usr/local/lib/libflann_cpp-gd.so

- Eigen

cache variable	meaning	sample value
EIGEN_INCLUDE_DIR	path to eigen headers directory	/usr/local/include/eigen3



---

### Building PCL's dependencies from source on Windows

---

This tutorial explains how to build the Point Cloud Library needed dependencies **from source** on Microsoft Windows platforms, and tries to guide you through the download and the compilation process. As an example, we will be building the sources with Microsoft Visual Studio 2008 to get 32bit libraries. The procedure is almost the same for other compilers and for 64bit libraries.

---

**Note:** Don't forget that all the dependencies must be compiled using the same compiler options and architecture specifications, i.e. you can't mix 32 bit libraries with 64 bit libraries.

---



**Contents**

- *Building PCL's dependencies from source on Windows*
  - *Requirements*
  - *Building dependencies*
  - *Building PCL*

## 6.1 Requirements

In order to compile every component of the PCL library we need to download and compile a series of 3rd party library dependencies:

- **Boost** version  $\geq 1.46.1$  (<http://www.boost.org/>)  
used for shared pointers, and threading. **mandatory**
- **Eigen** version  $\geq 3.0.0$  (<http://eigen.tuxfamily.org/>)  
used as the matrix backend for SSE optimized math. **mandatory**
- **FLANN** version  $\geq 1.6.8$  (<http://www.cs.ubc.ca/research/flann/>)  
used in *kdtree* for fast approximate nearest neighbors search. **mandatory**
- **Visualization ToolKit (VTK)** version  $\geq 5.6.1$  (<http://www.vtk.org/>)  
used in *visualization* for 3D point cloud rendering and visualization. **mandatory**
- **googletest** version  $\geq 1.6.0$  (<http://code.google.com/p/googletest/>)  
used to build test units. **optional**
- **QHULL** version  $\geq 2011.1$  (<http://www.qhull.org/>)  
used for convex/concave hull decompositions in *surface*. **optional**
- **OpenNI** version  $\geq 1.1.0.25$  (<http://www.openni.org/>)  
used to grab point clouds from OpenNI compliant devices. **optional**
- **Qt** version  $\geq 4.6$  (<http://qt.digia.com/>)  
used for developing applications with a graphical user interface (GUI) **optional**

---

**Note:** Though not a dependency per se, don't forget that you also need the CMake build system (<http://www.cmake.org/>), at least version **3.5.0**. A Git client for Windows is also required to download the PCL source code.

---

## 6.2 Building dependencies

In this tutorial, we'll be compiling these libraries versions:

```
Boost : 1.48.0
Flann : 1.7.1
Qhull : 2011.1
Qt : 4.8.0
VTK : 5.8.0
GTest : 1.6.0
```

Let's unpack all our libraries in C:/PCL\_dependencies so that it would like like:

```
C:/PCL_dependencies
C:/PCL_dependencies/boost-cmake
C:/PCL_dependencies/eigen
C:/PCL_dependencies/flann-1.7.1-src
C:/PCL_dependencies/gtest-1.6.0
C:/PCL_dependencies/qhull
C:/PCL_dependencies/VTK
```

- **Boost :**

Let's start with *Boost*. We will be using the *CMake-able Boost* project which provide a CMake based build system for Boost.

To build Boost, open the CMake-gui and fill in the fields:

```
Where is my source code: C:/PCL_dependencies/boost-cmake
Where to build binaries: C:/PCL_dependencies/boost-cmake/build
```

Before clicking on “Configure”, click on “Add Entry” button in the top right of CMake gui, in the popup window, fill the fields as follows:

```
Name : LIBPREFIX
Type : STRING
Value : lib
```

**Note:** If you are using **Visual Studio 2010**, then add also these 3 CMake entries before clicking “Configure”:

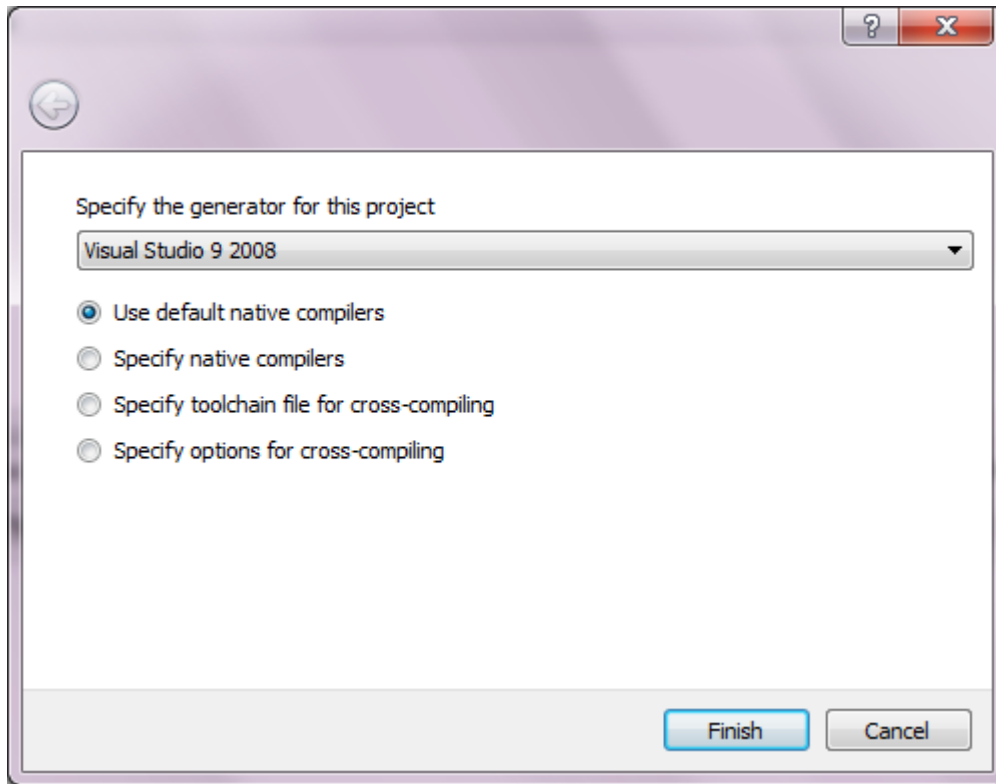
```
Name : BOOST_TOOLSET
Type : STRING
Value : vc100

Name : BOOST_COMPILER
Type : STRING
Value : msvc

Name : BOOST_COMPILER_VERSION
Type : STRING
Value : 10.0
```

Hit the “Configure” button and CMake will tell that the binaries folder doesn't exist yet (e.g., *C:/PCL\_dependencies/boost-cmake/build*) and it will ask for a confirmation.

Proceed and be sure to choose the correct “Generator” on the next window. So, we choose “Visual Studio 9 2008” generator.



---

**Note:** If you want to build 64 bit libraries, then choose “Visual Studio 9 2008 Win64” as generator.

---

By default, all of the Boost modules will be built. If you want to build only the required modules for PCL, then fill the **BUILD\_PROJECTS** CMake entry (which is set to *ALL* by default) with a semicolon-separated list of boost modules:

```
BUILD_PROJECTS : system;filesystem;date_time;iostreams;trl;serialization
```

Also, uncheck the **ENABLE\_STATIC\_RUNTIME** checkbox. Then, click “Configure” again. If you get some errors related to Python, then uncheck **WITH\_PYTHON** checkbox, and click “Configure” again. Now, in the CMake log, you should see something like:

```
Reading boost project directories (per BUILD_PROJECTS)

+ date_time
+ serialization
+ system
+ filesystem
+-- optional python bindings disabled since PYTHON_FOUND is false.
+ trl
```

Now, click “Generate”. A Visual Studio solution file will be generated inside the build folder (e.g. C:/PCL\_dependencies/boost-cmake/build). Open the *Boost.sln* file, then right click on *INSTALL* project and choose *Build*. The ‘INSTALL’ project will trigger the build of all the projects in the solution file, and then will install the build libraries along with the header files to the default installation folder (e.g. C:/Program Files (x86)/Boost).

---

**Note:** If you get some errors during the installation process, it could be caused by the UAC of MS

---

Windows Vista or Seven. To fix this, close Visual Studio, right click on its icon on the Desktop or in the Start Menu, and choose “Run as administrator”. Then Open the *Boost.sln* file, and build the **INSTALL** project.

- **Eigen :**

Eigen is a headers only library, so you can use the Eigen installer provided on the [downloads page](#).

- **Flann :**

Let’s move on to *FLANN*. Then open CMake-gui and fill in the fields:

Where **is** my source code: C:/PCL\_dependencies/flann-1.7.1-src  
 Where to build binaries: C:/PCL\_dependencies/flann-1.7.1-src/build

Hit the “Configure” button. Proceed and be sure to choose the correct “Generator” on the next window. You can safely ignore any warning message about hdf5.

Now, on my machine I had to manually set the *BUILD\_PYTHON\_BINDINGS* and *BUILD\_MATLAB\_BINDINGS* to OFF otherwise it would not continue to the next step as it is complaining about unable to find Python and Matlab. Click on “Advanced mode” and find them, or alternatively, add those entries by clicking on the “Add Entry” button in the top right of the CMake-gui window. Add one entry named “BUILD\_PYTHON\_BINDINGS”, set its type to “Bool” and its value to unchecked. Do the same with the “BUILD\_MATLAB\_BINDINGS” entry.

Now hit the “Configure” button and it should work. Go for the “Generate” This will generate the required project files/makefiles to build the library. Now you can simply go to *C:/PCL\_dependencies/flann-1.7.1-src/build* and proceed with the compilation using your toolchain. In case you use Visual Studio, you will find the Visual Studio Solution file in that folder.

Build the **INSTALL** project in **release** mode.

---

**Note:** If you don’t have a Python interpreter installed CMake would probably not allow you to generate the project files. To solve this problem you can install the Python interpreter (<https://www.python.org/download/windows/>) or comment the *add\_subdirectory( test )* line from *C:/PCL\_dependencies/flann-1.7.1-src/CMakeLists.txt* .

---

- **QHull :**

Setup the CMake fields with the *qhull* paths:

Where **is** my source code: C:/PCL\_dependencies/qhull-2011.1  
 Where to build binaries: C:/PCL\_dependencies/qhull-2011.1/build

Before clicking on “Configure”, click on “Add Entry” button in the top right of CMake gui, in the popup window, fill the fields as follows:

Name : CMAKE\_DEBUG\_POSTFIX  
 Type : STRING  
 Value : \_d

Then click “Ok”. This entry will define a postfix to distinguish between debug and release libraries.

Then hit “Configure” twice and “Generate”. Then build the **INSTALL** project, both in **debug** and **release** configuration.

- **VTK :**

**Note:** If you want to build PCL GUI tools, you need to build VTK with Qt support, so obviously, you need to build/install Qt before VTK.

To configure Qt, we need to have Perl installed on your system. If it is not, just download and install it from <http://strawberryperl.com>.

To build Qt from sources, download the source archive from Qt website. Unpack it some where on your disk (C:\Qt\4.8.0 e.g. for Qt 4.8.0). Then open a *Visual Studio Command Prompt* :

Click **Start**, point to **All Programs**, point to **Microsoft Visual Studio 20XX**, point to **Visual Studio Tools**, and then click **Visual Studio Command Prompt** if you are building in 32bit, or **Visual Studio x64 Win64 Command Prompt** if you are building in 64bit.

In the command prompt, **cd** to Qt directory:

```
prompt> cd c:\Qt\4.8.0
```

We configure a minimal build of Qt using the Open Source licence. If you need a custom build, adjust the options as needed:

```
prompt> configure -opensource -confirm-license -fast -debug-and-release -  
↪nomake examples -nomake demos -no-qt3support -no-xmlpatterns -no-  
↪multimedia -no-phonon -no-accessibility -no-opengl -no-webkit -no-  
↪script -no-scripttools -no-dbus -no-declarative
```

Now, let's build Qt:

```
prompt> nmake
```

Now, we can clear all the intermediate files to free some disk space:

```
prompt> nmake clean
```

We're done with Qt! But before building VTK, we need to set an environment variable:

```
QtDir = C:\Qt\4.8.0
```

and then, append `%QtDir%\bin` to your PATH environment variable.

Now, configure VTK using CMake (make sure to restart CMake after setting the environment variables). First, setup the CMake fields with the *VTK* paths, e.g.:

```
Where is my source code: C:/PCL_dependencies/VTK  
Where to build binaries: C:/PCL_dependencies/VTK/bin32
```

Then hit "Configure". Check this checkbox and click "Configure":

```
VTK_USE_QT
```

Make sure CMake did find Qt by looking at `QT_QMAKE_EXECUTABLE` CMake entry. If not, set it to the path of `qmake.exe`, e.g. `C:\Qt\4.8.0\bin\qmake.exe`, then click "Configure".

If Qt is found, then check this checkbox and click "Configure":

```
VTK_USE_QVTK_QTOPENGL
```

Then, click "Generate", open the generated solution file, and build it in debug and release.

That's it, we're done with the dependencies!

- **GTest :**

In case you want PCL tests (not recommended for users), you need to compile the *googletest* library (GTest). Setup the CMake fields as usual:

Where <b>is</b> my source code: C:/PCL_dependencies/gtest-1.6.0
Where to build binaries: C:/PCL_dependencies/gtest-1.6.0/bin32

Hit “Configure” and set the following options:

BUILD_SHARED_LIBS	OFF
gtest_force_shared_crt	ON

Generate and build the resulting project.

## 6.3 Building PCL

Now that you built and installed PCL dependencies, you can follow the “*Compiling PCL from source on Windows*” tutorial to build PCL itself.



---

## Compiling PCL from source on Windows

---

This tutorial explains how to build the Point Cloud Library **from source** on Microsoft Windows platforms. In this tutorial, we assume that you have built and installed all the required dependencies, or that you have installed them using the dependencies installers provided on the [downloads page](#).

### Contents

- *Compiling PCL from source on Windows*
  - *Requirements*
  - *Downloading PCL source code*
  - *Configuring PCL*
  - *Building PCL*
  - *Installing PCL*
  - *Advanced topics*
  - *Using PCL*

---

**Note:** If you installed PCL using one of the **all-in-one** provided installers, then this tutorial is not for you. The **all-in-one** installer already contains prebuilt PCL binaries which are ready to be used without any compilation step.

---

---

**Note:** If there is no installers for your compiler, it is recommended that you build the dependencies out of source. The *Building PCL's dependencies from source on Windows* tutorial should guide you through the download and the compilation of all the required dependencies.

---



## 7.1 Requirements

we assume that you have built and installed all the required dependencies, or that you have installed them using the dependencies installers provided on the [downloads page](#). Installing them to the default locations will make configuring PCL easier.

- **Boost**

used for shared pointers, and threading. **mandatory**

- **Eigen**

used as the matrix backend for SSE optimized math. **mandatory**

- **FLANN**

used in *kdtree* for fast approximate nearest neighbors search. **mandatory**

- **Visualization ToolKit (VTK)**

used in *visualization* for 3D point cloud rendering and visualization. **mandatory**

- **Qt**

used for applications with a graphical user interface (GUI) **optional**

- **QHULL**

used for convex/concave hull decompositions in *surface*. **optional**

- **OpenNI and patched Sensor Module**

used to grab point clouds from OpenNI compliant devices. **optional**

- **GTest** version  $\geq 1.6.0$  (<http://code.google.com/p/googletest/>)

is needed only to build PCL tests. We do not provide GTest installers. **optional**

**Note:** Though not a dependency per se, don't forget that you also need the CMake build system (<http://www.cmake.org/>), at least version **3.5.0**. A Git client for Windows is also required to download the PCL source code.

---

## 7.2 Downloading PCL source code

To build the current official release, download the source archive from <http://pointclouds.org/downloads/> and extract it somewhere on your disk, say C:\PCL\PCL-1.5.1-Source. In this case, you can go directly to Configuring PCL section, and pay attention to adjust the paths accordingly.

Or, you might want to build an experimental version of PCL to test some new features not yet available in the official releases. For this, you will need git ( <http://git-scm.com/download> ).

The invocation to download the source code is thus, using a command line:

```
cd wherever/you/want/to/put/the/repo/ git clone https://github.com/PointCloudLibrary/pcl.git
```

You could also use Github for Windows (<https://windows.github.com/>), but that is potentially more troublesome than setting up git on windows.

## 7.3 Configuring PCL

On Windows, we recommend to build **shared** PCL libraries with **static** dependencies. In this tutorial, we will use static dependencies when possible to build shared PCL. You can easily switch to using shared dependencies. Then, you need to make sure you put the dependencies' dlls either in your *PATH* or in the same folder as PCL dlls and executables. You can also build static PCL libraries if you want.

Run the CMake-gui application and fill in the fields:

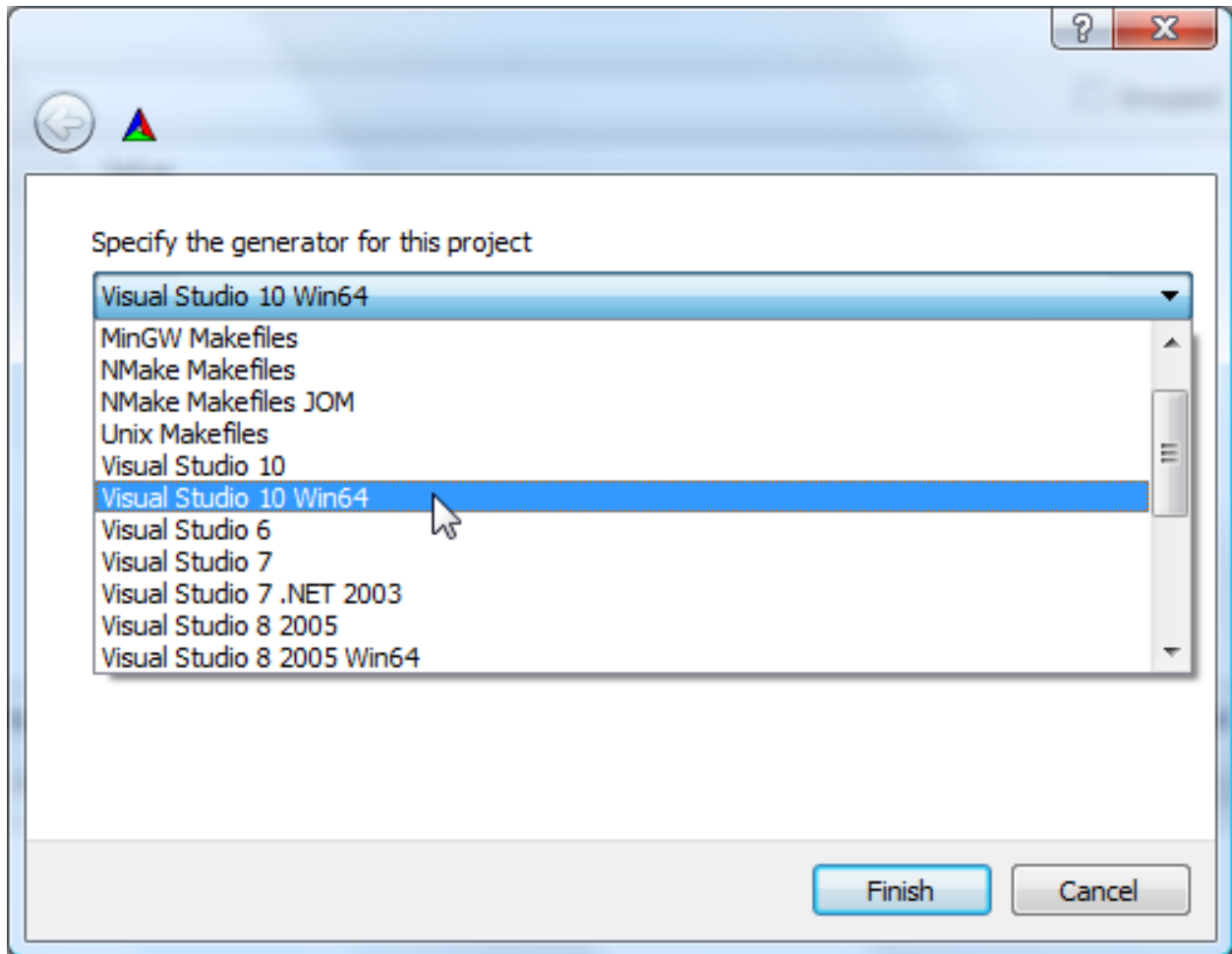
```
Where is the source code    : C:/PCL/pcl
Where to build the binaries: C:/PCL
```

Now hit the “Configure” button. You will be asked for a *generator*. A generator is simply a compiler.

