

# The PCL Plugin for ParaView — Developer Guide

P. Marion, R. Kwitt and B. Davis

Kitware Inc., June 14, 2012, **Version 1.0**

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Download &amp; Build Instructions</b>	<b>2</b>
2.1	Download and build ParaView . . . . .	2
2.2	Download and build PCL . . . . .	3
2.3	Download and build the PCL Plugin . . . . .	3
<b>3</b>	<b>Contributing to the PCL Plugin</b>	<b>4</b>
<b>4</b>	<b>How to cite the PCL Plugin</b>	<b>4</b>
<b>5</b>	<b>Notation</b>	<b>4</b>
<b>6</b>	<b>Core Functionality</b>	<b>4</b>
6.1	Data Type Compatibility . . . . .	4
6.2	Creating Attribute Arrays . . . . .	5
<b>7</b>	<b>Wrapping PCL algorithms as VTK Filters</b>	<b>6</b>
7.1	vtkPCLRadiusOutlierRemoval — Radius-based Outlier Removal . . . . .	6
7.2	vtkPCLNormalEstimation — Estimating Surface Normals . . . . .	8
<b>8</b>	<b>Using the Python Bindings</b>	<b>10</b>

## About this Document

This is the developer guide to the *PCL Plugin for ParaView*, v1.0. The document was created using  $\text{\LaTeX}$  and `minted.sty`<sup>1</sup> for typesetting source code listings.

## 1 Introduction

This developer guide provides a reference for researchers in the field of point cloud processing to use the PCL Plugin for ParaView. Throughout the developer guide, we'll refer to the PCL Plugin for Paraview as the *PCL Plugin* or just the *plugin*. This guide addresses issues such as data type compatibility, wrapping PCL algorithms as VTK filters, or extending the plugin by implementing new functionality. We assume that the reader is somewhat familiar with VTK filter development and refer to [2] for further information on VTK details. For information about ParaView, we recommend taking a look at the ParaView Wiki.<sup>2</sup>

*Note: While we are working on extending the plugin, this guide will get constantly updated to reflect the latest changes.*

## 2 Download & Build Instructions

The PCL plugin is currently distributed as a stand-alone tarball. It might — at some point — be distributed directly with ParaView. For now, we support building against ParaView v3.14 and PCL v1.5.1. We are however working on supporting the latest PCL trunk.

We propose a three-stage build process to compile and run the PCL plugin for ParaView: 1) building a vanilla ParaView, 2) building PCL (with ParaView's VTK) and 3) building the PCL plugin. While other strategies are possible, we strongly recommend to follow this process to ensure that both PCL and ParaView use the same VTK version.

### 2.1 Download and build ParaView

Download the ParaView 3.14.1 source tarball<sup>3</sup> and follow the build instructions (on the Wiki) to compile ParaView. In general, it should suffice to 1) unpack the tarball, 2) create a `Build` directory and 3) run `cmake` to modify or adapt `cmake` variables.

```
$ tar xvfz ParaView-3.14.1-Source.tar.gz
$ cd ParaView-3.14.1-Source
$ mkdir Build
$ cmake ..
```

---

<sup>1</sup><http://code.google.com/p/minted/>

<sup>2</sup><http://www.paraview.org/Wiki/ParaView>

<sup>3</sup>available at <http://www.paraview.org/paraview/resources/software.php>

**Note:** Make sure that `BUILD_SHARED_LIBS` is set to `ON`. Hitting 'c' followed by 'g' will configure and generate all files for the build process. Running

```
$ make
```

will build ParaView as well as all the submodules (such as VTK). Optionally, you can specify `-jN`, e.g., `make -j4`, to speed up the compilation process by using multiple cores.