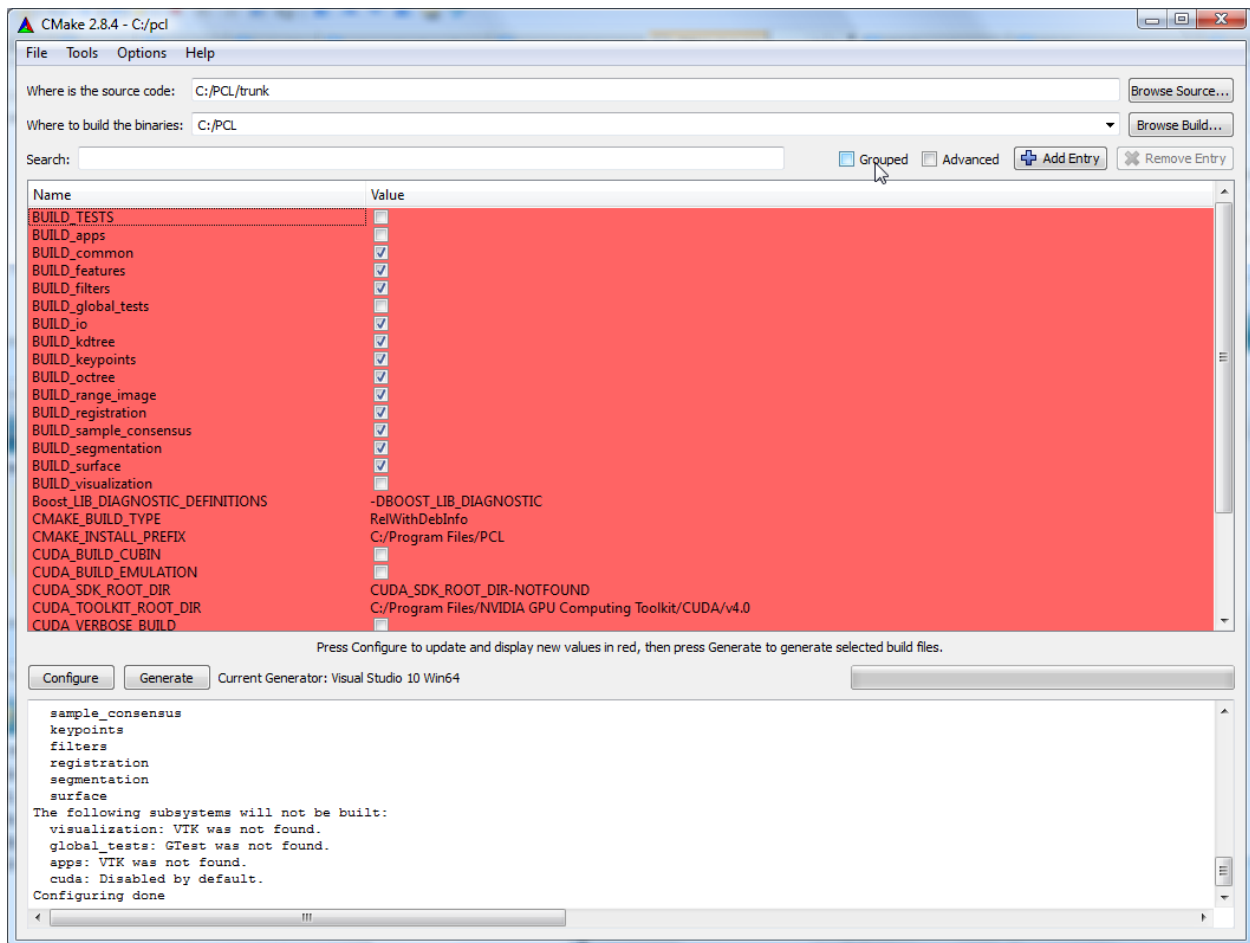
**Note:** In this tutorial, we will be using Microsoft Visual C++ 2010 compiler. If you want to build 32bit PCL, then pick the “**Visual Studio 10**” generator. If you want to build 64bit PCL, then pick the “**Visual Studio 10 Win64**”.

Make sure you have installed the right third party dependencies. You cannot mix 32bit and 64bit code, and it is highly recommended to not mix codes compiled with different compilers.

---

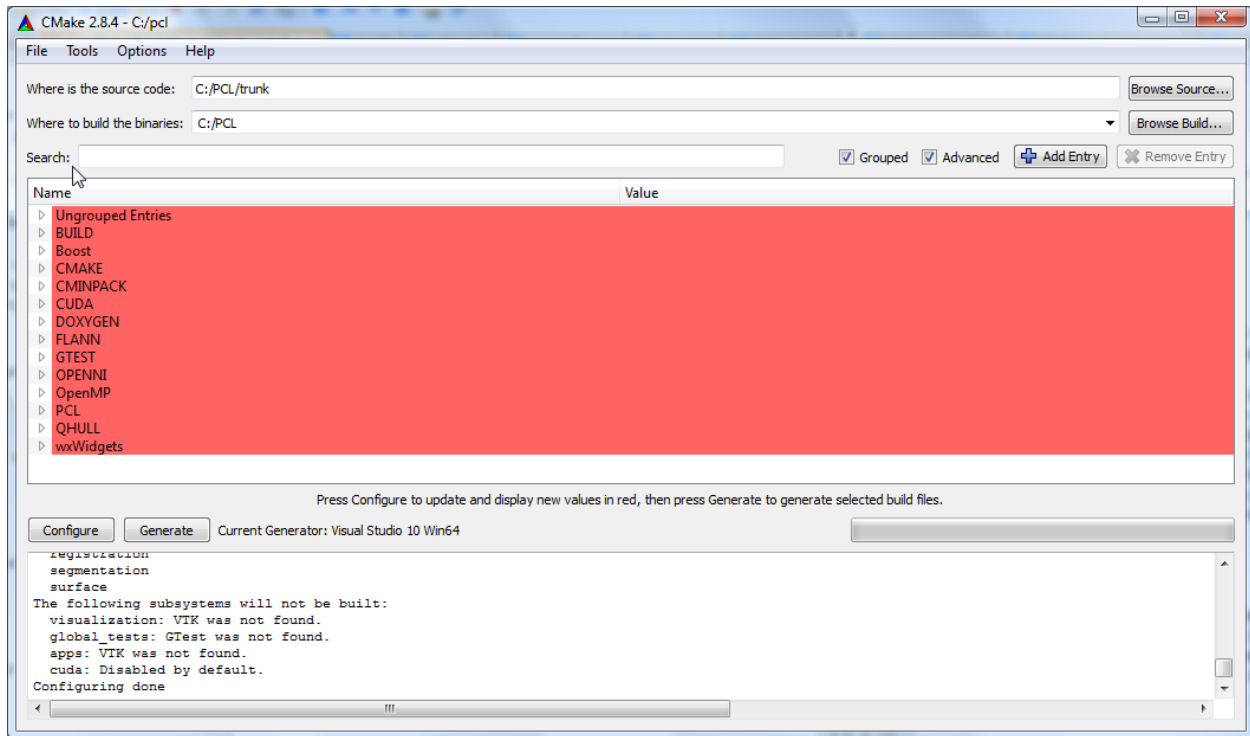


In the remaining of this tutorial, we will be using “**Visual Studio 10 Win64**” generator. Once you picked your generator, hit finish to close the dialog window. CMake will start configuring PCL and looking for its dependencies. For example, we can get this output :



The upper part of CMake window contains a list of CMake variables and its respective values. The lower part contains some logging output that can help figure out what is happening. We can see, for example, that VTK was not found, thus, the visualization module will not get built.

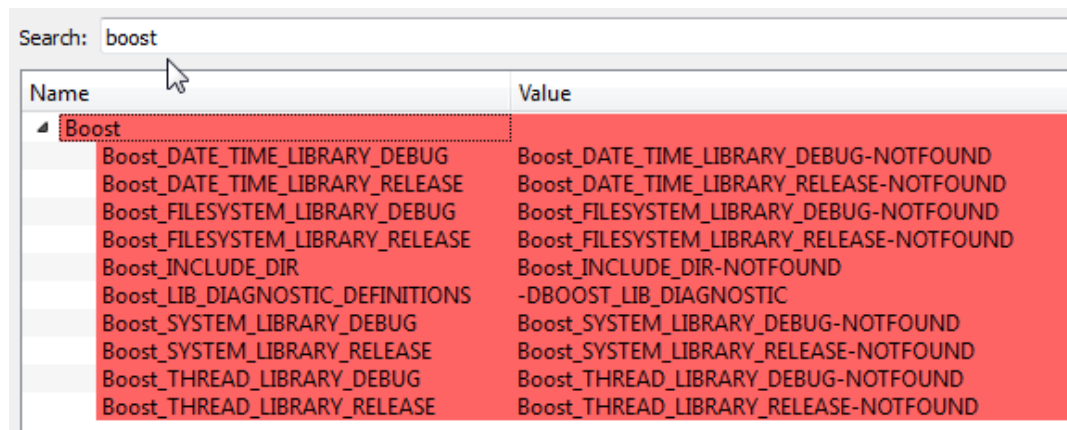
Before solving the VTK issue, let's organize the CMake variables in groups by checking the *Grouped* checkbox in the top right of CMake window. Let's check also the *Advanced* checkbox to show some advanced CMake variables. Now, if we want to look for a specific variable value, we can either browse the CMake variables to look for it, or we can use the *Search:* field to type the variable name.



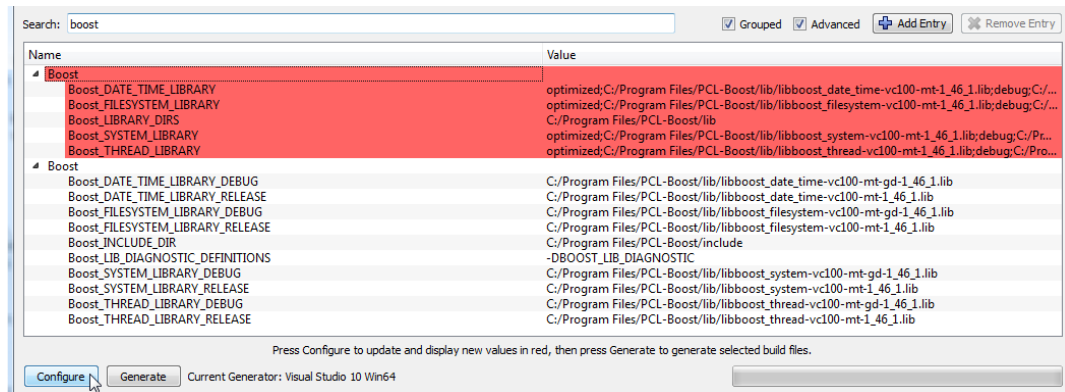
Let's check whether CMake did actually find the needed third party dependencies or not :

- **Boost :**

CMake was not able to find boost automatically. No problem, we will help it find it :) . If CMake has found your boost installation, then skip to the next bullet item.



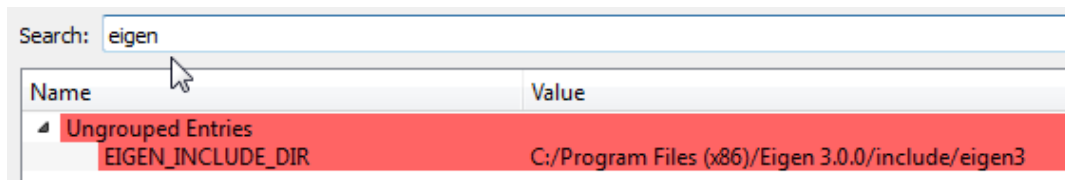
Let's tell CMake where boost headers are by specifying the headers path in **Boost\_INCLUDE\_DIR** variable. For example, my boost headers are in C:\Program Files\PCL-Boost\include (C:\Program Files\Boost\include for newer installers). Then, let's hit *configure* again ! Hopefully, CMake is now able to find all the other items (the libraries).



**Note:** This behaviour is not common for all libraries. Generally, if CMake is not able to find a specific library or package, we have to manually set the values of all the CMake related variables. Hopefully, the CMake script responsible of finding boost is able to find libraries using the headers path.

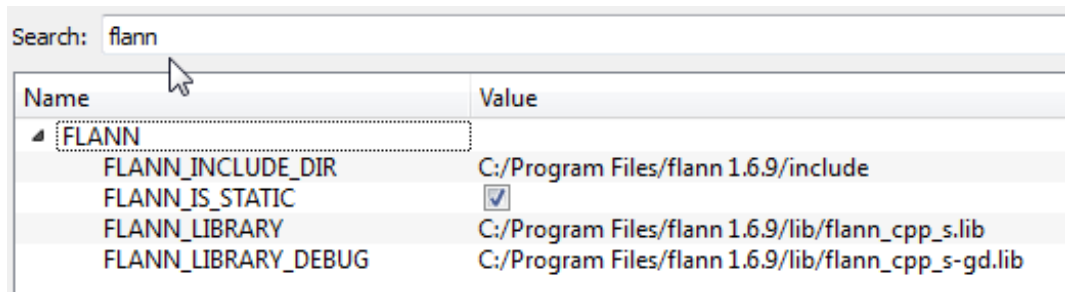
- **Eigen :**

Eigen is a header-only library, thus, we need only **EIGEN\_INCLUDE\_DIR** to be set. Hopefully, CMake did find Eigen.



- **FLANN :**

CMake was able to find my FLANN installation. By default on windows, PCL will pick the static FLANN libraries with **\_s** suffix. Thus, the **FLANN\_IS\_STATIC** checkbox is checked by default.



**Note:** If you rather want to use the **shared** FLANN libraries (those without the **\_s** suffix), you need to manually edit the **FLANN\_LIBRARY** and **FLANN\_LIBRARY\_DEBUG** variables to remove the **\_s** suffix and do not forget to uncheck **FLANN\_IS\_STATIC**. Make sure the FLANN dlls are either in your PATH or in the same folder as your executables.

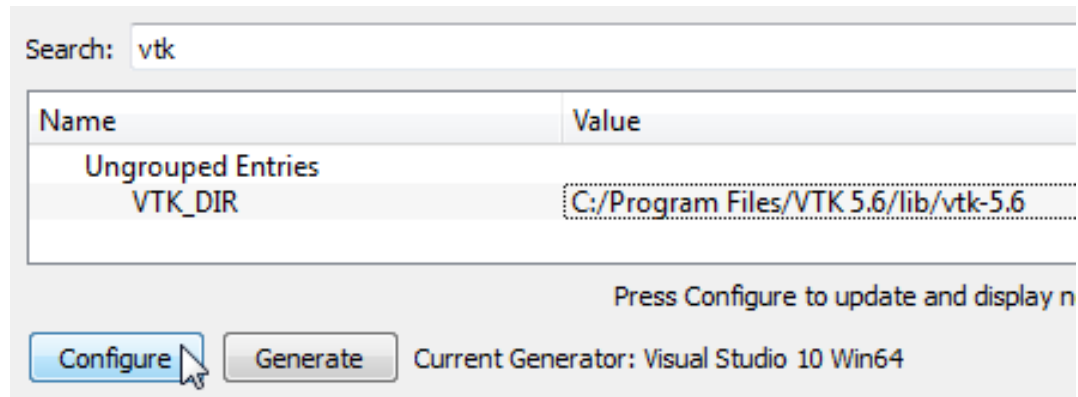
**Note:** In recent PCL, the **FLANN\_IS\_STATIC** checkbox no longer exists.

- Qt :

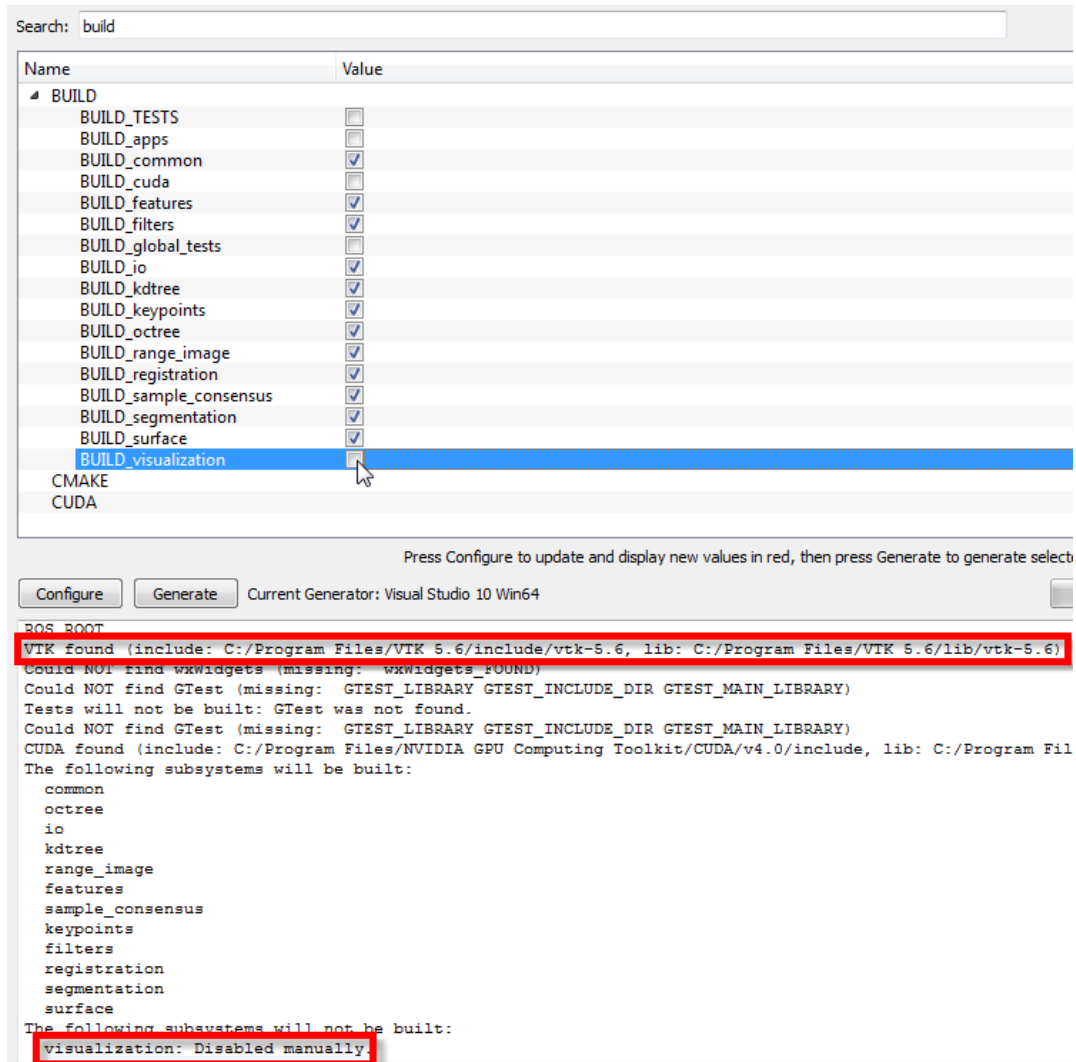
It is highly recommended to install Qt to the default path suggested by the installer. You need then to define an environment variable named **QTDIR** to point to Qt installation path (e.g. `C:\Qt\4.8.0`). Also, you need to append the bin folder to the **PATH** environment variable. Once you modify the environment variables, you need to restart CMake and click “Configure” again. If Qt is not found, you need at least to fill **QT\_QMAKE\_EXECUTABLE** CMake entry with the path of *qmake.exe* (e.g. `C:\Qt\4.8.0\bin\qmake.exe`), then click “Configure”.

- VTK :

CMake did not find my VTK installation. There is only one VTK related CMake variable called **VTK\_DIR**. We have to set it to the path of the folder containing **VTKConfig.cmake**, which is in my case : `C:\Program Files\VTK 5.6\lib\vtk-5.6` (`C:\Program Files\VTK 5.8.0\lib\vtk-5.8` for VTK 5.8). After you set **VTK\_DIR**, hit *configure* again.

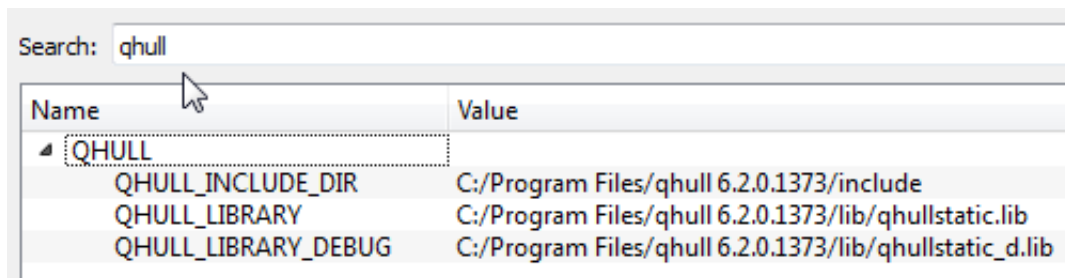


After clicking *configure*, in the logging window, we can see that VTK is found, but the *visualization* module is still disabled *manually*. We have then to enable it by checking the **BUILD\_visualization** checkbox. You can also do the same thing with the *apps* module. Then, hit *configure* again.



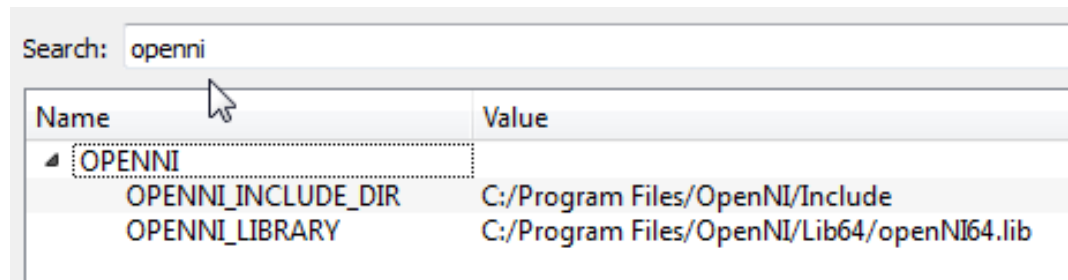
- QHull :

CMake was able to find my QHull installation. By default on windows, PCL will pick the static QHull libraries with *static* suffix.



- OpenNI :

CMake was able to find my OpenNI installation.



The screenshot shows the CMake GUI search results for the query 'openni'. A search bar at the top contains the text 'openni'. Below it, a table lists the search results. The first entry is 'OPENNI', which is expanded to show two sub-entries: 'OPENNI\_INCLUDE\_DIR' with the value 'C:/Program Files/OpenNI/Include' and 'OPENNI\_LIBRARY' with the value 'C:/Program Files/OpenNI/Lib64/openNI64.lib'.

Name	Value
OPENNI	
OPENNI_INCLUDE_DIR	C:/Program Files/OpenNI/Include
OPENNI_LIBRARY	C:/Program Files/OpenNI/Lib64/openNI64.lib

---

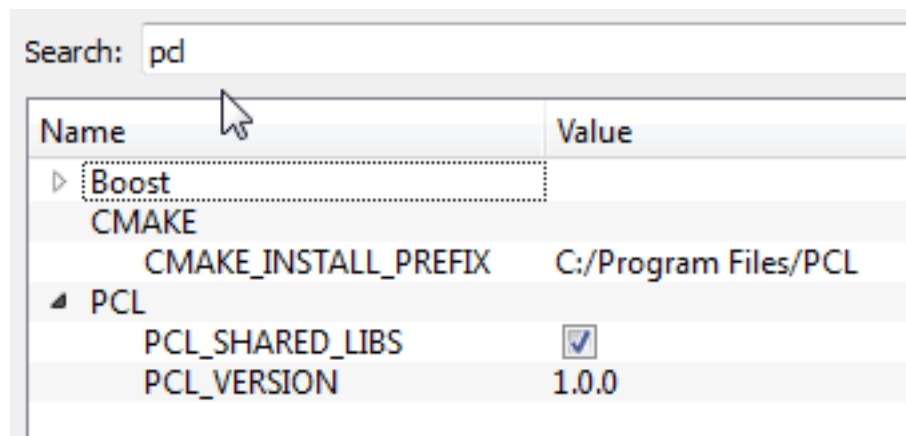
**Note:** CMake do not look for the installed OpenNI Sensor module. It is needed at runtime.

---

- **GTest :**

If you want to build PCL tests, you need to download GTest and build it yourself. In this tutorial, we will not build tests.

Once CMake has found all the needed dependencies, let's see the PCL specific CMake variables :



The screenshot shows the CMake GUI search results for the query 'pcl'. A search bar at the top contains the text 'pcl'. Below it, a table lists the search results. The first entry is 'Boost', followed by 'CMAKE', which is expanded to show 'CMAKE\_INSTALL\_PREFIX' with the value 'C:/Program Files/PCL'. The next entry is 'PCL', which is expanded to show 'PCL\_SHARED\_LIBS' with a checked checkbox and 'PCL\_VERSION' with the value '1.0.0'.

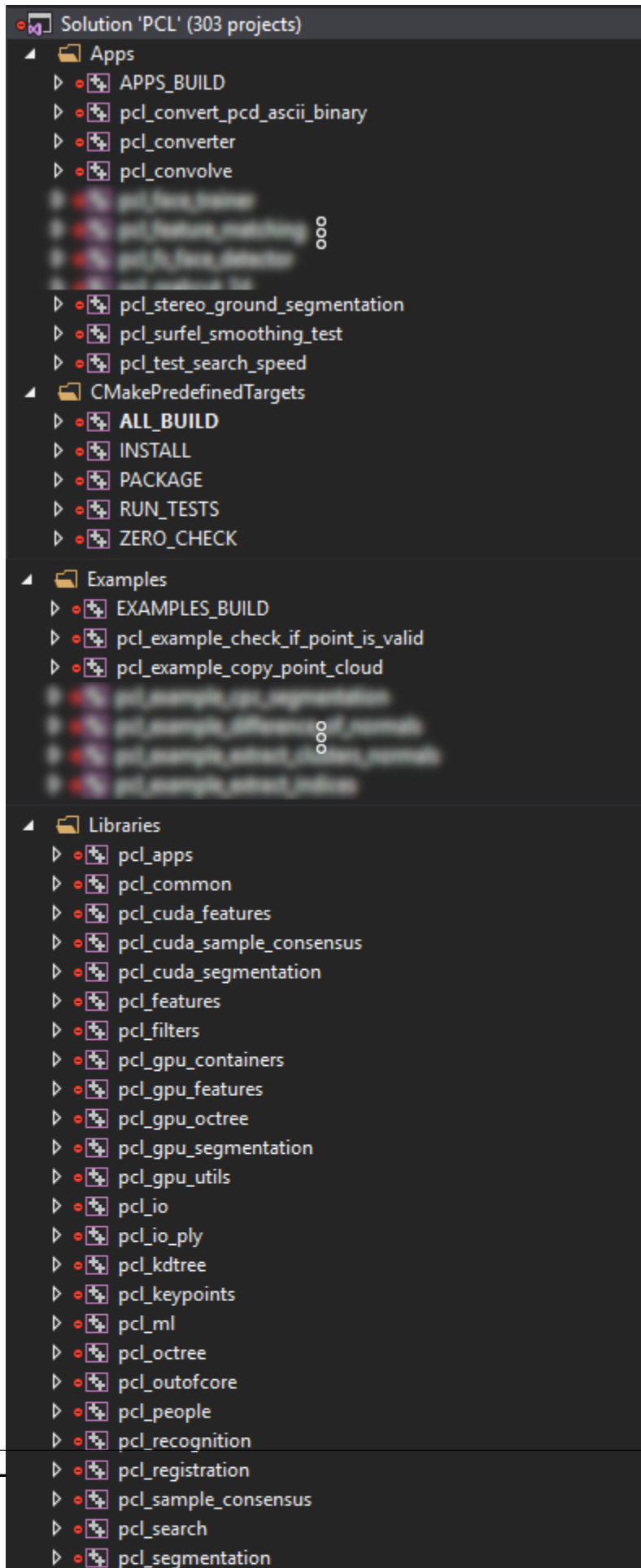
Name	Value
Boost	
CMAKE	
CMAKE_INSTALL_PREFIX	C:/Program Files/PCL
PCL	
PCL_SHARED_LIBS	<input checked="" type="checkbox"/>
PCL_VERSION	1.0.0

- **PCL\_SHARED\_LIBS** is checked by default. Uncheck it if you want static PCL libs (not recommended).
- **CMAKE\_INSTALL\_PREFIX** is where PCL will be installed after building it (more information on this later).

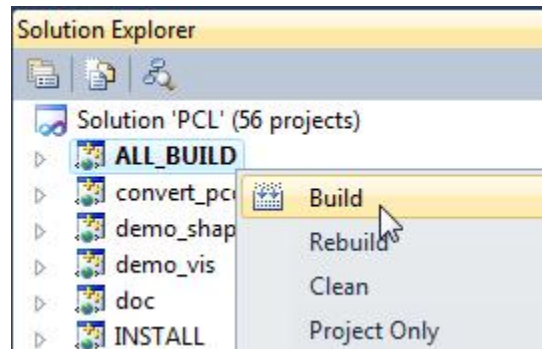
Once PCL configuration is ok, hit the *Generate* button. CMake will then generate Visual Studio project files (vcproj files) and the main solution file (PCL.sln) in C:\PCL directory.

## 7.4 Building PCL

Open that generated solution file (PCL.sln) to finally build the PCL libraries. This is how your solution will look like.



Building the “ALL\_BUILD” project will build everything.



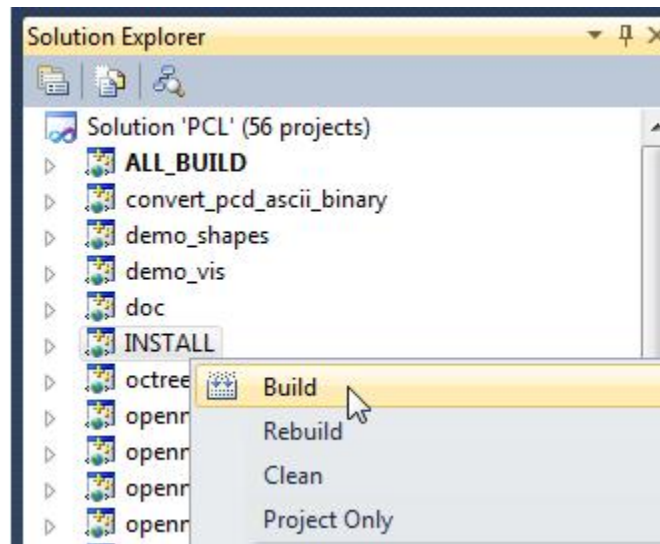
---

**Note:** Make sure to build the “ALL\_BUILD” project in both **debug** and **release** mode.

---

## 7.5 Installing PCL

To install the built libraries and executables, you need to build the “INSTALL” project in the solution explorer. This utility project will copy PCL headers, libraries and executable to the directory defined by the `CMAKE_INSTALL_PREFIX` CMake variable.



---

**Note:** Make sure to build the “INSTALL” project in both **debug** and **release** mode.

---

---

**Note:** It is highly recommended to add the bin folder in PCL installation tree (e.g. `C:\Program Files\PCL\bin`) to your **PATH** environment variable.

---

## 7.6 Advanced topics

- **Building PCL Tests :**

If you want to build PCL tests, you need to download *GTest* 1.6 (<http://code.google.com/p/googletest/>) and build it yourself. Make sure, when you configure GTest via CMake to check the **gtest\_force\_shared\_crt** checkbox. You need, as usual, to build *GTest* in both **release** and **debug**.

Back to PCL's CMake settings, you have to fill the **GTEST\_\*** CMake entries (include directory, gtest libraries (debug and release) and gtestmain libraries (debug and release)). Then, you have to check **BUILD\_TEST** and **BUILD\_global\_tests** CMake checkboxes, and hit *Configure* and *Generate*.

- **Building the documentation :**

You can build the doxygen documentation of PCL in order to have a local up-to-date api documentation. For this, you need Doxygen (<http://www.doxygen.org>). You will need also the Graph Visualization Software (GraphViz, <http://www.graphviz.org/>) to get the doxygen graphics, specifically the *dot* executable.

Once you installed these two packages, hit *Configure*. Three CMake variables should be set (if CMake cannot find them, you can fill them manually) :

- **DOXYGEN\_EXECUTABLE** : path to *doxygen.exe* (e.g. C:/Program Files (x86)/doxygen/bin/doxygen.exe)
- **DOXYGEN\_DOT\_EXECUTABLE** : path to *dot.exe* from GraphViz (e.g. C:/Program Files (x86)/Graphviz2.26.3/bin/dot.exe)
- **DOXYGEN\_DOT\_PATH** : path of the folder containing *dot.exe* from GraphViz (e.g. C:/Program Files (x86)/Graphviz2.26.3/bin)

Then, you need to enable the *documentation* project in Visual Studio by checking the **BUILD\_DOCUMENTATION** checkbox in CMake.

You can also build one single CHM file that will gather all the generated html files into one file. You need the [Microsoft HTML HELP Workshop](#). After you install the *Microsoft HTML HELP Workshop*, hit *Configure*. If CMake is not able to find **HTML\_HELP\_COMPILER**, then fill it manually with the path to *hhc.exe* (e.g. C:/Program Files (x86)/HTML Help Workshop/hhc.exe), then click *Configure* and *Generate*.

Now, in PCL Visual Studio solution, you will have a new project called *doc*. To generate the documentation files, right click on it, and choose *Build*. Then, you can build the *INSTALL* project so that the generated documentation files get copied to **CMAKE\_INSTALL\_PREFIX/PCL/share/doc/pcl/html** folder (e.g. C:\Program Files\PCL\share\doc\pcl\html).

## 7.7 Using PCL

We finally managed to compile the Point Cloud Library (PCL) as binaries for Windows. You can start using them in your project by following the [Using PCL in your own project](#) tutorial.



---

# Compiling PCL and its dependencies from MacPorts and source on Mac OS X

---

This tutorial explains how to build the Point Cloud Library **from MacPorts and source** on Mac OS X platforms, and tries to guide you through the download and building *of all the required dependencies*.



### Contents

- *Compiling PCL and its dependencies from MacPorts and source on Mac OS X*
  - *Prerequisites*
  - *PCL Dependencies*

- \* *Required*
- \* *Optional*
- \* *Advanced (Developers)*
- *Building, Compiling and Installing PCL Dependencies*
  - \* *Install CMake*
  - \* *Install Boost*
  - \* *Install Eigen*
  - \* *Install FLANN*
  - \* *Install VTK*
  - \* *Install Qhull*
  - \* *Install libusb*
  - \* *Install Patched OpenNI and Sensor*
- *Building PCL*
- *Using PCL*
- *Advanced (Developers)*
  - \* *Testing (googletest)*
  - \* *API Documentation (Doxygen)*
  - \* *Tutorials (Sphinx)*

## 8.1 Prerequisites

Before getting started download and install the following prerequisites for Mac OS X:

- **XCode** (<https://developer.apple.com/xcode/>) Apple’s powerful integrated development environment
- **MacPorts** (<http://www.macports.org>) An open-source community initiative to design an easy-to-use system for compiling, installing, and upgrading either command-line, X11 or Aqua based open-source software on the Mac OS X operating system.

## 8.2 PCL Dependencies

In order to compile every component of the PCL library we need to download and compile a series of 3rd party library dependencies. We’ll cover the building, compiling and installing of everything in the following sections:

### 8.2.1 Required

The following libraries are **Required** to build PCL.

- **CMake version >= 3.5.0** (<http://www.cmake.org>) Cross-platform, open-source build system.

---

**Note:** Though not a dependency per se, the PCL community relies heavily on CMake for the libraries build process.

---

- **Boost version  $\geq 1.46.1$**  (<http://www.boost.org/>) Provides free peer-reviewed portable C++ source libraries. Used for shared pointers, and threading.
- **Eigen version  $\geq 3.0.0$**  (<http://eigen.tuxfamily.org/>) Unified matrix library. Used as the matrix backend for SSE optimized math.
- **FLANN version  $\geq 1.6.8$**  (<http://www.cs.ubc.ca/research/flann/>) Library for performing fast approximate nearest neighbor searches in high dimensional spaces. Used in *kdtree* for fast approximate nearest neighbors search.
- **Visualization ToolKit (VTK) version  $\geq 5.6.1$**  (<http://www.vtk.org/>) Software system for 3D computer graphics, image processing and visualization. Used in *visualization* for 3D point cloud rendering and visualization.

## 8.2.2 Optional

The following libraries are **Optional** and provide extended functionality within PCL, ie Kinect support.

- **Qhull version  $\geq 2011.1$**  (<http://www.qhull.org/>) computes the convex hull, Delaunay triangulation, Voronoi diagram, halfspace intersection about a point, furthest-site Delaunay triangulation, and furthest-site Voronoi diagram. Used for convex/concave hull decompositions in *surface*.
- **libusb** (<http://www.libusb.org/>) A library that gives user level applications uniform access to USB devices across many different operating systems.
- **PCL Patched OpenNI/Sensor** (<http://www.openni.org/>) The OpenNI Framework provides the interface for physical devices and for middleware components. Used to grab point clouds from OpenNI compliant devices.

## 8.2.3 Advanced (Developers)

The following libraries are **Advanced** and provide additional functionality for PCL developers:

- **googletest version  $\geq 1.6.0$**  (<http://code.google.com/p/googletest/>) Google's framework for writing C++ tests on a variety of platforms. Used to build test units.
- **Doxygen** (<http://www.doxygen.org>) A documentation system for C++, C, Java, Objective-C, Python, IDL (Corba and Microsoft flavors), Fortran, VHDL, PHP, C#, and to some extent D.
- **Sphinx** (<http://sphinx-doc.org/>) A tool that makes it easy to create intelligent and beautiful documentation.

## 8.3 Building, Compiling and Installing PCL Dependencies

By now you should have downloaded and installed the latest versions of XCode and MacPorts under the *Prerequisites* section. We'll be installing most dependencies available via MacPorts and the rest will be built from source.

### 8.3.1 Install CMake

```
$ sudo port install cmake
```

### 8.3.2 Install Boost

```
$ sudo port install boost
```

### 8.3.3 Install Eigen

```
$ sudo port install eigen3
```

### 8.3.4 Install FLANN

```
$ sudo port install flann
```

### 8.3.5 Install VTK

To install via MacPorts:

```
$ sudo port install vtk5 +qt4_mac
```

To install from source download the source from <http://www.vtk.org/VTK/resources/software.html>

Follow the README.html for compiling on UNIX / Cygwin / Mac OSX:

```
$ cd VTK
$ mkdir VTK-build
$ cd VTK-build
$ cmake ../VTK
```

**Within the CMake configuration:** Press [c] for initial configuration

Press [t] to get into advanced mode and change the following:

```
VTK_USE_CARBON:OFF
VTK_USE_COCOA:ON
VTK_USE_X:OFF
```

---

**Note:** VTK *must* be built with Cocoa support and *must* be installed, in order for the visualization module to be able to compile. If you do not require visualisation, you may omit this step.

---

Press [g] to generate the make files.

Press [q] to quit.

Then run:

```
$ make && make install
```

### 8.3.6 Install Qhull

```
$ sudo port install qhull
```

### 8.3.7 Install libusb

```
$ sudo port install libusb-devel +universal
```

### 8.3.8 Install Patched OpenNI and Sensor

Download the patched versions of OpenNI and Sensor from the PCL downloads page <http://pointclouds.org/downloads/macosx.html>

Extract, build, fix permissions and install OpenNI:

```
$ unzip openni_osx.zip -d openni_osx
$ cd openni_osx/Redist
$ chmod -R a+r Bin Include Lib
$ chmod -R a+x Bin Lib
$ chmod a+x Include/MacOSX Include/Linux-*
$ sudo ./install
```

In addition the following primesense xml config found within the patched OpenNI download needs its permissions fixed and copied to the correct location to for the Kinect to work on Mac OS X:

```
$ chmod a+r openni_osx/Redist/Samples/Config/SamplesConfig.xml
$ sudo cp openni_osx/Redist/Samples/Config/SamplesConfig.xml /etc/primesense/
```

Extract, build, fix permissions and install Sensor:

```
$ unzip ps_engine_osx.zip -d ps_engine_osx
$ cd ps_engine_osx/Redist
$ chmod -R a+r Bin Lib Config Install
$ chmod -R a+x Bin Lib
$ sudo ./install
```

## 8.4 Building PCL

At this point you should have everything needed installed to build PCL with almost no additional configuration.