## 2.2 Download and build PCL

Once we have build a vanilla ParaView, we can build PCL against ParaView's VTK version. For detailed instructions on how to obtain PCL, we refer to PCL's installation instructions<sup>4</sup> for compiling from source. For convenience, we repeat the basic commands: 1) Unpack the tarball, 2) Create a Build directory, and 3) Run `ccmake`:

```
% tar xvjpf PCL-1.5.1-Source.tar.bz2
$ cd PCL-1.5.1-Source
$ mkdir Build
$ ccmake ..
```

**Note:** Ensure that you set the `VTK_DIR` variable correctly, i.e., the VTK directory in ParaView's Build dir, i.e., `<FullPathToParaView>/ParaView-3.14.1-Source/Build/VTK`. After customizing any other `cmake` variables, run `make` to build PCL.

## 2.3 Download and build the PCL Plugin

Now that everything is set up, we are ready to actually build the plugin. First, download the tarball<sup>5</sup> and unpack it

```
$ tar xvfz PointCloudLibraryPlugin-v1.0.tar.gz
```

Then, create a Build directory and run `ccmake`, i.e.,

```
$ cd PointCloudLibraryPlugin-v1.0
$ mkdir Build
$ ccmake ..
```

**Note:** Make sure to set the `ParaView_DIR` and `PCL_DIR` `cmake` variables to

```
PCL_DIR      <FullPathToPCLSource>/PCL-1.5.1-Source/Build
ParaView_DIR <FullPathToParaViewSource>/ParaView-3.14.1-Source/Build
```

---

<sup>4</sup>see <http://www.pointclouds.org>

<sup>5</sup>available at <http://www.paraview.org/Wiki/images/d/d2/PointCloudLibraryPlugin-v1.0.tar.gz>

Hit 'c' and 'g' to configure and generate. Alternatively, you can set all on the command line by

```
$ cmake -DParaView_DIR=<...> -DPCL_DIR=<...> ..
```

Running make will eventually compile the plugin and make it available for use within ParaView.

### 3 Contributing to the PCL Plugin

tbd.

### 4 How to cite the PCL Plugin

In case you use the plugin for academic publications, please cite the following paper:

```
@inproceedings{Marion12a,  
  author    = {P.~Marion and R.~Kwitt, B.~Davis and M. Gschwandtner},  
  title     = {PCL and ParaView - Connecting the Dots},  
  booktitle = {CVPR Workshop on Point Cloud Processing (PCP)},  
  year      = 2012}
```

### 5 Notation

Throughout this document, class names will be referred to in boldface, e.g., **vtkPCLConversions**. Data types and filenames will be referred to in typewriter font, e.g., `vtkPolyData` or `vtkPCLConversions.h`. PCL classes will always be prefixed by the namespace `pcl`, e.g., `pcl::PointCloud` to avoid confusion with VTK data types. All source code listings shown in this document also list the corresponding file in the PCL plugin source directory.

### 6 Core Functionality

#### 6.1 Data Type Compatibility

The core routines of the PCL plugin facilitate conversion between PCL and VTK data types. In PCL, point clouds are stored in a `pcl::PointCloud` container that holds points of a specific point type `PointT`, e.g., `pcl::PointXYZ` or `pcl::PointXYZRGB`. The basic data structure to represent point cloud data in the plugin, however, is VTK's `vtkPolyData`. A `vtkPolyData` object can hold vertices, cells and attributes. In order to use PCL algorithms we need to be able to convert back and forth between those data structures. This functionality is implemented in the **vtkPCLConversions** class.

The basic routine to enable the use of PCL's processing algorithms is to convert VTK's `vtkPolyData` to PCL's `pcl::PointCloud` container, using a suitable PCL point type. At the current stage of development, we only support `pcl::PointXYZ`. The declaration of the conversion routine is shown in Listing 1.