Checkout the PCL source from the Github:

```
$ git clone https://github.com/PointCloudLibrary/pcl $ cd pcl
```

Create the build directories, configure CMake, build and install:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
$ sudo make install
```

The customization of the build process is out of the scope of this tutorial and is covered in greater detail in the *Customizing the PCL build process* tutorial.

## 8.5 Using PCL

We finally managed to compile the Point Cloud Library (PCL) for Mac OS X. You can start using them in your project by following the *Using PCL in your own project* tutorial.

## 8.6 Advanced (Developers)

### 8.6.1 Testing (googletest)

### 8.6.2 API Documentation (Doxygen)

Install Doxygen via MacPorts:

```
$ sudo port install doxygen
```

Or install the Prebuilt binary for Mac OS X (<http://www.stack.nl/~dimitri/doxygen/download.html#latestsrc>)

After installed you can build the documentation:

```
$ make doc
```

### 8.6.3 Tutorials (Sphinx)

In addition to the API documentation there is also tutorial documentation built using Sphinx. The easiest way to get this installed is using python's *easy\_install*:

```
$ easy_install -U Sphinx
```

The Sphinx documentation also requires the third party contrib extension *sphinxcontrib-doxylink* (<https://pypi.python.org/pypi/sphinxcontrib-doxylink>) to reference the Doxygen built documentation.

To install from source you'll also need Mercurial:

```
$ sudo port install mercurial
$ hg clone http://bitbucket.org/birkenfeld/sphinx-contrib
$ cd sphinx-contrib/doxylink
$ python setup.py install
```

After installed you can build the tutorials:

```
$ make Tutorials
```

---

**Note:** Sphinx can be installed via MacPorts but is a bit of a pain getting all the PYTHON\_PATH's in order

---

---

### Installing on Mac OS X using Homebrew

---

This tutorial explains how to install the Point Cloud Library on Mac OS X using Homebrew.



#### Contents

- *Installing on Mac OS X using Homebrew*
  - *Prerequisites*
  - *Using the formula*
  - *Using PCL*

## 9.1 Prerequisites

You will need to have Homebrew installed. If you do not already have a Homebrew installation, see the [Homebrew homepage](#) for installation instructions.

## 9.2 Using the formula

The PCL formula is in the Homebrew official repositories. This will automatically install all necessary dependencies and provides options for controlling which parts of PCL are installed.

---

**Note:** To prepare it, follow these steps:

1. Install Homebrew. See the Homebrew website for instructions.
  2. Execute `brew update`.
  3. Execute `brew tap homebrew/science`.
- 

To install the latest version using the formula, execute the following command:

```
$ brew install pcl
```

You can specify options to control which parts of PCL are installed. For example, to build just the libraries without extra dependencies, execute the following command:

```
$ brew install pcl --without-apps --without-tools --without-vtk --without-qt
```

For a full list of the available options, see the formula's help:

```
$ brew options pcl
```

Once PCL is installed, you may wish to periodically upgrade it. Update Homebrew and, if a PCL update is available, upgrade:

```
$ brew update
$ brew upgrade pcl
```

## 9.3 Using PCL

Now that PCL is installed, you can start using the library in your own projects by following the *Using PCL in your own project* tutorial.

# CHAPTER 10

---

## Using PCL with Eclipse

---

This tutorial explains how to use Eclipse as an IDE to manage your PCL projects. It was tested under Ubuntu 14.04 with Eclipse Luna; do not hesitate to modify this tutorial by submitting a pull request on GitHub to add other configurations etc.

### Contents

- *Using PCL with Eclipse*
  - *Prerequisites*
  - *Creating the eclipse project files*
  - *Importing into Eclipse*
  - *Configuring Eclipse*
  - *Setting the PCL code style in Eclipse*
    - \* *Global*
    - \* *Project specific*
    - \* *How to format the code*
  - *Launching the program*
  - *Where to get more information*

## 10.1 Prerequisites

We assume you have downloaded and extracted a PCL version (either PCL trunk or a stable version) on your machine. For the example, we will use the `pcl_visualizer` code.

## 10.2 Creating the eclipse project files

The files are organized like the following tree:

```

.
├── build
└── src
    ├── CMakeLists.txt
    └── pcl_visualizer_demo.cpp

```

Open a terminal, navigate to your project root folder and configure the project:

```

$ cd /path_to_my_project/build
$ cmake -G "Eclipse CDT4 - Unix Makefiles" ../src

```

You will see something that should look like:

```

-- The C compiler identification is GNU 4.8.2
-- The CXX compiler identification is GNU 4.8.2
-- Could not determine Eclipse version, assuming at least 3.6 (Helios). Adjust CMAKE_
↪ECLIPSE_VERSION if this is wrong.
-- Check for working C compiler: /usr/lib/ccache/cc
-- Check for working C compiler: /usr/lib/ccache/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/lib/ccache/c++
-- Check for working CXX compiler: /usr/lib/ccache/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- checking for module 'eigen3'
-- found eigen3, version 3.2.0
-- Found eigen: /usr/include/eigen3
-- Boost version: 1.54.0
-- Found the following Boost libraries:
-- system
-- filesystem
-- thread
-- date_time
-- iostreams
-- mpi
-- serialization
-- chrono
-- checking for module 'openni-dev'
-- package 'openni-dev' not found
-- Found oppenni: /usr/lib/libOpenNI.so
-- checking for module 'openni2-dev'
-- package 'openni2-dev' not found
-- Found OpenNI2: /usr/lib/libOpenNI2.so
** WARNING ** io features related to pcap will be disabled
** WARNING ** io features related to png will be disabled
-- Found libusb-1.0: /usr/include
-- checking for module 'flann'
-- found flann, version 1.8.4
-- Found Flann: /usr/lib/x86_64-linux-gnu/libflann_cpp_s.a
-- Found qhull: /usr/lib/x86_64-linux-gnu/libqhull.so
-- checking for module 'openni-dev'
-- package 'openni-dev' not found

```

(continues on next page)

(continued from previous page)

```

-- checking for module 'openni2-dev'
--   package 'openni2-dev' not found
-- looking for PCL_COMMON
-- Found PCL_COMMON: /usr/local/lib/libpcl_common.so
-- looking for PCL_OCTREE
-- Found PCL_OCTREE: /usr/local/lib/libpcl_octree.so
-- looking for PCL_IO
-- Found PCL_IO: /usr/local/lib/libpcl_io.so
-- looking for PCL_KDTREE
-- Found PCL_KDTREE: /usr/local/lib/libpcl_kdtree.so
-- looking for PCL_SEARCH
-- Found PCL_SEARCH: /usr/local/lib/libpcl_search.so
-- looking for PCL_SAMPLE_CONSENSUS
-- Found PCL_SAMPLE_CONSENSUS: /usr/local/lib/libpcl_sample_consensus.so
-- looking for PCL_FILTERS
-- Found PCL_FILTERS: /usr/local/lib/libpcl_filters.so
-- looking for PCL_2D
-- Found PCL_2D: /usr/local/include/pcl-1.7
-- looking for PCL_FEATURES
-- Found PCL_FEATURES: /usr/local/lib/libpcl_features.so
-- looking for PCL_GEOMETRY
-- Found PCL_GEOMETRY: /usr/local/include/pcl-1.7
-- looking for PCL_KEYPOINTS
-- Found PCL_KEYPOINTS: /usr/local/lib/libpcl_keypoints.so
-- looking for PCL_SURFACE
-- Found PCL_SURFACE: /usr/local/lib/libpcl_surface.so
-- looking for PCL_REGISTRATION
-- Found PCL_REGISTRATION: /usr/local/lib/libpcl_registration.so
-- looking for PCL_ML
-- Found PCL_ML: /usr/local/lib/libpcl_ml.so
-- looking for PCL_SEGMENTATION
-- Found PCL_SEGMENTATION: /usr/local/lib/libpcl_segmentation.so
-- looking for PCL_RECOGNITION
-- Found PCL_RECOGNITION: /usr/local/lib/libpcl_recognition.so
-- looking for PCL_VISUALIZATION
-- Found PCL_VISUALIZATION: /usr/local/lib/libpcl_visualization.so
-- looking for PCL_PEOPLE
-- Found PCL_PEOPLE: /usr/local/lib/libpcl_people.so
-- looking for PCL_OUTOFCORE
-- Found PCL_OUTOFCORE: /usr/local/lib/libpcl_outofcore.so
-- looking for PCL_TRACKING
-- Found PCL_TRACKING: /usr/local/lib/libpcl_tracking.so
-- looking for PCL_STEREO
-- Found PCL_STEREO: /usr/local/lib/libpcl_stereo.so
-- looking for PCL_GPU_CONTAINERS
-- Found PCL_GPU_CONTAINERS: /usr/local/lib/libpcl_gpu_containers.so
-- looking for PCL_GPU_UTILS
-- Found PCL_GPU_UTILS: /usr/local/lib/libpcl_gpu_utils.so
-- looking for PCL_GPU_OCTREE
-- Found PCL_GPU_OCTREE: /usr/local/lib/libpcl_gpu_octree.so
-- looking for PCL_GPU_FEATURES
-- Found PCL_GPU_FEATURES: /usr/local/lib/libpcl_gpu_features.so
-- looking for PCL_GPU_KINFU
-- Found PCL_GPU_KINFU: /usr/local/lib/libpcl_gpu_kinfu.so
-- looking for PCL_GPU_KINFU_LARGE_SCALE
-- Found PCL_GPU_KINFU_LARGE_SCALE: /usr/local/lib/libpcl_gpu_kinfu_large_scale.so
-- looking for PCL_GPU_SEGMENTATION

```

(continues on next page)

(continued from previous page)

```
-- Found PCL_GPU_SEGMENTATION: /usr/local/lib/libpcl_gpu_segmentation.so
-- looking for PCL_CUDA_COMMON
-- Found PCL_CUDA_COMMON: /usr/local/include/pcl-1.7
-- looking for PCL_CUDA_FEATURES
-- Found PCL_CUDA_FEATURES: /usr/local/lib/libpcl_cuda_features.so
-- looking for PCL_CUDA_SEGMENTATION
-- Found PCL_CUDA_SEGMENTATION: /usr/local/lib/libpcl_cuda_segmentation.so
-- looking for PCL_CUDA_SAMPLE_CONSENSUS
-- Found PCL_CUDA_SAMPLE_CONSENSUS: /usr/local/lib/libpcl_cuda_sample_consensus.so
-- Found PCL: /usr/lib/x86_64-linux-gnu/libboost_system.so;/usr/lib/x86_64-linux-gnu/
↳ libboost_filesystem.so;/usr/lib/x86_64-linux-gnu/libboost_thread.so;/usr/lib/x86_64-
↳ linux-gnu/libboost_date_time.so;/usr/lib/x86_64-linux-gnu/libboost_iostreams.so;/
↳ usr/lib/x86_64-linux-gnu/libboost_mpi.so;/usr/lib/x86_64-linux-gnu/libboost_
↳ serialization.so;/usr/lib/x86_64-linux-gnu/libboost_chrono.so;/usr/lib/x86_64-linux-
↳ gnu/libpthread.so;optimized;/usr/local/lib/libpcl_common.so;debug;/usr/local/lib/
↳ libpcl_common.so;optimized;/usr/local/lib/libpcl_octree.so;debug;/usr/local/lib/
↳ libpcl_octree.so;/usr/lib/libOpenNI.so;/usr/lib/libOpenNI2.so;vtkCommon;
↳ vtkFiltering;vtkImaging;vtkGraphics;vtkGenericFiltering;vtkIO;vtkRendering;
↳ vtkVolumeRendering;vtkHybrid;vtkWidgets;vtkParallel;vtkInfovis;vtkGeovis;vtkViews;
↳ vtkCharts;optimized;/usr/local/lib/libpcl_io.so;debug;/usr/local/lib/libpcl_io.so;
↳ optimized;/usr/lib/x86_64-linux-gnu/libflann_cpp_s.a;debug;/usr/lib/x86_64-linux-
↳ gnu/libflann_cpp_s.a;optimized;/usr/local/lib/libpcl_kdtree.so;debug;/usr/local/lib/
↳ libpcl_kdtree.so;optimized;/usr/local/lib/libpcl_search.so;debug;/usr/local/lib/
↳ libpcl_search.so;optimized;/usr/local/lib/libpcl_sample_consensus.so;debug;/usr/
↳ local/lib/libpcl_sample_consensus.so;optimized;/usr/local/lib/libpcl_filters.so;
↳ debug;/usr/local/lib/libpcl_filters.so;optimized;/usr/local/lib/libpcl_features.so;
↳ debug;/usr/local/lib/libpcl_features.so;optimized;/usr/local/lib/libpcl_keypoints.
↳ so;debug;/usr/local/lib/libpcl_keypoints.so;optimized;/usr/lib/x86_64-linux-gnu/
↳ libqhull.so;debug;/usr/lib/x86_64-linux-gnu/libqhull.so;optimized;/usr/local/lib/
↳ libpcl_surface.so;debug;/usr/local/lib/libpcl_surface.so;optimized;/usr/local/lib/
↳ libpcl_registration.so;debug;/usr/local/lib/libpcl_registration.so;optimized;/usr/
↳ local/lib/libpcl_ml.so;debug;/usr/local/lib/libpcl_ml.so;optimized;/usr/local/lib/
↳ libpcl_segmentation.so;debug;/usr/local/lib/libpcl_segmentation.so;optimized;/usr/
↳ local/lib/libpcl_recognition.so;debug;/usr/local/lib/libpcl_recognition.so;
↳ optimized;/usr/local/lib/libpcl_visualization.so;debug;/usr/local/lib/libpcl_
↳ visualization.so;optimized;/usr/local/lib/libpcl_people.so;debug;/usr/local/lib/
↳ libpcl_people.so;optimized;/usr/local/lib/libpcl_outofcore.so;debug;/usr/local/lib/
↳ libpcl_outofcore.so;optimized;/usr/local/lib/libpcl_tracking.so;debug;/usr/local/
↳ lib/libpcl_tracking.so;optimized;/usr/local/lib/libpcl_stereo.so;debug;/usr/local/
↳ lib/libpcl_stereo.so;optimized;/usr/local/lib/libpcl_gpu_containers.so;debug;/usr/
↳ local/lib/libpcl_gpu_containers.so;optimized;/usr/local/lib/libpcl_gpu_utils.so;
↳ debug;/usr/local/lib/libpcl_gpu_utils.so;optimized;/usr/local/lib/libpcl_gpu_octree.
↳ so;debug;/usr/local/lib/libpcl_gpu_octree.so;optimized;/usr/local/lib/libpcl_gpu_
↳ features.so;debug;/usr/local/lib/libpcl_gpu_features.so;optimized;/usr/local/lib/
↳ libpcl_gpu_kinfu.so;debug;/usr/local/lib/libpcl_gpu_kinfu.so;optimized;/usr/local/
↳ lib/libpcl_gpu_kinfu_large_scale.so;debug;/usr/local/lib/libpcl_gpu_kinfu_large_
↳ scale.so;optimized;/usr/local/lib/libpcl_gpu_segmentation.so;debug;/usr/local/lib/
↳ libpcl_gpu_segmentation.so;optimized;/usr/local/lib/libpcl_cuda_features.so;debug;/
↳ usr/local/lib/libpcl_cuda_features.so;optimized;/usr/local/lib/libpcl_cuda_
↳ segmentation.so;debug;/usr/local/lib/libpcl_cuda_segmentation.so;optimized;/usr/
↳ local/lib/libpcl_cuda_sample_consensus.so;debug;/usr/local/lib/libpcl_cuda_sample_
↳ consensus.so;/usr/lib/x86_64-linux-gnu/libboost_system.so;/usr/lib/x86_64-linux-gnu/
↳ libboost_filesystem.so;/usr/lib/x86_64-linux-gnu/libboost_thread.so;/usr/lib/x86_64-
↳ linux-gnu/libboost_date_time.so;/usr/lib/x86_64-linux-gnu/libboost_iostreams.so;/
↳ usr/lib/x86_64-linux-gnu/libboost_mpi.so;/usr/lib/x86_64-linux-gnu/libboost_
↳ serialization.so;/usr/lib/x86_64-linux-gnu/libboost_chrono.so;/usr/lib/x86_64-linux-
↳ gnu/libpthread.so;optimized;/usr/lib/x86_64-linux-gnu/libqhull.so;debug;/usr/lib/
↳ x86_64-linux-gnu/libqhull.so;/usr/lib/libOpenNI.so;/usr/lib/libOpenNI2.so;/usr/lib/
↳ x86_64-linux-gnu/libflann_cpp_s.a;debug;/usr/lib/x86_64-linux-gnu/libflann_
↳ cpp_s.a;vtkCommon;vtkFiltering;vtkImaging;vtkGraphics;vtkGenericFiltering;vtkIO;
↳ vtkRendering;vtkVolumeRendering;vtkHybrid;vtkWidgets;vtkParallel;vtkInfovis;
↳ vtkGeovis;vtkViews;vtkCharts (Required is at least version "1.7")
```

(continued from previous page)

```
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dell/visualizer/build
```

## 10.3 Importing into Eclipse

- Launch [Eclipse CDT](#) and select `File > Import`.
- In the list select `General > Existing Projects into Workspace` and then next.
- Browse (Select root directory) to the root folder of the project and select the build folder (in the example case, `/home/dell/visualizer/build`).
- Click `Finish`.

**Warning:** The Eclipse indexer is going to parse the files in the project (and all the includes), this can take a lot of time and might crash Eclipse if it's not configured for big projects. Take a look at the bottom right of Eclipse's window to see the indexer status; it is advised not to do anything until the indexer has finished it's job.

## 10.4 Configuring Eclipse

If Eclipse fails to open your PCL project you might need to change Eclipse configuration; here are some values that should solve all problems (but might not work on light hardware configurations):

```
$ sudo gedit /usr/lib/eclipse/eclipse.ini
```

Change the values in the last lines:

```
org.eclipse.platform
--launcher.XXMaxPermSize
1024m
--launcher.defaultAction
openFile
--launcher.appendVmargs
-vmargs
-Dosgi.requiredJavaVersion=1.7
-XX:MaxPermSize=512m
-Xms1024m
-Xmx1024m
```

Restart Eclipse and go to `Windows > Preferences`, then `C/C++ > Indexer > Cache Limits`. Set the limits to `[50% | 512 | 512]`.

## 10.5 Setting the PCL code style in Eclipse

You can find a PCL code style file for Eclipse in [PCL GitHub trunk](#)

### 10.5.1 Global

If you want to apply the PCL style guide to all projects: `Windows > Preferences > C/C++ > Code Style > Formatter`

### 10.5.2 Project specific

If you want to apply the style guide only to one project: Go to `Project > Properties`, then select `Code Style` in the left field and `Enable project specific settings`, then `Import` and select where your profile file (.xml) is.

### 10.5.3 How to format the code

If you want to format the whole project use `Source > Format`. If you want to format only your selection use the shortcut `Ctrl + Shift + F`

## 10.6 Launching the program

To build the project, click on the build icon

- Create a launch configuration, select the project on the left panel (left click on the project name); `Run > Run Configurations...`
- Create a new `C/C++ Application` click on `Select Project` and choose the executable to be launched.
- Go the second tab (`Arguments`) and enter your arguments; remember this is not a terminal and `~` won't work to get to your home folder for example !

Run the program by clicking on the run icon

The Eclipse console doesn't manage ANSI colours, you could use [an ANSI console plugin](#) to get rid of the “[0m” characters in the output.

## 10.7 Where to get more information

You can get more information about the Eclipse CDT4 Generator [here](#).

---

## Generate a local documentation for PCL

---

For practical reasons you might want to have a local documentation which corresponds to your PCL version. In this tutorial you will learn how to generate it and how to set up Apache so that the search bar works.

This tutorial was written for Ubuntu 12.04 and 14.04, feel free to edit it on GitHub to add your platform.

### Contents

- *Generate a local documentation for PCL*
  - *Dependencies*
  - *Generate the documentation*
  - *Installing and configuring Apache*

## 11.1 Dependencies

You need to install a few dependencies in order to be able to generate the documentation:

```
$ sudo apt-get install doxygen graphviz sphinx3 python-pip
$ sudo pip install sphinxcontrib-doxylink
```

## 11.2 Generate the documentation

Go into the build folder of PCL where you've configured it ([see tutorial](#)) and enter:

```
$ make doc
```

Then you can open the documentation with your browser, for example:

```
$ firefox doc/doxygen/html/index.html
```

The documentation has been generated in your PCL build directory but it is not installed; if you wish to install it just do:

```
$ sudo make install
```

The default PCL CMAKE\_INSTALL\_PREFIX is /usr/local, this means the documentation will be located in /usr/local/share/doc/pcl-1.7/html/index.html

---

**Note:** You will quickly notice that the search bar doesn't work! (searching opens "search.php" instead of searching)

---

## 11.3 Installing and configuring Apache

Apache ([The Apache HTTP Server](#)) is a web server application, in this section you will learn how to configure Apache in order to be able to use the search feature within your offline documentation.

First you need to install Apache and php:

```
$ sudo apt-get install apache2 php5 libapache2-mod-php5
```

Then you need to edit the default website location:

```
$ sudo gedit /etc/apache2/sites-available/000-default.conf
```

Change DocumentRoot (default = /var/www/html) to /usr/local/share/doc/pcl-1.7/html/ (or your local PCL doc build path)

After that change the Apache directory options:

```
$ sudo gedit +153 /etc/apache2/apache2.conf
```

Replace the paragraph at line 153 with:

```
<Directory />
    #Options FollowSymLinks
    Options Indexes FollowSymLinks Includes ExecCGI
    AllowOverride All
    Order deny,allow
    Allow from all
</Directory>
```

Restart Apache and the search bar will now work if you open localhost:

```
$ sudo /etc/init.d/apache2 restart
$ firefox localhost
```

---

## Using a matrix to transform a point cloud

---

In this tutorial we will learn how to transform a point cloud using a 4x4 matrix. We will apply a rotation and a translation to a loaded point cloud and display then result.

This program is able to load one PCD or PLY file; apply a matrix transformation on it and display the original and transformed point cloud.

### Contents

- *Using a matrix to transform a point cloud*
  - *The code*
  - *The explanation*
  - *Compiling and running the program*
  - *More about transformations*

## 12.1 The code

First, create a file, let's say, `matrix_transform.cpp` in your favorite editor, and place the following code inside it:

## 12.2 The explanation

Now, let's break down the code piece by piece.

We include all the headers we will make use of. `#include <pcl/common/transforms.h>` allows us to use `pcl::transformPointCloud` function.

This function display the help in case the user didn't provide expected arguments.

We parse the arguments on the command line, either using **-h** or **-help** will display the help. This terminates the program

We look for .ply or .pcd filenames in the arguments. If not found; terminate the program. The bool **file\_is\_pcd** will help us choose between loading PCD or PLY file.

We now load the PCD/PLY file and check if the file was loaded successfully. Otherwise terminate the program.

This is a first approach to create a transformation. This will help you understand how transformation matrices work. We initialize a 4x4 matrix to identity;

```
i = | 1 0 0 0 |  
    | 0 1 0 0 |  
    | 0 0 1 0 |  
    | 0 0 0 1 |
```

---

**Note:** The identity matrix is the equivalent of “1” when multiplying numbers; it changes nothing. It is a square matrix with ones on the main diagonal and zeros elsewhere.

---

This means no transformation (no rotation and no translation). We do not use the last row of the matrix.

The first 3 rows and columns (top left) components are the rotation matrix. The first 3 rows of the last column is the translation.

Here we defined a 45° (PI/4) rotation around the Z axis and a translation on the X axis. This is the transformation we just defined

```
R = | cos(θ) -sin(θ) 0.0 |  
    | sin(θ)  cos(θ) 0.0 |  
    | 0.0      0.0  1.0 |  
  
t = < 2.5, 0.0, 0.0 >
```

This second approach is easier to understand and is less error prone. Be careful if you want to apply several rotations; rotations are not commutative ! This means than in most cases:  $\text{rotA} * \text{rotB} \neq \text{rotB} * \text{rotA}$ .

Now we apply this matrix on the point cloud **source\_cloud** and we save the result in the newly created **transformed\_cloud**.

We then visualize the result using the **PCLVisualizer**. The original point cloud will be displayed white and the transformed one in red. The coordinates axis will be displayed. We also set the background color of the visualizer and the point display size.

## 12.3 Compiling and running the program

Add the following lines to your CMakeLists.txt file:

After you have made the executable, run it passing a path to a PCD or PLY file. To reproduce the results shown below, you can download the [cube.ply](#) file:

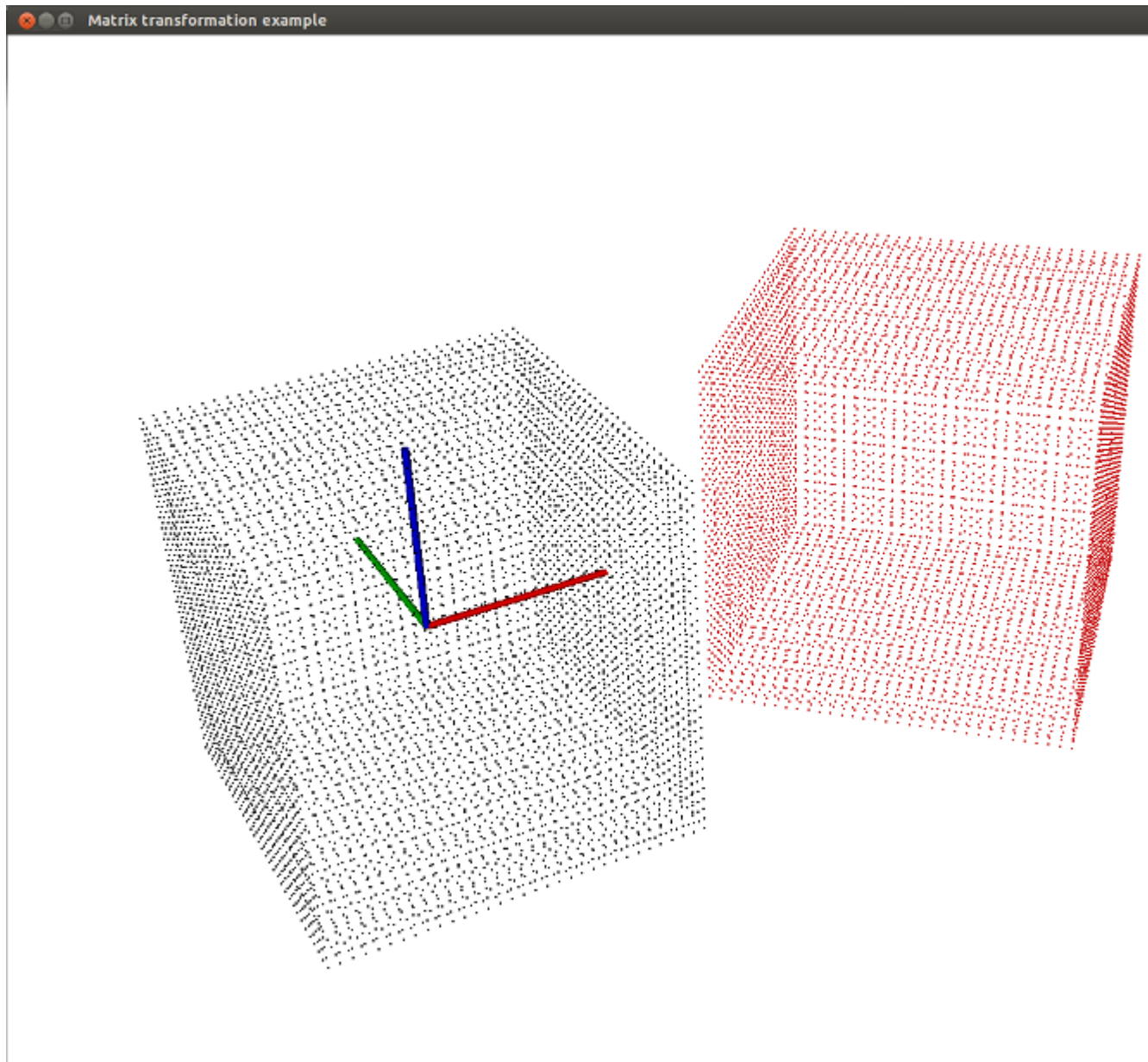
```
$ ./matrix_transform cube.ply
```

You will see something similar to this:

```
./matrix_transform cube.ply
[pcl::PLYReader] /home/victor/cube.ply:12: property 'list uint8 uint32 vertex_indices
↪' of element 'face' is not handled
Method #1: using a Matrix4f
0.707107 -0.707107 0 2.5
0.707107 0.707107 0 0
0 0 1 0
0 0 0 1

Method #2: using an Affine3f
0.707107 -0.707107 0 2.5
0.707107 0.707107 0 0
0 0 1 0
0 0 0 1

Point cloud colors : white = original point cloud
                    red   = transformed point cloud
```



## 12.4 More about transformations

So now you successfully transformed a point cloud using a transformation matrix.  
What if you want to transform a single point ? A vector ?

A point is defined in 3D space with its three coordinates;  $x, y, z$  (in a cartesian coordinate system).  
How can you multiply a vector (with 3 coordinates) with a  $4 \times 4$  matrix ? You simply can't ! If you don't know why please refer to [matrix multiplications on wikipedia](#).

We need a vector with 4 components. What do you put in the last component ? It depends on what you want to do:

- If you want to transform a point: put 1 at the end of the vector so that the translation is taken in account.
- If you want to transform the direction of a vector: put 0 at the end of the vector to ignore the translation.

Here's a quick example, we want to transform the following vector:

```
[10, 5, 0, 3, 0, -1]
```

Where the first 3 components defines the origin coordinates and the last 3 components the direction.

This vector starts at point 10, 5, 0 and ends at 13, 5, -1.

This is what you need to do to transform the vector:

```
[10, 5, 0, 1] * 4x4_transformation_matrix  
[3, 0, -1, 0] * 4x4_transformation_matrix
```



---

## Adding your own custom *PointT* type

---

The current document explains not only how to add your own *PointT* point type, but also what templated point types are in PCL, why do they exist, and how are they exposed. If you're already familiar with this information, feel free to skip to the last part of the document.

### Contents

- *Adding your own custom PointT type*
  - *Why PointT types*
  - *What PointT types are available in PCL?*
  - *How are the point types exposed?*
  - *How to add a new PointT type*
  - *Example*

---

**Note:** The current document is valid only for PCL 0.x and 1.x. Note that at the time of this writing we are expecting things to be changed in PCL 2.x.

---

PCL comes with a variety of pre-defined point types, ranging from SSE-aligned structures for XYZ data, to more complex n-dimensional histogram representations such as PFH (Point Feature Histograms). These types should be enough to support all the algorithms and methods implemented in PCL. However, there are cases where users would like to define new types. This document describes the steps involved in defining your own custom *PointT* type and making sure that your project can be compiled successfully and ran.

## 13.1 Why *PointT* types

PCL's *PointT* legacy goes back to the days where it was a library developed within ROS. The consensus then was that a *Point Cloud* is a complicated *n-D* structure that needs to be able to represent different types of information. However,

the user should know and understand what types of information need to be passed around, in order to make the code easier to debug, think about optimizations, etc.

One example is represented by simple operations on XYZ data. The most efficient way for SSE-enabled processors, is to store the 3 dimensions as floats, followed by an extra float for padding:

```
1 struct PointXYZ
2 {
3     float x;
4     float y;
5     float z;
6     float padding;
7 };
```

As an example however, in case an user is looking at compiling PCL on an embedded platform, adding the extra padding can be a waste of memory. Therefore, a simpler *PointXYZ* structure without the last float could be used instead.

Moreover, if your application requires a *PointXYZRGBNormal* which contains XYZ 3D data, *RGBA* information (colors), and surface normals estimated at each point, it is trivial to define a structure with all the above. Since all algorithms in PCL should be templated, there are no other changes required other than your structure definition.

## 13.2 What *PointT* types are available in PCL?

To cover all possible cases that we could think of, we defined a plethora of point types in PCL. The following might be only a snippet, please see [point\\_types.hpp](#) for the complete list.

This list is important, because before defining your own custom type, you need to understand why the existing types were created they way they were. In addition, the type that you want, might already be defined for you.

- *PointXYZ* - Members: float x, y, z;

This is one of the most used data types, as it represents 3D xyz information only. The 3 floats are padded with an additional float for SSE alignment. The user can either access *points[i].data[0]* or *points[i].x* for accessing say, the x coordinate.

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
```

- *PointXYZI* - Members: float x, y, z, intensity;

Simple XYZ + intensity point type. In an ideal world, these 4 components would create a single structure, SSE-aligned. However, because the majority of point operations will either set the last component of the *data[4]* array (from the xyz union) to 0 or 1 (for transformations), we cannot make *intensity* a member of the same structure, as its contents will be overwritten. For example, a dot product between two points will set their 4th component to 0, otherwise the dot product doesn't make sense, etc.

Therefore for SSE-alignment, we pad intensity with 3 extra floats. Inefficient in terms of storage, but good in terms of memory alignment.

```

union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    struct
    {
        float intensity;
    };
    float data_c[4];
};

```

- *PointXYZRGBA* - Members: float x, y, z; std::uint32\_t rgba;

Similar to *PointXYZI*, except *rgba* contains the RGBA information packed into an unsigned 32-bit integer. Thanks to the *union* declaration, it is also possible to access color channels individually by name.

---

**Note:** The nested *union* declaration provides yet another way to look at the RGBA data—as a single precision floating point number. This is present for historical reasons and should not be used in new code.

---

```

union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    union
    {
        struct
        {
            std::uint8_t b;
            std::uint8_t g;
            std::uint8_t r;
            std::uint8_t a;
        };
        float rgb;
    };
    std::uint32_t rgba;
};

```

- *PointXYZRGB* - float x, y, z; std::uint32\_t rgba;

Same as *PointXYZRGBA*.

- *PointXY* - float x, y;

Simple 2D x-y point structure.

```
struct
{
    float x;
    float y;
};
```

- *InterestPoint* - float x, y, z, strength;

Similar to *PointXYZI*, except *strength* contains a measure of the strength of the keypoint.

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    struct
    {
        float strength;
    };
    float data_c[4];
};
```

- *Normal* - float normal[3], curvature;

One of the other most used data types, the *Normal* structure represents the surface normal at a given point, and a measure of curvature (which is obtained in the same call as a relationship between the eigenvalues of a surface patch – see the *NormalEstimation* class API for more information).

Because operation on surface normals are quite common in PCL, we pad the 3 components with a fourth one, in order to be SSE-aligned and computationally efficient. The user can either access *points[i].data\_n[0]* or *points[i].normal[0]* or *points[i].normal\_x* for accessing say, the first coordinate of the normal vector. Again, *curvature* cannot be stored in the same structure as it would be overwritten by operations on the normal data.

```
union
{
    float data_n[4];
    float normal[3];
    struct
    {
        float normal_x;
        float normal_y;
        float normal_z;
    };
};
union
{
    struct
    {
```

(continues on next page)

(continued from previous page)

```

    float curvature;
};
float data_c[4];
};

```

- *PointNormal* - float x, y, z; float normal[3], curvature;

A point structure that holds XYZ data, together with surface normals and curvatures.

```

union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    float data_n[4];
    float normal[3];
    struct
    {
        float normal_x;
        float normal_y;
        float normal_z;
    };
};
union
{
    struct
    {
        float curvature;
    };
    float data_c[4];
};

```

- *PointXYZRGBNormal* - float x, y, z, normal[3], curvature; std::uint32\_t rgba;

A point structure that holds XYZ data, and RGBA colors, together with surface normals and curvatures.

---

**Note:** Despite the name, this point type does contain the alpha color channel.

---

```

union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};

```

(continues on next page)

(continued from previous page)

```

union
{
    float data_n[4];
    float normal[3];
    struct
    {
        float normal_x;
        float normal_y;
        float normal_z;
    };
}
union
{
    struct
    {
        union
        {
            union
            {
                struct
                {
                    std::uint8_t b;
                    std::uint8_t g;
                    std::uint8_t r;
                    std::uint8_t a;
                };
                float rgb;
            };
            std::uint32_t rgba;
        };
        float curvature;
    };
    float data_c[4];
};

```

- *PointXYZINormal* - float x, y, z, intensity, normal[3], curvature;

A point structure that holds XYZ data, and intensity values, together with surface normals and curvatures.

```

union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    float data_n[4];
    float normal[3];
    struct
    {
        float normal_x;
        float normal_y;

```

(continues on next page)

(continued from previous page)

```

    float normal_z;
};
}
union
{
    struct
    {
        float intensity;
        float curvature;
    };
    float data_c[4];
};

```

- *PointWithRange* - float x, y, z (union with float point[4]), range;

Similar to *PointXYZI*, except *range* contains a measure of the distance from the acquisition viewpoint to the point in the world.

```

union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    struct
    {
        float range;
    };
    float data_c[4];
};

```

- *PointWithViewpoint* - float x, y, z, vp\_x, vp\_y, vp\_z;

Similar to *PointXYZI*, except *vp\_x*, *vp\_y*, and *vp\_z* contain the acquisition viewpoint as a 3D point.

```

union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    struct
    {
        float vp_x;
        float vp_y;

```

(continues on next page)

(continued from previous page)

```
float vp_z;
};
float data_c[4];
};
```

- *MomentInvariants* - float j1, j2, j3;

Simple point type holding the 3 moment invariants at a surface patch. See *MomentInvariantsEstimation* for more information.

```
struct
{
    float j1, j2, j3;
};
```

- *PrincipalRadiiRSD* - float r\_min, r\_max;

Simple point type holding the 2 RSD radii at a surface patch. See *RSDEstimation* for more information.

```
struct
{
    float r_min, r_max;
};
```

- *Boundary* - std::uint8\_t boundary\_point;

Simple point type holding whether the point is lying on a surface boundary or not. See *BoundaryEstimation* for more information.

```
struct
{
    std::uint8_t boundary_point;
};
```

- *PrincipalCurvatures* - float principal\_curvature[3], pc1, pc2;

Simple point type holding the principal curvatures of a given point. See *PrincipalCurvaturesEstimation* for more information.

```
struct
{
    union
    {
        float principal_curvature[3];
        struct
        {
            float principal_curvature_x;
            float principal_curvature_y;
            float principal_curvature_z;
        };
    };
    float pc1;
    float pc2;
};
```

- *PFHSignature125* - float pfh[125];

Simple point type holding the PFH (Point Feature Histogram) of a given point. See *PFHEstimation* for more information.

```
struct
{
    float histogram[125];
};
```

- *FPFHSiganture33* - float fpfh[33];

Simple point type holding the FPFH (Fast Point Feature Histogram) of a given point. See *FPFHEstimation* for more information.

```
struct
{
    float histogram[33];
};
```

- *VFHSiganture308* - float vfh[308];

Simple point type holding the VFH (Viewpoint Feature Histogram) of a given point. See *VFHEstimation* for more information.

```
struct
{
    float histogram[308];
};
```

- *Narf36* - float x, y, z, roll, pitch, yaw; float descriptor[36];

Simple point type holding the NARF (Normally Aligned Radius Feature) of a given point. See *NARFEstimation* for more information.

```
struct
{
    float x, y, z, roll, pitch, yaw;
    float descriptor[36];
};
```

- *BorderDescription* - int x, y; BorderTraits traits;

Simple point type holding the border type of a given point. See *BorderEstimation* for more information.

```
struct
{
    int x, y;
    BorderTraits traits;
};
```

- *IntensityGradient* - float gradient[3];

Simple point type holding the intensity gradient of a given point. See *IntensityGradientEstimation* for more information.

```
struct
{
    union
    {
        float gradient[3];
        struct
        {
            float gradient_x;
```

(continues on next page)

(continued from previous page)

```
    float gradient_y;  
    float gradient_z;  
};  
};  
};
```

- *Histogram* - float histogram[N];

General purpose n-D histogram placeholder.

```
template <int N>  
struct Histogram  
{  
    float histogram[N];  
};
```

- *PointWithScale* - float x, y, z, scale;

Similar to *PointXYZI*, except *scale* contains the scale at which a certain point was considered for a geometric operation (e.g. the radius of the sphere for its nearest neighbors computation, the window size, etc).

```
struct  
{  
    union  
    {  
        float data[4];  
        struct  
        {  
            float x;  
            float y;  
            float z;  
        };  
    };  
    float scale;  
};
```

- *PointSurfel* - float x, y, z, normal[3], rgba, radius, confidence, curvature;

A complex point type containing XYZ data, surface normals, together with RGB information, scale, confidence, and surface curvature.

```
union  
{  
    float data[4];  
    struct  
    {  
        float x;  
        float y;  
        float z;  
    };  
};  
union  
{  
    float data_n[4];  
    float normal[3];  
    struct  
    {
```

(continues on next page)

(continued from previous page)

```

    float normal_x;
    float normal_y;
    float normal_z;
};
};
union
{
    struct
    {
        std::uint32_t rgba;
        float radius;
        float confidence;
        float curvature;
    };
    float data_c[4];
};
};

```

### 13.3 How are the point types exposed?

Because of its large size, and because it's a template library, including many PCL algorithms in one source file can slow down the compilation process. At the time of writing this document, most C++ compilers still haven't been properly optimized to deal with large sets of templated files, especially when optimizations (*-O2* or *-O3*) are involved.

To speed up user code that includes and links against PCL, we are using *explicit template instantiations*, by including all possible combinations in which all algorithms could be called using the already defined point types from PCL. This means that once PCL is compiled as a library, any user code will not require to compile templated code, thus speeding up compile time. The trick involves separating the templated implementations from the headers which forward declare our classes and methods, and resolving at link time. Here's a fictitious example:

```

1 // foo.h
2
3 #ifndef PCL_FOO_
4 #define PCL_FOO_
5
6 template <typename PointT>
7 class Foo
8 {
9     public:
10         void
11         compute (const pcl::PointCloud<PointT> &input,
12                 pcl::PointCloud<PointT> &output);
13 }
14
15 #endif // PCL_FOO_

```

The above defines the header file which is usually included by all user code. As we can see, we're defining methods and classes, but we're not implementing anything yet.