---

```
_____ see vtkPCLConversions.h _____  
// VTK -> PCL (PointXYZ)  
static pcl::PointCloud<pcl::PointXYZ>::Ptr PointCloudFromPolyData(  
    vtkPolyData* polyData);
```

---

**Listing 1:** Conversion routines (VTK to PCL).

Converting back from a PCL point clouds to a `vtkPolyData` data object is more versatile in the sense that we support three PCL point types, i.e., `pcl::PointXYZ`, `pcl::PointXYZRGB` and `pcl::PointXYZRGBA`. The conversion routines are shown in Listing 2. All of them return a `vtkSmartPointer` to a `vtkPolyData` instance.

---

```
_____ see vtkPCLConversions.h _____  
// PCL (PointXYZ) -> VTK  
static vtkSmartPointer<vtkPolyData> PolyDataFromPointCloud(  
    pcl::PointCloud<pcl::PointXYZ>::ConstPtr cloud);  
  
// PCL (PointXYZRGB) -> VTK  
static vtkSmartPointer<vtkPolyData> PolyDataFromPointCloud(  
    pcl::PointCloud<pcl::PointXYZRGB>::ConstPtr cloud);  
  
// PCL (PointXYZRGBA) -> VTK  
static vtkSmartPointer<vtkPolyData> PolyDataFromPointCloud(  
    pcl::PointCloud<pcl::PointXYZRGBA>::ConstPtr cloud);
```

---

**Listing 2:** Conversion routines (PCL to VTK).

## 6.2 Creating Attribute Arrays

Apart from converting point clouds, the plugin also provides a few methods to create new attribute arrays from different types of PCL representations of index vectors. The static `NewLabelsArray` method of the `vtkPCLConversions` class supports exactly this functionality. The method is overloaded three times, each time taking a different type of PCL index vector as a first argument and the number of elements in that vector as a second argument. The method declarations are shown in Listing 3.

---

```

                                see vtkPCLConversions.h
static vtkSmartPointer<vtkIntArray> NewLabelsArray(
    pcl::IndicesConstPtr indices, vtkIdType length);

static vtkSmartPointer<vtkIntArray> NewLabelsArray(
    pcl::PointIndices::ConstPtr indices, vtkIdType length);

static vtkSmartPointer<vtkIntArray> NewLabelsArray(
    const std::vector<pcl::PointIndices>& indices, vtkIdType length);

```

---

**Listing 3:** Creating new label (attribute) arrays.

## 7 Wrapping PCL algorithms as VTK Filters

Having point clouds in PCL’s point cloud data container allows to call PCL’s processing algorithms. The output of a PCL algorithm eventually needs to be converted back to a suitable VTK data type, so that the data can be visualized and further processed in ParaView.

The conversion routines in 6.1 are very generic and it’s not always necessary to return a new point cloud upon the completion of a PCL algorithm. For instance, PCL’s *VoxelGrid* filter requires to return a new point cloud, however, an outlier removal filter does not have that requirement. In fact, it suffices to only return the indices of the outliers (or inliers, resp.) and then use ParaView’s internal thresholding mechanism to prune the point cloud.

The PCL Plugin follows exactly that strategy of avoiding to return new point clouds whenever it is possible. Instead, attributes (or attribute arrays, resp.) are appended to existing point clouds, represented as `vtkPolyData`. The following examples of PCL algorithms, implemented as VTK filters, illustrate the interaction between PCL and VTK data structures and show the difference between returning new points clouds versus adding new attributes.

### 7.1 `vtkPCLRadiusOutlierRemoval` — Radius-based Outlier Removal

As an introductory example, we show how to implement *Radius-based Outlier Removal* as a VTK filter that can be used within ParaView. This is implemented in the `vtkPCLRadiusOutlierRemoval` class. The filter takes two input arguments: 1) a search radius and 2) the number of neighbors for a point *not* to be an outlier.

To implement the VTK filter, the `vtkPCLRadiusOutlierRemoval` class is derived from VTK’s `vtkPolyDataAlgorithm` class. `vtkPolyDataAlgorithm` is a superclass for algorithms that produce `vtkPolyData` as output. Listing 4 shows the important parts of the class declaration. Note that we use `vtkGetMacro` and `vtkSetMacro` to implement the *Get/Set* routines for the filter configuration parameters `SearchRadius` and `NeighborsInSearchRadius`.