```

1 // impl/foo.hpp
2
3 #ifndef PCL_IMPL_FOO_
4 #define PCL_IMPL_FOO_
5
6 #include "foo.h"

```

(continues on next page)

(continued from previous page)

```

7
8 template <typename PointT> void
9 Foo::compute (const pcl::PointCloud<PointT> &input,
10              pcl::PointCloud<PointT> &output)
11 {
12     output = input;
13 }
14
15 #endif // PCL_IMPL_FOO_

```

The above defines the actual template implementation of the method *Foo::compute*. This should normally be hidden from user code.

```

1 // foo.cpp
2
3 #include "pcl/point_types.h"
4 #include "pcl/impl/instantiate.hpp"
5 #include "foo.h"
6 #include "impl/foo.hpp"
7
8 // Instantiations of specific point types
9 PCL_INSTANTIATE(Foo, PCL_XYZ_POINT_TYPES);

```

And finally, the above shows the way the explicit instantiations are done in PCL. The macro *PCL\_INSTANTIATE* does nothing else but go over a given list of types and creates an explicit instantiation for each. From *pcl/include/pcl/impl/instantiate.hpp*:

```

// PCL_INSTANTIATE: call to instantiate template TEMPLATE for all
// POINT_TYPES

#define PCL_INSTANTIATE_IMPL(r, TEMPLATE, POINT_TYPE) \
    BOOST_PP_CAT(PCL_INSTANTIATE_, TEMPLATE)(POINT_TYPE)

#define PCL_INSTANTIATE(TEMPLATE, POINT_TYPES) \
    BOOST_PP_SEQ_FOR_EACH(PCL_INSTANTIATE_IMPL, TEMPLATE, POINT_TYPES);

```

Where *PCL\_XYZ\_POINT\_TYPES* is (from *pcl/include/pcl/impl/point\_types.hpp*):

```

// Define all point types that include XYZ data
#define PCL_XYZ_POINT_TYPES \
    (pcl::PointXYZ) \
    (pcl::PointXYZI) \
    (pcl::PointXYZRGBA) \
    (pcl::PointXYZRGB) \
    (pcl::InterestPoint) \
    (pcl::PointNormal) \
    (pcl::PointXYZRGBNormal) \
    (pcl::PointXYZINormal) \
    (pcl::PointWithRange) \
    (pcl::PointWithViewpoint) \
    (pcl::PointWithScale)

```

Basically, if you only want to explicitly instantiate *Foo* for *pcl::PointXYZ*, you don't need to use the macro, as something as simple as the following would do:

```

1 // foo.cpp
2
3 #include "pcl/point_types.h"
4 #include "pcl/impl/instantiate.hpp"
5 #include "foo.h"
6 #include "impl/foo.hpp"
7
8 template class Foo<pcl::PointXYZ>;

```

**Note:** For more information about explicit instantiations, please see *C++ Templates - The Complete Guide*, by David Vandervoorde and Nicolai M. Josuttis.

## 13.4 How to add a new *PointT* type

To add a new point type, you first have to define it. For example:

```

1 struct MyPointType
2 {
3     float test;
4 };

```

Then, you need to make sure your code includes the template header implementation of the specific class/algorithm in PCL that you want your new point type *MyPointType* to work with. For example, say you want to use *pcl::PassThrough*. All you would have to do is:

```

#define PCL_NO_PRECOMPILE
#include <pcl/filters/passthrough.h>
#include <pcl/filters/impl/passthrough.hpp>

// the rest of the code goes here

```

If your code is part of the library, which gets used by others, it might also make sense to try to use explicit instantiations for your *MyPointType* types, for any classes that you expose (from PCL or outside PCL).

**Note:** Starting with PCL-1.7 you need to define `PCL_NO_PRECOMPILE` before you include any PCL headers to include the templated algorithms as well.

## 13.5 Example

The following code snippet example creates a new point type that contains XYZ data (SSE padded), together with a test float.

```

1 #define PCL_NO_PRECOMPILE
2 #include <pcl/pcl_macros.h>
3 #include <pcl/point_types.h>
4 #include <pcl/point_cloud.h>
5 #include <pcl/io/pcd_io.h>
6

```

(continues on next page)

(continued from previous page)

```
7 struct MyPointType
8 {
9     PCL_ADD_POINT4D;           // preferred way of adding a XYZ+padding
10    float test;
11    PCL_MAKE_ALIGNED_OPERATOR_NEW // make sure our new allocators are aligned
12 } EIGEN_ALIGN16;               // enforce SSE padding for correct memory_
    ↪alignment
13
14 POINT_CLOUD_REGISTER_POINT_STRUCT (MyPointType,           // here we assume a XYZ +
    ↪"test" (as fields)
15                                     (float, x, x)
16                                     (float, y, y)
17                                     (float, z, z)
18                                     (float, test, test)
19 )
20
21
22 int
23 main (int argc, char** argv)
24 {
25     pcl::PointCloud<MyPointType> cloud;
26     cloud.points.resize (2);
27     cloud.width = 2;
28     cloud.height = 1;
29
30     cloud.points[0].test = 1;
31     cloud.points[1].test = 2;
32     cloud.points[0].x = cloud.points[0].y = cloud.points[0].z = 0;
33     cloud.points[1].x = cloud.points[1].y = cloud.points[1].z = 3;
34
35     pcl::io::savePCDFile ("test.pcd", cloud);
36 }
```

---

## Writing a new PCL class

---

Converting code to a PCL-like mentality/syntax for someone that comes in contact for the first time with our infrastructure might appear difficult, or raise certain questions.

This short guide is to serve as both a HowTo and a FAQ for writing new PCL classes, either from scratch, or by adapting old code.

Besides converting your code, this guide also explains some of the advantages of contributing your code to an already existing open source project. Here, we advocate for PCL, but you can certainly apply the same ideology to other similar projects.

### Contents

- *Writing a new PCL class*
  - *Advantages: Why contribute?*
  - *Example: a bilateral filter*
  - *Setting up the structure*
    - \* *bilateral.h*
    - \* *bilateral.hpp*
    - \* *bilateral.cpp*
    - \* *CMakeLists.txt*
  - *Filling in the class structure*
    - \* *bilateral.cpp*
    - \* *bilateral.h*
    - \* *bilateral.hpp*
  - *Taking advantage of other PCL concepts*

- \* *Point indices*
- \* *Licenses*
- \* *Proper naming*
- \* *Code comments*
- *Testing the new class*

## 14.1 Advantages: Why contribute?

The first question that someone might ask and we would like to answer is:

*Why contribute to PCL, as in what are its advantages?*

This question assumes you’ve already identified that the set of tools and libraries that PCL has to offer are useful for your project, so you have already become an *user*.

Because open source projects are mostly voluntary efforts, usually with developers geographically distributed around the world, it’s very common that the development process has a certain *incremental*, and *iterative* flavor. This means that:

- it’s impossible for developers to think ahead of all the possible uses a new piece of code they write might have, but also...
- figuring out solutions for corner cases and applications where bugs might occur is hard, and might not be desirable to tackle at the beginning, due to limited resources (mostly a cost function of free time).

In both cases, everyone has definitely encountered situations where either an algorithm/method that they need is missing, or an existing one is buggy. Therefore the next natural step is obvious:

*change the existing code to fit your application/problem.*

While we’re going to discuss how to do that in the next sections, we would still like to provide an answer for the first question that we raised, namely “why contribute?”.

In our opinion, there are many advantages. To quote Eric Raymond’s *Linus’s Law*: “**given enough eyeballs, all bugs are shallow**”. What this means is that by opening your code to the world, and allowing others to see it, the chances of it getting fixed and optimized are higher, especially in the presence of a dynamic community such as the one that PCL has.

In addition to the above, your contribution might enable, amongst many things:

- others to create new work based on your code;
- you to learn about new uses (e.g., thinks that you haven’t thought it could be used when you designed it);
- worry-free maintainership (e.g., you can go away for some time, and then return and see your code still working. Others will take care of adapting it to the newest platforms, newest compilers, etc);
- your reputation in the community to grow - everyone likes free stuff (!).

For most of us, all of the above apply. For others, only some (your mileage might vary).

## 14.2 Example: a bilateral filter

To illustrate the code conversion process, we selected the following example: apply a bilateral filter over intensity data from a given input point cloud, and save the results to disk.

```

1  #include <pcl/point_types.h>
2  #include <pcl/io/pcd_io.h>
3  #include <pcl/kdtree/kdtree_flann.h>
4
5  typedef pcl::PointXYZI PointT;
6
7  float
8  G (float x, float sigma)
9  {
10     return std::exp (- (x*x)/(2*sigma*sigma));
11 }
12
13 int
14 main (int argc, char *argv[])
15 {
16     std::string incloudfile = argv[1];
17     std::string outcloudfile = argv[2];
18     float sigma_s = atof (argv[3]);
19     float sigma_r = atof (argv[4]);
20
21     // Load cloud
22     pcl::PointCloud<PointT>::Ptr cloud (new pcl::PointCloud<PointT>);
23     pcl::io::loadPCDFile (incloudfile.c_str (), *cloud);
24     int pnumber = (int)cloud->size ();
25
26     // Output Cloud = Input Cloud
27     pcl::PointCloud<PointT> outcloud = *cloud;
28
29     // Set up KDTree
30     pcl::KdTreeFLANN<PointT>::Ptr tree (new pcl::KdTreeFLANN<PointT>);
31     tree->setInputCloud (cloud);
32
33     // Neighbors containers
34     std::vector<int> k_indices;
35     std::vector<float> k_distances;
36
37     // Main Loop
38     for (int point_id = 0; point_id < pnumber; ++point_id)
39     {
40         float BF = 0;
41         float W = 0;
42
43         tree->radiusSearch (point_id, 2 * sigma_s, k_indices, k_distances);
44
45         // For each neighbor
46         for (std::size_t n_id = 0; n_id < k_indices.size (); ++n_id)
47         {
48             float id = k_indices.at (n_id);
49             float dist = sqrt (k_distances.at (n_id));
50             float intensity_dist = std::abs (cloud->points[point_id].intensity - cloud->
↳ points[id].intensity);
51
52             float w_a = G (dist, sigma_s);
53             float w_b = G (intensity_dist, sigma_r);
54             float weight = w_a * w_b;
55
56             BF += weight * cloud->points[id].intensity;

```

(continues on next page)

(continued from previous page)

```

57     W += weight;
58 }
59
60 outcloud.points[point_id].intensity = BF / W;
61 }
62
63 // Save filtered output
64 pcl::io::savePCDFile (outcloudfile.c_str (), outcloud);
65 return (0);
66 }

```

The presented code snippet contains:

- an I/O component: lines 21-27 (reading data from disk), and 64 (writing data to disk)
- an initialization component: lines 29-35 (setting up a search method for nearest neighbors using a KdTree)
- the actual algorithmic component: lines 7-11 and 37-61

Our goal here is to convert the algorithm given into an useful PCL class so that it can be reused elsewhere.

## 14.3 Setting up the structure

**Note:** If you're not familiar with the PCL file structure already, please go ahead and read the [PCL C++ Programming Style Guide](#) to familiarize yourself with the concepts.

There're two different ways we could set up the structure: i) set up the code separately, as a standalone PCL class, but outside of the PCL code tree; or ii) set up the files directly in the PCL code tree. Since our assumption is that the end result will be contributed back to PCL, it's best to concentrate on the latter, also because it is a bit more complex (i.e., it involves a few additional steps). You can obviously repeat these steps with the former case as well, with the exception that you don't need the files copied in the PCL tree, nor you need the fancier *cmake* logic.

Assuming that we want the new algorithm to be part of the PCL Filtering library, we will begin by creating 3 different files under filters:

- *include/pcl/filters/bilateral.h* - will contain all definitions;
- *include/pcl/filters/impl/bilateral.hpp* - will contain the templated implementations;
- *src/bilateral.cpp* - will contain the explicit template instantiations<sup>0</sup>.

We also need a name for our new class. Let's call it *BilateralFilter*.

### 14.3.1 bilateral.h

As previously mentioned, the *bilateral.h* header file will contain all the definitions pertinent to the *BilateralFilter* class. Here's a minimal skeleton:

```

1  #ifndef PCL_FILTERS_BILATERAL_H_
2  #define PCL_FILTERS_BILATERAL_H_
3

```

(continues on next page)

<sup>0</sup> Some PCL filter algorithms provide two implementations: one for `PointCloud<T>` types and another one operating on legacy `PCLPointCloud2` types. This is no longer required.

(continued from previous page)

```

4  #include <pcl/filters/filter.h>
5
6  namespace pcl
7  {
8      template<typename PointT>
9      class BilateralFilter : public Filter<PointT>
10     {
11     };
12 }
13
14 #endif // PCL_FILTERS_BILATERAL_H_

```

### 14.3.2 bilateral.hpp

While we're at it, let's set up two skeleton *bilateral.hpp* and *bilateral.cpp* files as well. First, *bilateral.hpp*:

```

1  #ifndef PCL_FILTERS_BILATERAL_IMPL_H_
2  #define PCL_FILTERS_BILATERAL_IMPL_H_
3
4  #include <pcl/filters/bilateral.h>
5
6  #endif // PCL_FILTERS_BILATERAL_IMPL_H_

```

This should be straightforward. We haven't declared any methods for *BilateralFilter* yet, therefore there is no implementation.

### 14.3.3 bilateral.cpp

Let's write *bilateral.cpp* too:

```

1  #include <pcl/filters/bilateral.h>
2  #include <pcl/filters/impl/bilateral.hpp>

```

Because we are writing templated code in PCL (1.x) where the template parameter is a point type (see [Adding your own custom PointT type](#)), we want to explicitly instantiate the most common use cases in *bilateral.cpp*, so that users don't have to spend extra cycles when compiling code that uses our *BilateralFilter*. To do this, we need to access both the header (*bilateral.h*) and the implementations (*bilateral.hpp*).

### 14.3.4 CMakeLists.txt

Let's add all the files to the PCL Filtering *CMakeLists.txt* file, so we can enable the build.

```

1  # Find "set (srcs", and add a new entry there, e.g.,
2  set (srcs
3      src/conditional_removal.cpp
4      # ...
5      src/bilateral.cpp)
6
7
8  # Find "set (incs", and add a new entry there, e.g.,
9  set (incs
10     include pcl/${SUBSYS_NAME}/conditional_removal.h

```

(continues on next page)

(continued from previous page)

```

11     # ...
12     include pcl/${SUBSYS_NAME}/bilateral.h
13 )
14
15 # Find "set (impl_incs", and add a new entry there, e.g.,
16 set (impl_incs
17     include/pcl/${SUBSYS_NAME}/impl/conditional_removal.hpp
18     # ...
19     include/pcl/${SUBSYS_NAME}/impl/bilateral.hpp
20 )

```

## 14.4 Filling in the class structure

If you correctly edited all the files above, recompiling PCL using the new filter classes in place should work without problems. In this section, we'll begin filling in the actual code in each file. Let's start with the *bilateral.cpp* file, as its content is the shortest.

### 14.4.1 bilateral.cpp

As previously mentioned, we're going to explicitly instantiate and *precompile* a number of templated specializations for the *BilateralFilter* class. While this might lead to an increased compilation time for the PCL Filtering library, it will save users the pain of processing and compiling the templates on their end, when they use the class in code they write. The simplest possible way to do this would be to declare each instance that we want to precompile by hand in the *bilateral.cpp* file as follows:

```

1  #include <pcl/point_types.h>
2  #include <pcl/filters/bilateral.h>
3  #include <pcl/filters/impl/bilateral.hpp>
4
5  template class PCL_EXPORTS pcl::BilateralFilter<pcl::PointXYZ>;
6  template class PCL_EXPORTS pcl::BilateralFilter<pcl::PointXYZI>;
7  template class PCL_EXPORTS pcl::BilateralFilter<pcl::PointXYZRGB>;
8  // ...

```

However, this becomes cumbersome really fast, as the number of point types PCL supports grows. Maintaining this list up to date in multiple files in PCL is also painful. Therefore, we are going to use a special macro called *PCL\_INSTANTIATE* and change the above code as follows:

```

1  #include <pcl/point_types.h>
2  #include <pcl/impl/instantiate.hpp>
3  #include <pcl/filters/bilateral.h>
4  #include <pcl/filters/impl/bilateral.hpp>
5
6  PCL_INSTANTIATE(BilateralFilter, PCL_XYZ_POINT_TYPES);

```

This example, will instantiate a *BilateralFilter* for all XYZ point types defined in the *point\_types.h* file (see **`pcl::PCL_XYZ_POINT_TYPES<PCL_XYZ_POINT_TYPES>`** for more information).

By looking closer at the code presented in *Example: a bilateral filter*, we notice constructs such as *cloud->points[point\_id].intensity*. This indicates that our filter expects the presence of an **intensity** field in the point type. Because of this, using **PCL\_XYZ\_POINT\_TYPES** won't work, as not all the types defined there have intensity data present. In fact, it's easy to notice that only two of the types contain intensity, namely:

**:pcl::PointXYZI<pcl::PointXYZI>** and **:pcl::PointXYZINormal<pcl::PointXYZINormal>**. We therefore replace **PCL\_XYZ\_POINT\_TYPES** and the final *bilateral.cpp* file becomes:

```

1  #include <pcl/point_types.h>
2  #include <pcl/impl/instantiate.hpp>
3  #include <pcl/filters/bilateral.h>
4  #include <pcl/filters/impl/bilateral.hpp>
5
6  PCL_INSTANTIATE(BilateralFilter, (pcl::PointXYZI) (pcl::PointXYZINormal));

```

Note that at this point we haven't declared the **PCL\_INSTANTIATE** template for *BilateralFilter*, nor did we actually implement the pure virtual functions in the abstract class **:pcl::Filter<pcl::Filter>** so attempting to compile the code will result in errors like:

```

filters/src/bilateral.cpp:6:32: error: expected constructor, destructor, or type_
↪conversion before '(' token

```

## 14.4.2 bilateral.h

We begin filling the *BilateralFilter* class by first declaring the constructor, and its member variables. Because the bilateral filtering algorithm has two parameters, we will store these as class members, and implement setters and getters for them, to be compatible with the PCL 1.x API paradigms.

```

1  ...
2  namespace pcl
3  {
4      template<typename PointT>
5      class BilateralFilter : public Filter<PointT>
6      {
7      public:
8          BilateralFilter () : sigma_s_ (0),
9                               sigma_r_ (std::numeric_limits<double>::max ())
10         {
11         }
12
13         void
14         setSigmaS (const double sigma_s)
15         {
16             sigma_s_ = sigma_s;
17         }
18
19         double
20         getSigmaS () const
21         {
22             return (sigma_s_);
23         }
24
25         void
26         setSigmaR (const double sigma_r)
27         {
28             sigma_r_ = sigma_r;
29         }
30
31         double
32         getSigmaR () const
33         {

```

(continues on next page)

(continued from previous page)

```

34     return (sigma_r_);
35 }
36
37 private:
38     double sigma_s_;
39     double sigma_r_;
40 };
41 }
42
43 #endif // PCL_FILTERS_BILATERAL_H_

```

Nothing out of the ordinary so far, except maybe lines 8-9, where we gave some default values to the two parameters. Because our class inherits from `pcl::Filter<pcl::Filter>`, and that inherits from `pcl::PCLBase<pcl::PCLBase>`, we can make use of the `pcl::setInputCloud<pcl::PCLBase::setInputCloud>` method to pass the input data to our algorithm (stored as `pcl::input_<pcl::PCLBase::input_>`). We therefore add an *using* declaration as follows:

```

1  ...
2  template<typename PointT>
3  class BilateralFilter : public Filter<PointT>
4  {
5      using Filter<PointT>::input_;
6      public:
7          BilateralFilter () : sigma_s_ (0),
8  ...

```

This will make sure that our class has access to the member variable *input\_* without typing the entire construct. Next, we observe that each class that inherits from `pcl::Filter<pcl::Filter>` must inherit a `pcl::applyFilter<pcl::Filter::applyFilter>` method. We therefore define:

```

1  ...
2      using Filter<PointT>::input_;
3      typedef typename Filter<PointT>::PointCloud PointCloud;
4
5      public:
6          BilateralFilter () : sigma_s_ (0),
7                               sigma_r_ (std::numeric_limits<double>::max ())
8          {
9          }
10
11         void
12         applyFilter (PointCloud &output);
13     ...

```

The implementation of *applyFilter* will be given in the *bilateral.hpp* file later. Line 3 constructs a typedef so that we can use the type *PointCloud* without typing the entire construct.

Looking at the original code from section [Example: a bilateral filter](#), we notice that the algorithm consists of applying the same operation to every point in the cloud. To keep the *applyFilter* call clean, we therefore define method called *computePointWeight* whose implementation will contain the corpus defined in between lines 45-58:

```

1  ...
2      void
3      applyFilter (PointCloud &output);
4
5      double

```

(continues on next page)

(continued from previous page)

```

6     computePointWeight (const int pid, const std::vector<int> &indices, const_
↪std::vector<float> &distances);
7     ...

```

In addition, we notice that lines 29-31 and 43 from section *Example: a bilateral filter* construct a `pcl::KdTree<pcl::KdTree>` structure for obtaining the nearest neighbors for a given point. We therefore add:

```

1  #include <pcl/kdtree/kdtree.h>
2  ...
3      using Filter<PointT>::input_;
4      typedef typename Filter<PointT>::PointCloud PointCloud;
5      typedef typename pcl::KdTree<PointT>::Ptr KdTreePtr;
6
7  public:
8  ...
9
10     void
11     setSearchMethod (const KdTreePtr &tree)
12     {
13         tree_ = tree;
14     }
15
16 private:
17 ...
18     KdTreePtr tree_;
19 ...

```

Finally, we would like to add the kernel method ( $G(\text{float } x, \text{float } \sigma)$ ) inline so that we speed up the computation of the filter. Because the method is only useful within the context of the algorithm, we will make it private. The header file becomes:

```

1  #ifndef PCL_FILTERS_BILATERAL_H_
2  #define PCL_FILTERS_BILATERAL_H_
3
4  #include <pcl/filters/filter.h>
5  #include <pcl/kdtree/kdtree.h>
6
7  namespace pcl
8  {
9      template<typename PointT>
10     class BilateralFilter : public Filter<PointT>
11     {
12     public:
13         using Filter<PointT>::input_;
14         typedef typename Filter<PointT>::PointCloud PointCloud;
15         typedef typename pcl::KdTree<PointT>::Ptr KdTreePtr;
16
17         BilateralFilter () : sigma_s_ (0),
18                             sigma_r_ (std::numeric_limits<double>::max ())
19         {
20         }
21
22         void
23         applyFilter (PointCloud &output);
24     };
25

```

(continues on next page)

(continued from previous page)

```

26     double
27     computePointWeight (const int pid, const std::vector<int> &indices, const_
↪std::vector<float> &distances);
28
29     void
30     setSigmaS (const double sigma_s)
31     {
32         sigma_s_ = sigma_s;
33     }
34
35     double
36     getSigmaS () const
37     {
38         return (sigma_s_);
39     }
40
41     void
42     setSigmaR (const double sigma_r)
43     {
44         sigma_r_ = sigma_r;
45     }
46
47     double
48     getSigmaR () const
49     {
50         return (sigma_r_);
51     }
52
53     void
54     setSearchMethod (const KdTreePtr &tree)
55     {
56         tree_ = tree;
57     }
58
59 private:
60
61     inline double
62     kernel (double x, double sigma)
63     {
64         return (std::exp (- (x*x)/(2*sigma*sigma)));
65     }
66
67     double sigma_s_;
68     double sigma_r_;
69     KdTreePtr tree_;
70
71 };
72
73
74 #endif // PCL_FILTERS_BILATERAL_H_

```

### 14.4.3 bilateral.hpp

There're two methods that we need to implement here, namely *applyFilter* and *computePointWeight*.

```

1  template <typename PointT> double
2  pcl::BilateralFilter<PointT>::computePointWeight (const int pid,
3                                                    const std::vector<int> &indices,
4                                                    const std::vector<float> &
5  ↪distances)
6  {
7      double BF = 0, W = 0;
8
9      // For each neighbor
10     for (std::size_t n_id = 0; n_id < indices.size (); ++n_id)
11     {
12         double id = indices[n_id];
13         double dist = std::sqrt (distances[n_id]);
14         double intensity_dist = std::abs (input_>points[pid].intensity - input_>
15 ↪points[id].intensity);
16
17         double weight = kernel (dist, sigma_s_) * kernel (intensity_dist, sigma_r_);
18
19         BF += weight * input_>points[id].intensity;
20         W += weight;
21     }
22     return (BF / W);
23 }
24
25 template <typename PointT> void
26 pcl::BilateralFilter<PointT>::applyFilter (PointCloud &output)
27 {
28     tree_>setInputCloud (input_);
29
30     std::vector<int> k_indices;
31     std::vector<float> k_distances;
32
33     output = *input_;
34
35     for (std::size_t point_id = 0; point_id < input_>points.size (); ++point_id)
36     {
37         tree_>radiusSearch (point_id, sigma_s_ * 2, k_indices, k_distances);
38
39         output.points[point_id].intensity = computePointWeight (point_id, k_indices, k_
40 ↪distances);
41     }
42 }

```

The *computePointWeight* method should be straightforward as it's *almost identical* to lines 45-58 from section [Example: a bilateral filter](#). We basically pass in a point index that we want to compute the intensity weight for, and a set of neighboring points with distances.

In *applyFilter*, we first set the input data in the tree, copy all the input data into the output, and then proceed at computing the new weighted point intensities.

Looking back at [Filling in the class structure](#), it's now time to declare the *PCL\_INSTANTIATE* entry for the class:

```

1  #ifndef PCL_FILTERS_BILATERAL_IMPL_H_
2  #define PCL_FILTERS_BILATERAL_IMPL_H_
3
4  #include <pcl/filters/bilateral.h>
5

```

(continues on next page)

(continued from previous page)

```

6  ...
7
8  #define PCL_INSTANTIATE_BilateralFilter(T) template class PCL_EXPORTS_
↪pcl::BilateralFilter<T>;
9
10 #endif // PCL_FILTERS_BILATERAL_IMPL_H_

```

One additional thing that we can do is error checking on:

- whether the two *sigma\_s\_* and *sigma\_r\_* parameters have been given;
- whether the search method object (i.e., *tree\_*) has been set.

For the former, we're going to check the value of *sigma\_s\_*, which was set to a default of 0, and has a critical importance for the behavior of the algorithm (it basically defines the size of the support region). Therefore, if at the execution of the code, its value is still 0, we will print an error using the `:pcl:'PCL_ERROR<PCL_ERROR>'` macro, and return.

In the case of the search method, we can either do the same, or be clever and provide a default option for the user. The best default options are:

- use an organized search method via `:pcl:'pcl::OrganizedNeighbor<pcl::OrganizedNeighbor>'` if the point cloud is organized;
- use a general purpose kdtree via `:pcl:'pcl::KdTreeFLANN<pcl::KdTreeFLANN>'` if the point cloud is unorganized.

```

1  #include <pcl/kdtree/kdtree_flann.h>
2  #include <pcl/kdtree/organized_data.h>
3
4  ...
5  template <typename PointT> void
6  pcl::BilateralFilter<PointT>::applyFilter (PointCloud &output)
7  {
8      if (sigma_s_ == 0)
9      {
10         PCL_ERROR ("[pcl::BilateralFilter::applyFilter] Need a sigma_s value given_
↪before continuing.\n");
11         return;
12     }
13     if (!tree_)
14     {
15         if (input_>isOrganized ())
16             tree_.reset (new pcl::OrganizedNeighbor<PointT> ());
17         else
18             tree_.reset (new pcl::KdTreeFLANN<PointT> (false));
19     }
20     tree_>setInputCloud (input_);
21     ...

```

The implementation file header thus becomes:

```

1  #ifndef PCL_FILTERS_BILATERAL_IMPL_H_
2  #define PCL_FILTERS_BILATERAL_IMPL_H_
3
4  #include <pcl/filters/bilateral.h>
5  #include <pcl/kdtree/kdtree_flann.h>
6  #include <pcl/kdtree/organized_data.h>
7

```

(continues on next page)

(continued from previous page)

```

8  template <typename PointT> double
9  pcl::BilateralFilter<PointT>::computePointWeight (const int pid,
10                                                     const std::vector<int> &indices,
11                                                     const std::vector<float> &
12  ↪distances)
13  {
14      double BF = 0, W = 0;
15
16      // For each neighbor
17      for (std::size_t n_id = 0; n_id < indices.size (); ++n_id)
18      {
19          double id = indices[n_id];
20          double dist = std::sqrt (distances[n_id]);
21          double intensity_dist = std::abs (input_>points[pid].intensity - input_>
22  ↪points[id].intensity);
23
24          double weight = kernel (dist, sigma_s_) * kernel (intensity_dist, sigma_r_);
25
26          BF += weight * input_>points[id].intensity;
27          W += weight;
28      }
29      return (BF / W);
30
31  template <typename PointT> void
32  pcl::BilateralFilter<PointT>::applyFilter (PointCloud &output)
33  {
34      if (sigma_s_ == 0)
35      {
36          PCL_ERROR ("[pcl::BilateralFilter::applyFilter] Need a sigma_s value given_
37  ↪before continuing.\n");
38          return;
39      }
40      if (!tree_)
41      {
42          if (input_>isOrganized ())
43              tree_.reset (new pcl::OrganizedNeighbor<PointT> ());
44          else
45              tree_.reset (new pcl::KdTreeFLANN<PointT> (false));
46      }
47      tree_>setInputCloud (input_);
48
49      std::vector<int> k_indices;
50      std::vector<float> k_distances;
51
52      output = *input_;
53
54      for (std::size_t point_id = 0; point_id < input_>points.size (); ++point_id)
55      {
56          tree_>radiusSearch (point_id, sigma_s_ * 2, k_indices, k_distances);
57
58          output.points[point_id].intensity = computePointWeight (point_id, k_indices, k_
59  ↪distances);
60      }
61
62  #define PCL_INSTANTIATE_BilateralFilter(T) template class PCL_EXPORTS_
63  ↪pcl::BilateralFilter<T>;

```

(continues on next page)

(continued from previous page)

```

61
62 #endif // PCL_FILTERS_BILATERAL_IMPL_H_

```

## 14.5 Taking advantage of other PCL concepts

### 14.5.1 Point indices

The standard way of passing point cloud data into PCL algorithms is via `pcl::setInputCloud<pcl::PCLBase::setInputCloud>` calls. In addition, PCL also defines a way to define a region of interest / *list of point indices* that the algorithm should operate on, rather than the entire cloud, via `pcl::setIndices<pcl::PCLBase::setIndices>`.

All classes inheriting from `pcl::PCLBase<pcl::PCLBase>` exhibit the following behavior: in case no set of indices is given by the user, a fake one is created once and used for the duration of the algorithm. This means that we could easily change the implementation code above to operate on a `<cloud, indices>` tuple, which has the added advantage that if the user does pass a set of indices, only those will be used, and if not, the entire cloud will be used.

The new *bilateral.hpp* class thus becomes:

```

1  #include <pcl/kdtree/kdtree_flann.h>
2  #include <pcl/kdtree/organized_data.h>
3
4  ...
5  template <typename PointT> void
6  pcl::BilateralFilter<PointT>::applyFilter (PointCloud &output)
7  {
8      if (sigma_s_ == 0)
9      {
10         PCL_ERROR ("[pcl::BilateralFilter::applyFilter] Need a sigma_s value given_
↳before continuing.\n");
11         return;
12     }
13     if (!tree_)
14     {
15         if (input_>isOrganized ())
16             tree_.reset (new pcl::OrganizedNeighbor<PointT> ());
17         else
18             tree_.reset (new pcl::KdTreeFLANN<PointT> (false));
19     }
20     tree_>setInputCloud (input_);
21     ...

```

The implementation file header thus becomes:

```

1  #ifndef PCL_FILTERS_BILATERAL_IMPL_H_
2  #define PCL_FILTERS_BILATERAL_IMPL_H_
3
4  #include <pcl/filters/bilateral.h>
5  #include <pcl/kdtree/kdtree_flann.h>
6  #include <pcl/kdtree/organized_data.h>
7
8  template <typename PointT> double
9  pcl::BilateralFilter<PointT>::computePointWeight (const int pid,
10                                                    const std::vector<int> &indices,

```

(continues on next page)

(continued from previous page)

```

11                                     const std::vector<float> &
12 distances)
13 {
14     double BF = 0, W = 0;
15     // For each neighbor
16     for (std::size_t n_id = 0; n_id < indices.size (); ++n_id)
17     {
18         double id = indices[n_id];
19         double dist = std::sqrt (distances[n_id]);
20         double intensity_dist = std::abs (input_->points[pid].intensity - input_->
11 points[id].intensity);
21
22         double weight = kernel (dist, sigma_s_) * kernel (intensity_dist, sigma_r_);
23
24         BF += weight * input_->points[id].intensity;
25         W += weight;
26     }
27     return (BF / W);
28 }
29
30 template <typename PointT> void
31 pcl::BilateralFilter<PointT>::applyFilter (PointCloud &output)
32 {
33     if (sigma_s_ == 0)
34     {
35         PCL_ERROR ("[pcl::BilateralFilter::applyFilter] Need a sigma_s value given_
11 before continuing.\n");
36         return;
37     }
38     if (!tree_)
39     {
40         if (input_->isOrganized ())
41             tree_.reset (new pcl::OrganizedNeighbor<PointT> ());
42         else
43             tree_.reset (new pcl::KdTreeFLANN<PointT> (false));
44     }
45     tree_->setInputCloud (input_);
46
47     std::vector<int> k_indices;
48     std::vector<float> k_distances;
49
50     output = *input_;
51
52     for (std::size_t i = 0; i < indices_->size (); ++i)
53     {
54         tree_->radiusSearch ((*indices_)[i], sigma_s_ * 2, k_indices, k_distances);
55
56         output.points[(*indices_)[i]].intensity = computePointWeight ((*indices_)[i], k_
11 indices, k_distances);
57     }
58 }
59
60 #define PCL_INSTANTIATE_BilateralFilter(T) template class PCL_EXPORTS_
11 pcl::BilateralFilter<T>;
61
62 #endif // PCL_FILTERS_BILATERAL_IMPL_H

```

To make `pcl::indices_<pcl::PCLBase::indices_>` work without typing the full construct, we need to add a new line to `bilateral.h` that specifies the class where `indices_` is declared:

```
1  ...
2  template<typename PointT>
3  class BilateralFilter : public Filter<PointT>
4  {
5      using Filter<PointT>::input_;
6      using Filter<PointT>::indices_;
7      public:
8          BilateralFilter () : sigma_s_ (0),
9      ...
```

## 14.5.2 Licenses

It is advised that each file contains a license that describes the author of the code. This is very useful for our users that need to understand what sort of restrictions are they bound to when using the code. PCL is 100% **BSD licensed**, and we insert the corpus of the license as a C++ comment in the file, as follows:

```
1  /*
2   * Software License Agreement (BSD License)
3   *
4   * Point Cloud Library (PCL) - www.pointclouds.org
5   * Copyright (c) 2010-2011, Willow Garage, Inc.
6   *
7   * All rights reserved.
8   *
9   * Redistribution and use in source and binary forms, with or without
10  * modification, are permitted provided that the following conditions
11  * are met:
12  *
13  *   * Redistributions of source code must retain the above copyright
14  *     notice, this list of conditions and the following disclaimer.
15  *   * Redistributions in binary form must reproduce the above
16  *     copyright notice, this list of conditions and the following
17  *     disclaimer in the documentation and/or other materials provided
18  *     with the distribution.
19  *   * Neither the name of Willow Garage, Inc. nor the names of its
20  *     contributors may be used to endorse or promote products derived
21  *     from this software without specific prior written permission.
22  *
23  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
24  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
25  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
26  * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
27  * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
28  * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
29  * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
30  * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
31  * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
32  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
33  * ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
34  * POSSIBILITY OF SUCH DAMAGE.
35  *
36  */
```

An additional like can be inserted if additional copyright is needed (or the original copyright can be changed):

```
* Copyright (c) XXX, respective authors.
```

### 14.5.3 Proper naming

We wrote the tutorial so far by using *silly named* setters and getters in our example, like `setSigmaS` or `setSigmaR`. In reality, we would like to use a better naming scheme, that actually represents what the parameter is doing. In a final version of the code we could therefore rename the setters and getters to `set/getHalfSize` and `set/getStdDev` or something similar.

### 14.5.4 Code comments

PCL is trying to maintain a *high standard* with respect to user and API documentation. This sort of Doxygen documentation has been stripped from the examples shown above. In reality, we would have had the `bilateral.h` header class look like:

```
/*
 * Software License Agreement (BSD License)
 *
 * Point Cloud Library (PCL) - www.pointclouds.org
 * Copyright (c) 2010-2011, Willow Garage, Inc.
 *
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * * Redistributions of source code must retain the above copyright
 *   notice, this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above
 *   copyright notice, this list of conditions and the following
 *   disclaimer in the documentation and/or other materials provided
 *   with the distribution.
 * * Neither the name of Willow Garage, Inc. nor the names of its
 *   contributors may be used to endorse or promote products derived
 *   from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 * ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#ifndef PCL_FILTERS_BILATERAL_H_
#define PCL_FILTERS_BILATERAL_H_
```

(continues on next page)

(continued from previous page)

```

40
41 #include <pcl/filters/filter.h>
42 #include <pcl/kdtree/kdtree.h>
43
44 namespace pcl
45 {
46     /** \brief A bilateral filter implementation for point cloud data. Uses the
47     ↪intensity data channel.
48     * \note For more information please see
49     * <b>C. Tomasi and R. Manduchi. Bilateral Filtering for Gray and Color Images.
50     * In Proceedings of the IEEE International Conference on Computer Vision,
51     * 1998.</b>
52     * \author Luca Penasa
53     */
54     template<typename PointT>
55     class BilateralFilter : public Filter<PointT>
56     {
57     public:
58         using Filter<PointT>::input_;
59         using Filter<PointT>::indices_;
60         typedef typename Filter<PointT>::PointCloud PointCloud;
61         typedef typename pcl::KdTree<PointT>::Ptr KdTreePtr;
62
63         /** \brief Constructor.
64         * Sets \ref sigma_s_ to 0 and \ref sigma_r_ to MAXDBL
65         */
66         BilateralFilter () : sigma_s_ (0),
67                             sigma_r_ (std::numeric_limits<double>::max ())
68         {
69
70
71         }
72
73         /** \brief Filter the input data and store the results into output
74         * \param[out] output the resultant point cloud message
75         */
76         void
77         applyFilter (PointCloud &output);
78
79         /** \brief Compute the intensity average for a single point
80         * \param[in] pid the point index to compute the weight for
81         * \param[in] indices the set of nearest neighbor indices
82         * \param[in] distances the set of nearest neighbor distances
83         * \return the intensity average at a given point index
84         */
85         double
86         computePointWeight (const int pid, const std::vector<int> &indices, const
87         ↪std::vector<float> &distances);
88
89         /** \brief Set the half size of the Gaussian bilateral filter window.
90         * \param[in] sigma_s the half size of the Gaussian bilateral filter window
91         ↪to use
92         */
93         inline void
94         setHalfSize (const double sigma_s)
95         {
96             sigma_s_ = sigma_s;
97         }
98     };

```

(continues on next page)

(continued from previous page)

```

94
95     /** \brief Get the half size of the Gaussian bilateral filter window as set by
96     ↳the user. */
97     double
98     getHalfSize () const
99     {
100         return (sigma_s_);
101     }
102
103     /** \brief Set the standard deviation parameter
104      * \param[in] sigma_r the new standard deviation parameter
105      */
106     void
107     setStdDev (const double sigma_r)
108     {
109         sigma_r_ = sigma_r;
110     }
111
112     /** \brief Get the value of the current standard deviation parameter of the
113     ↳bilateral filter. */
114     double
115     getStdDev () const
116     {
117         return (sigma_r_);
118     }
119
120     /** \brief Provide a pointer to the search object.
121      * \param[in] tree a pointer to the spatial search object.
122      */
123     void
124     setSearchMethod (const KdTreePtr &tree)
125     {
126         tree_ = tree;
127     }
128
129     private:
130
131     /** \brief The bilateral filter Gaussian distance kernel.
132      * \param[in] x the spatial distance (distance or intensity)
133      * \param[in] sigma standard deviation
134      */
135     inline double
136     kernel (double x, double sigma)
137     {
138         return (std::exp (- (x*x)/(2*sigma*sigma)));
139     }
140
141     /** \brief The half size of the Gaussian bilateral filter window (e.g.,
142     ↳spatial extents in Euclidean). */
143     double sigma_s_;
144
145     /** \brief The standard deviation of the bilateral filter (e.g., standard
146     ↳deviation in intensity). */
147     double sigma_r_;
148
149     /** \brief A pointer to the spatial search object. */
150     KdTreePtr tree_;
151 };

```

(continues on next page)

(continued from previous page)