The constructor of that class is shown in Listing 5. The constructor sets the default configuration parameter values as well as the number of input and output ports of the filter (usually 1/1). We will later see an example of using two input ports.

---

```

see vtkPCLRadiusOutlierRemoval.h
class vtkPCLRadiusOutlierRemoval : public vtkPolyDataAlgorithm
{
public:
    vtkTypeMacro(vtkPCLRadiusOutlierRemoval, vtkPolyDataAlgorithm);
    void PrintSelf(ostream& os, vtkIndent indent);

    static vtkPCLRadiusOutlierRemoval *New();

    vtkSetMacro(SearchRadius, double);
    vtkGetMacro(SearchRadius, double);

    vtkSetMacro(NeighborsInSearchRadius, int);
    vtkGetMacro(NeighborsInSearchRadius, int);

protected:

    double SearchRadius;
    int NeighborsInSearchRadius;

    virtual int RequestData(vtkInformation *request,
                            vtkInformationVector **inputVector,
                            vtkInformationVector *outputVector);

    vtkPCLRadiusOutlierRemoval();
    virtual ~vtkPCLRadiusOutlierRemoval();

private:
    vtkPCLRadiusOutlierRemoval(const vtkPCLRadiusOutlierRemoval&); // Not implemented.
    void operator=(const vtkPCLRadiusOutlierRemoval&); // Not implemented.
};

```

---

**Listing 4:** Class declaration of the `vtkPCLRadiusOutlierRemoval` class.

The interface to PCL, i.e., calling the implementation of the algorithm in PCL, is implemented in the `ApplyRadiusOutlierRemoval` routine. Note that this routine is not a class method. Listing 6 shows the corresponding source code (which closely resembles one of the PCL tutorials on outlier removal).

The actual implementation of the VTK filter’s functionality is done in the `RequestData` routine that is shown in Listing 7. First, we obtain pointers to the input and output data. We then create a PCL point cloud from the `vtkPolyData` objects using our conversion routine `PointCloudFromPolyData` and eventually call the `vtkPCLOutlierRemoval` routine. Next, a new attribute array `labels` is created. This is done by using the static `NewLabelsArray` method of the `vtkPCLConversions` class, explained in Section 6.2. In our example, the `labels` array holds a binary-valued attribute, indicating whether a point is a model inlier (i.e., 0) or a model outlier (i.e., 1). The attribute array is of dimensionality  $N \times 1$  where  $N$  denotes the number of points in the point cloud (obtained via `input->GetNumberOfPoints()`). By setting the attribute name to `is_outlier`, we can refer to that name in ParaView for visualization or further processing (e.g., thresholding). In the last step, we shallow-copy the

---

```

see vtkPCLRadiusOutlierRemoval.cxx
vtkPCLRadiusOutlierRemoval::vtkPCLRadiusOutlierRemoval()
{
    this->SearchRadius = 0.3;
    this->NeighborsInSearchRadius = 10;
    this->SetNumberOfInputPorts(1);
    this->SetNumberOfOutputPorts(1);
}

```

---

**Listing 5:** Constructor of the `vtkPCLRadiusOutlierRemoval` class.

---

```

see vtkPCLRadiusOutlierRemoval.cxx
pcl::IndicesConstPtr ApplyRadiusOutlierRemoval(pcl::PointCloud<pcl::PointXYZ>::ConstPtr cloud,
                                               double searchRadius,
                                               int neighborsInSearchRadius)
{
    if (!cloud || !cloud->points.size())
    {
        return pcl::IndicesConstPtr(new std::vector<int>);
    }

    pcl::PointCloud<pcl::PointXYZ>::Ptr cloudFiltered (new pcl::PointCloud<pcl::PointXYZ>);
    pcl::RadiusOutlierRemoval<pcl::PointXYZ> outrem(true);
    outrem.setInputCloud(cloud);
    outrem.setRadiusSearch(searchRadius);
    outrem.setMinNeighborsInRadius(neighborsInSearchRadius);
    outrem.filter(*cloudFiltered);

    return outrem.getRemovedIndices();
}

```

---

**Listing 6:** Calling the PCL algorithm.

input to the output (i.e., the input point cloud will be our output point cloud) and append the labels attribute array.