```

147 }
148
149 #endif // PCL_FILTERS_BILATERAL_H_

```

And the *bilateral.hpp* likes:

```

1  /*
2   * Software License Agreement (BSD License)
3   *
4   * Point Cloud Library (PCL) - www.pointclouds.org
5   * Copyright (c) 2010-2011, Willow Garage, Inc.
6   *
7   * All rights reserved.
8   *
9   * Redistribution and use in source and binary forms, with or without
10  * modification, are permitted provided that the following conditions
11  * are met:
12  *
13  * * Redistributions of source code must retain the above copyright
14  *   notice, this list of conditions and the following disclaimer.
15  * * Redistributions in binary form must reproduce the above
16  *   copyright notice, this list of conditions and the following
17  *   disclaimer in the documentation and/or other materials provided
18  *   with the distribution.
19  * * Neither the name of Willow Garage, Inc. nor the names of its
20  *   contributors may be used to endorse or promote products derived
21  *   from this software without specific prior written permission.
22  *
23  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
24  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
25  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
26  * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
27  * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
28  * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
29  * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
30  * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
31  * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
32  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
33  * ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
34  * POSSIBILITY OF SUCH DAMAGE.
35  *
36  */
37
38 #ifndef PCL_FILTERS_BILATERAL_IMPL_H_
39 #define PCL_FILTERS_BILATERAL_IMPL_H_
40
41 #include <pcl/filters/bilateral.h>
42 #include <pcl/kdtree/kdtree_flann.h>
43 #include <pcl/kdtree/organized_data.h>
44
45 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
46 ↪ ///////////////////////////////////////////////////////////////////
47 template <typename PointT> double
48 pcl::BilateralFilter<PointT>::computePointWeight (const int pid,
49                                                    const std::vector<int> &indices,
50                                                    const std::vector<float> &
51 ↪ distances)

```

(continues on next page)

(continued from previous page)

```

50 {
51     double BF = 0, W = 0;
52
53     // For each neighbor
54     for (std::size_t n_id = 0; n_id < indices.size (); ++n_id)
55     {
56         double id = indices[n_id];
57         // Compute the difference in intensity
58         double intensity_dist = std::abs (input_>points[pid].intensity - input_>
↳points[id].intensity);
59
60         // Compute the Gaussian intensity weights both in Euclidean and in intensity_
↳space
61         double dist = std::sqrt (distances[n_id]);
62         double weight = kernel (dist, sigma_s_) * kernel (intensity_dist, sigma_r_);
63
64         // Calculate the bilateral filter response
65         BF += weight * input_>points[id].intensity;
66         W += weight;
67     }
68     return (BF / W);
69 }
70
71 //////////////////////////////////////
↳////////////////////////////////////
72 template <typename PointT> void
73 pcl::BilateralFilter<PointT>::applyFilter (PointCloud &output)
74 {
75     // Check if sigma_s has been given by the user
76     if (sigma_s_ == 0)
77     {
78         PCL_ERROR ("[pcl::BilateralFilter::applyFilter] Need a sigma_s value given_
↳before continuing.\n");
79         return;
80     }
81     // In case a search method has not been given, initialize it using some defaults
82     if (!tree_)
83     {
84         // For organized datasets, use an OrganizedNeighbor
85         if (input_>isOrganized ())
86             tree_.reset (new pcl::OrganizedNeighbor<PointT> ());
87         // For unorganized data, use a FLANN kdtree
88         else
89             tree_.reset (new pcl::KdTreeFLANN<PointT> (false));
90     }
91     tree_>setInputCloud (input_);
92
93     std::vector<int> k_indices;
94     std::vector<float> k_distances;
95
96     // Copy the input data into the output
97     output = *input_;
98
99     // For all the indices given (equal to the entire cloud if none given)
100     for (std::size_t i = 0; i < indices_>size (); ++i)
101     {
102         // Perform a radius search to find the nearest neighbors

```

(continues on next page)

(continued from previous page)

```

103     tree->radiusSearch ((*indices_)[i], sigma_s_ * 2, k_indices, k_distances);
104
105     // Overwrite the intensity value with the computed average
106     output.points[(*indices_)[i]].intensity = computePointWeight ((*indices_)[i], k_
->indices, k_distances);
107 }
108 }
109
110 #define PCL_INSTANTIATE_BilateralFilter(T) template class PCL_EXPORTS_
->pcl::BilateralFilter<T>;
111
112 #endif // PCL_FILTERS_BILATERAL_IMPL_H_

```

## 14.6 Testing the new class

Testing the new class is easy. We'll take the first code snippet example as shown above, strip the algorithm, and make it use the `pcl::BilateralFilter` class instead:

```

1  #include <pcl/point_types.h>
2  #include <pcl/io/pcd_io.h>
3  #include <pcl/kdtree/kdtree_flann.h>
4  #include <pcl/filters/bilateral.h>
5
6  typedef pcl::PointXYZI PointT;
7
8  int
9  main (int argc, char *argv[])
10 {
11     std::string incloudfile = argv[1];
12     std::string outcloudfile = argv[2];
13     float sigma_s = atof (argv[3]);
14     float sigma_r = atof (argv[4]);
15
16     // Load cloud
17     pcl::PointCloud<PointT>::Ptr cloud (new pcl::PointCloud<PointT>);
18     pcl::io::loadPCDFile (incloudfile.c_str (), *cloud);
19
20     pcl::PointCloud<PointT> outcloud;
21
22     // Set up KDTree
23     pcl::KdTreeFLANN<PointT>::Ptr tree (new pcl::KdTreeFLANN<PointT>);
24
25     pcl::BilateralFilter<PointT> bf;
26     bf.setInputCloud (cloud);
27     bf.setSearchMethod (tree);
28     bf.setHalfSize (sigma_s);
29     bf.setStdDev (sigma_r);
30     bf.filter (outcloud);
31
32     // Save filtered output
33     pcl::io::savePCDFile (outcloudfile.c_str (), outcloud);
34     return (0);
35 }

```

---

## How 3D Features work in PCL

---

This document presents an introduction to the 3D feature estimation methodologies in PCL, and serves as a guide for users or developers that are interested in the internals of the `pcl::Feature` class.

### Contents

- *How 3D Features work in PCL*
  - *Theoretical primer*
  - *Terminology*
  - *How to pass the input*
  - *An example for normal estimation*

## 15.1 Theoretical primer

From [RusuDissertation]:

*In their native representation, **points** as defined in the concept of 3D mapping systems are simply represented using their Cartesian coordinates  $x, y, z$ , with respect to a given origin. Assuming that the origin of the coordinate system does not change over time, there could be two points  $p1$  and  $p2$ , acquired at  $t1$  and  $t2$ , having the same coordinates. Comparing these points however is an ill-posed problem, because even though they are equal with respect to some distance measure (e.g. Euclidean metric), they could be sampled on completely different surfaces, and thus represent totally different information when taken together with the other surrounding points in their vicinity. That is because there are no guarantees that the world has not changed between  $t1$  and  $t2$ . Some acquisition devices might provide extra information for a sampled point, such as an intensity or surface remission value, or even a color; however that does not solve the problem completely and the comparison remains ambiguous.*

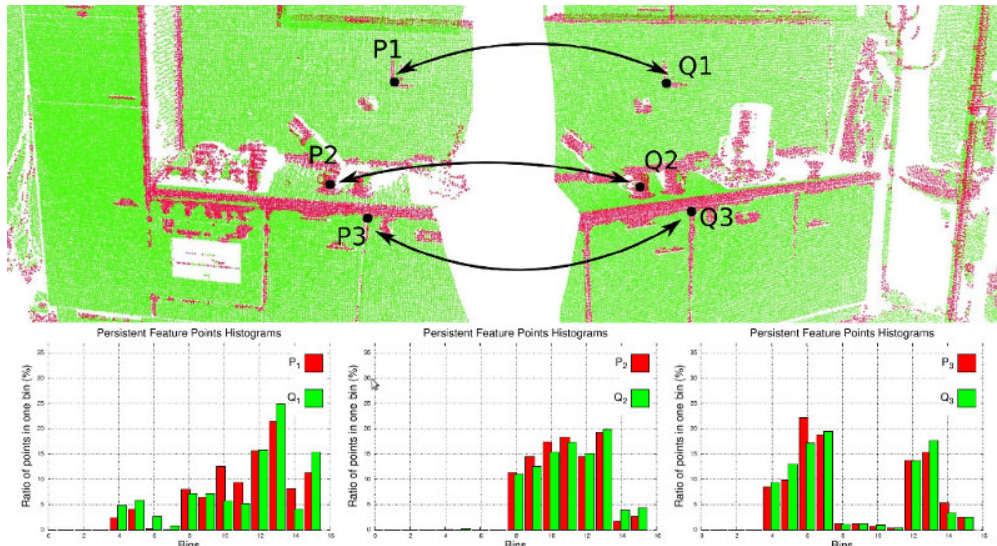
*Applications which need to compare points for various reasons require better characteristics and metrics to be able to distinguish between geometric surfaces. The concept of a 3D point as a singular entity with*

Cartesian coordinates therefore disappears, and a new concept, that of **local descriptor** takes its place. The literature is abundant of different naming schemes describing the same conceptualization, such as **shape descriptors** or **geometric features** but for the remaining of this document they will be referred to as **point feature representations**.

...

By including the surrounding neighbors, the underlying sampled surface geometry can be inferred and captured in the feature formulation, which contributes to solving the ambiguity comparison problem. Ideally, the resultant features would be very similar (with respect to some metric) for points residing on the same or similar surfaces, and different for points found on different surfaces, as shown in the figure below. A **good** point feature representation distinguishes itself from a **bad** one, by being able to capture the same local surface characteristics in the presence of:

- **rigid transformations** - that is, 3D rotations and 3D translations in the data should not influence the resultant feature vector  $F$  estimation;
- **varying sampling density** - in principle, a local surface patch sampled more or less densely should have the same feature vector signature;
- **noise** - the point feature representation must retain the same or very similar values in its feature vector in the presence of mild noise in the data.



In general, PCL features use approximate methods to compute the nearest neighbors of a query point, using fast kd-tree queries. There are two types of queries that we're interested in:

- determine the **k** (user given parameter) neighbors of a query point (also known as *k-search*);
- determine **all the neighbors** of a query point within a sphere of radius **r** (also known as *radius-search*).

---

**Note:** For a discussion on what the right **k** or **r** values should be, please see [\[RusuDissertation\]](#).

---

## 15.2 Terminology

For the reminder of this article, we will make certain abbreviations and introduce certain notations, to simplify the in-text explanations. Please see the table below for a reference on each of the terms used.

## 15.3 How to pass the input

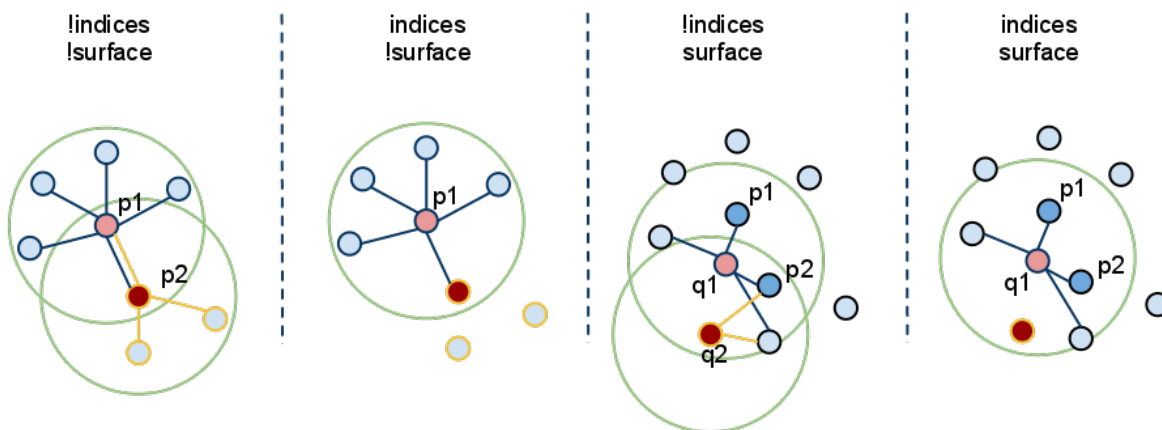
As almost all classes in PCL that inherit from the base `pcl::PCLBase` class, the `pcl::Feature` class accepts input data in two different ways:

1. an entire point cloud dataset, given via **`setInputCloud (PointCloudConstPtr &)`** - **mandatory**  
Any feature estimation class with attempt to estimate a feature at **every** point in the given input cloud.
2. a subset of a point cloud dataset, given via **`setInputCloud (PointCloudConstPtr &)`** and **`setIndices (IndicesConstPtr &)`** - **optional**

Any feature estimation class will attempt to estimate a feature at every point in the given input cloud that has an index in the given indices list. *By default, if no set of indices is given, all points in the cloud will be considered.\**

In addition, the set of point neighbors to be used, can be specified through an additional call, **`setSearchSurface (PointCloudConstPtr &)`**. This call is optional, and when the search surface is not given, the input point cloud dataset is used instead by default.

Because **`setInputCloud()`** is always required, there are up to four combinations that we can create using `<setInputCloud(), setIndices(), setSearchSurface(>`. Say we have two point clouds,  $P=\{p_1, p_2, \dots p_n\}$  and  $Q=\{q_1, q_2, \dots, q_n\}$ . The image below presents all four cases:



- **`setIndices() = false, setSearchSurface() = false`** - this is without a doubt the most used case in PCL, where the user is just feeding in a single PointCloud dataset and expects a certain feature estimated at *all the points in the cloud*.

Since we do not expect to maintain different implementation copies based on whether a set of indices and/or the search surface is given, whenever **`indices = false`**, PCL creates a set of internal indices (as a `std::vector<int>`) that basically point to the entire dataset (`indices=1..N`, where  $N$  is the number of points in the cloud).

In the figure above, this corresponds to the leftmost case. First, we estimate the nearest neighbors of  $p_1$ , then the nearest neighbors of  $p_2$ , and so on, until we exhaust all the points in  $P$ .

- **`setIndices() = true, setSearchSurface() = false`** - as previously mentioned, the feature estimation method will only compute features for those points which have an index in the given indices vector;

In the figure above, this corresponds to the second case. Here, we assume that  $p_2$ 's index is not part of the indices vector given, so no neighbors or features will be estimated at  $p_2$ .

- **`setIndices() = false, setSearchSurface() = true`** - as in the first case, features will be estimated for all points given as input, but, the underlying neighboring surface given in **`setSearchSurface()`** will be used to obtain nearest neighbors for the input points, rather than the input cloud itself;

In the figure above, this corresponds to the third case. If  $Q=\{q_1, q_2\}$  is another cloud given as input, different than  $P$ , and  $P$  is the search surface for  $Q$ , then the neighbors of  $q_1$  and  $q_2$  will be computed from  $P$ .

- **setIndices() = true, setSearchSurface() = true** - this is probably the rarest case, where both indices and a search surface is given. In this case, features will be estimated for only a subset from the `<input, indices>` pair, using the search surface information given in **setSearchSurface()**.

Finally, in the figure above, this corresponds to the last (rightmost) case. Here, we assume that  $q_2$ 's index is not part of the indices vector given for  $Q$ , so no neighbors or features will be estimated at  $q_2$ .

The most useful example when **setSearchSurface()** should be used, is when we have a very dense input dataset, but we do not want to estimate features at all the points in it, but rather at some keypoints discovered using the methods in `pcl_keypoints`, or at a downsampled version of the cloud (e.g., obtained using a `pcl::VoxelGrid<T>` filter). In this case, we pass the downsampled/keypoints input via **setInputCloud()**, and the original data as **setSearchSurface()**.

## 15.4 An example for normal estimation

Once determined, the neighboring points of a query point can be used to estimate a local feature representation that captures the geometry of the underlying sampled surface around the query point. An important problem in describing the geometry of the surface is to first infer its orientation in a coordinate system, that is, estimate its normal. Surface normals are important properties of a surface and are heavily used in many areas such as computer graphics applications to apply the correct light sources that generate shadings and other visual effects (See [RusuDissertation] for more information).

The following code snippet will estimate a set of surface normals for all the points in the input dataset.

```

1  #include <pcl/point_types.h>
2  #include <pcl/features/normal_3d.h>
3
4  {
5      pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::PointXYZ>);
6
7      ... read, pass in or create a point cloud ...
8
9      // Create the normal estimation class, and pass the input dataset to it
10     pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> ne;
11     ne.setInputCloud (cloud);
12
13     // Create an empty kdtree representation, and pass it to the normal estimation_
↪ object.
14     // Its content will be filled inside the object, based on the given input dataset_
↪ (as no other search surface is given).
15     pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZ>
↪ ());
16     ne.setSearchMethod (tree);
17
18     // Output datasets
19     pcl::PointCloud<pcl::Normal>::Ptr cloud_normals (new pcl::PointCloud<pcl::Normal>);
20
21     // Use all neighbors in a sphere of radius 3cm
22     ne.setRadiusSearch (0.03);
23
24     // Compute the features
25     ne.compute (*cloud_normals);
26

```

(continues on next page)

(continued from previous page)

```

27 // cloud_normals->points.size () should have the same size as the input cloud->
    ↪ points.size ()
28 }

```

The following code snippet will estimate a set of surface normals for a subset of the points in the input dataset.

```

1  #include <pcl/point_types.h>
2  #include <pcl/features/normal_3d.h>
3
4  {
5      pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::PointXYZ>);
6
7      ... read, pass in or create a point cloud ...
8
9      // Create a set of indices to be used. For simplicity, we're going to be using the
    ↪ first 10% of the points in cloud
10     std::vector<int> indices (std::floor (cloud->points.size () / 10));
11     for (std::size_t i = 0; i < indices.size (); ++i) indices[i] = i;
12
13     // Create the normal estimation class, and pass the input dataset to it
14     pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> ne;
15     ne.setInputCloud (cloud);
16
17     // Pass the indices
18     pcl::shared_ptr<std::vector<int> > indicesptr (new std::vector<int> (indices));
19     ne.setIndices (indicesptr);
20
21     // Create an empty kdtree representation, and pass it to the normal estimation
    ↪ object.
22     // Its content will be filled inside the object, based on the given input dataset
    ↪ (as no other search surface is given).
23     pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZ>
    ↪ ());
24     ne.setSearchMethod (tree);
25
26     // Output datasets
27     pcl::PointCloud<pcl::Normal>::Ptr cloud_normals (new pcl::PointCloud<pcl::Normal>);
28
29     // Use all neighbors in a sphere of radius 3cm
30     ne.setRadiusSearch (0.03);
31
32     // Compute the features
33     ne.compute (*cloud_normals);
34
35     // cloud_normals->points.size () should have the same size as the input indicesptr->
    ↪ size ()
36 }

```

Finally, the following code snippet will estimate a set of surface normals for all the points in the input dataset, but will estimate their nearest neighbors using another dataset. As previously mentioned, a good usecase for this is when the input is a downsampled version of the surface.

```

1  #include <pcl/point_types.h>
2  #include <pcl/features/normal_3d.h>
3
4  {

```

(continues on next page)

(continued from previous page)

```
5   pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::PointXYZ>);
6   pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_downsampled (new pcl::PointCloud
↪<pcl::PointXYZ>);
7
8   ... read, pass in or create a point cloud ...
9
10  ... create a downsampled version of it ...
11
12  // Create the normal estimation class, and pass the input dataset to it
13  pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> ne;
14  ne.setInputCloud (cloud_downsampled);
15
16  // Pass the original data (before downsampling) as the search surface
17  ne.setSearchSurface (cloud);
18
19  // Create an empty kdtree representation, and pass it to the normal estimation_
↪object.
20  // Its content will be filled inside the object, based on the given surface dataset.
21  pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZ>
↪());
22  ne.setSearchMethod (tree);
23
24  // Output datasets
25  pcl::PointCloud<pcl::Normal>::Ptr cloud_normals (new pcl::PointCloud<pcl::Normal>);
26
27  // Use all neighbors in a sphere of radius 3cm
28  ne.setRadiusSearch (0.03);
29
30  // Compute the features
31  ne.compute (*cloud_normals);
32
33  // cloud_normals->points.size () should have the same size as the input cloud_
↪downsampled->points.size ()
34 }
```

---

**Note:** @PhDThesis{RusuDoctoralDissertation, author = {Radu Bogdan Rusu}, title = {Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments}, school = {Computer Science department, Technische Universitaet Muenchen, Germany}, year = {2009}, month = {October} }

---

## CHAPTER 16

---

### Indices and tables

---

- `genindex`
- `modindex`
- *Search*



---

## Bibliography

---

[RusuDissertation] <http://mediatum.ub.tum.de/doc/800632/941254.pdf>