## 7.2 `vtkPCLNormalEstimation` — Estimating Surface Normals

As a second example, we discuss the PCL plugin implementation of PCL’s surface normal estimation algorithm. The example illustrates how to deal with PCL algorithms that do return point clouds with different types of data points, e.g., `pcl::Normal`. Surface normal estimation is implemented in PCL’s `pcl::NormalEstimation` class. As output, we obtain a `pcl::PointCloud` with `pcl::Normal` elements.

In the PCL plugin, normal estimation is implemented in the `vtkPCLNormalEstimation` class. In contrast to our previous example, the filter operates on *two* input point clouds, A and B, and requires a search radius to be specified. Passing two point clouds has the advantage that we can leverage PCL’s `setSearchSurface` capability which allows to compute surface normals for A, based on a different — usually smaller — set of points B. A



---

```

int vtkPCLRadiusOutlierRemoval::RequestData(
    vtkInformation* vtkNotUsed(request),
    vtkInformationVector **inputVector,
    vtkInformationVector *outputVector)
{
    // get input and output data objects
    vtkInformation *inInfo = inputVector[0]->GetInformationObject(0);
    vtkPolyData *input =
        vtkPolyData::SafeDownCast(inInfo->Get(vtkDataObject::DATA_OBJECT()));

    vtkInformation *outInfo = outputVector->GetInformationObject(0);
    vtkPolyData *output =
        vtkPolyData::SafeDownCast(outInfo->Get(vtkDataObject::DATA_OBJECT()));

    // perform outlier removal
    pcl::PointIndices::Ptr inlierIndices;
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud =
        vtkPCLConversions::PointCloudFromPolyData(input);

    pcl::IndicesConstPtr outlierIndices = ApplyRadiusOutlierRemoval(cloud,
        this->SearchRadius,
        this->NeighborsInSearchRadius);

    // pass thru input add labels
    vtkSmartPointer<vtkIntArray> labels = vtkPCLConversions::NewLabelsArray(
        outlierIndices, input->GetNumberOfPoints());
    labels->SetName("is_outlier");
    output->ShallowCopy(input);
    output->GetPointData()->AddArray(labels);

    return 1;
}

```

---

**Listing 7:** Core filter routine.

commonly used practice is to downsample the point cloud by means of a VoxelGrid filter and use the resulting decimated set of points as the search cloud B.

Calling the PCL algorithm is implemented by the ComputeNormalEstimation routine (see Listing 8). Note that this function is not a class method.

While the **vtkPCLNormalEstimation** class declaration is very similar (not shown here) to the one of the previous example, there are several interesting differences in the actual implementation that we need to point out: The first difference is in the constructor of the filter class, shown in Listing 9. Since we want to support to input two point clouds, we need to set two input ports, as opposed to one input and one output port in the **vtkPCLRadiusOutlierRemoval** class.

In Listing 10, we show the first part of the RequestData routine of the **vtkPCLNormalEstimation** filter that is responsible for fetching the two input point clouds provided to the filter. In case no second cloud is provided, no search cloud is set and the first one is

---

```

see vtkPCLNormalEstimation.cxx
pcl::PointCloud<pcl::Normal>::Ptr ComputeNormalEstimation(
    pcl::PointCloud<pcl::PointXYZ>::ConstPtr cloud,
    pcl::PointCloud<pcl::PointXYZ>::ConstPtr searchCloud,
    double searchRadius)
{
    pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> ne;
    pcl::PointCloud<pcl::Normal>::Ptr cloud_normals(new pcl::PointCloud<pcl::Normal>);
    pcl::search::KdTree<pcl::PointXYZ>::Ptr tree(new pcl::search::KdTree<pcl::PointXYZ> ());

    ne.setSearchMethod(tree);
    ne.setInputCloud(cloud);
    if (searchCloud)
    {
        ne.setSearchSurface(searchCloud);
    }

    ne.setRadiusSearch(searchRadius);
    ne.compute(*cloud_normals);
    return cloud_normals;
}

```

---

**Listing 8:** Calling PCL’s normal estimation algorithm.

---

```

see vtkPCLNormalEstimation.cxx
vtkPCLNormalEstimation::vtkPCLNormalEstimation()
{
    this->SearchRadius = 0.1;
    this->SetNumberOfInputPorts(2);
    this->SetNumberOfOutputPorts(1);
}

```

---

**Listing 9:** Constructor of the `vtkPCLNormalEstimation` class.

used for normal estimation.

In the second part of the filter’s `RequestData` routine, shown in Listing 11, we create a new attribute array `normals` and shallow copy the first input point cloud. The new attribute array is then appended. The attribute `normals` will contain a normal vector for each point of the first input point cloud. We use our conversion routines to convert both input clouds to PCL’s point cloud data type `pcl::PointCloud<pcl::PointXYZ>` and call our `ComputeNormalEstimation` routine returning a pointer to a `pcl::Pointcloud` with `<pcl::Normal>` elements. Eventually, we fill the `normals` attribute array with the returned surface normal vector information.

## 8 Using the Python Bindings

tbd.

---

```

...
see vtkPCLNormalEstimation.cxx
...
vtkInformation *outInfo = outputVector->GetInformationObject(0);
vtkPolyData *output = vtkPolyData::SafeDownCast(outInfo->Get(vtkDataObject::DATA_OBJECT()));

// information on first port
vtkInformation *inInfo = inputVector[0]->GetInformationObject(0);
vtkPolyData *input = vtkPolyData::SafeDownCast(inInfo->Get(vtkDataObject::DATA_OBJECT()));

// information on second port
vtkInformation *searchInfo = inputVector[1]->GetInformationObject(0);
vtkPolyData *searchCloudPolyData = 0;
if (searchInfo)
{
    searchCloudPolyData =
        vtkPolyData::SafeDownCast(searchInfo->Get(vtkDataObject::DATA_OBJECT()));
}
...

```

---

**Listing 10:** RequestData routine of the `vtkPCLNormalEstimation` filter (Part I).

## References

- [1] P. Marion, B. Davis R. Kwitt, and M. Gschwandtner. PCL and ParaView - connecting the dots. In *CVPR Workshop on Point Cloud Processing (PCP)*, 2012.
- [2] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit*. Kitware Inc., 4th edition, 2006.

---

see vtkPCLNormalEstimation.cxx

---

```

...
// create new normals array
vtkSmartPointer<vtkFloatArray> normals = vtkSmartPointer<vtkFloatArray>::New();
normals->SetNumberOfComponents(3);
normals->SetNumberOfTuples(input->GetNumberOfPoints());
normals->SetName("normals");

// pass input thru to output and add new array
output->ShallowCopy(input);
output->GetPointData()->AddArray(normals);

pcl::PointCloud<pcl::PointXYZ>::Ptr cloud = vtkPCLConversions::PointCloudFromPolyData(input);
pcl::PointCloud<pcl::PointXYZ>::Ptr searchCloud;
if (searchCloudPolyData)
{
    searchCloud = vtkPCLConversions::PointCloudFromPolyData(searchCloudPolyData);
}

pcl::PointCloud<pcl::Normal>::Ptr cloudNormals = ComputeNormalEstimation(cloud,
    searchCloud, this->SearchRadius);

assert(cloudNormals);
assert(cloudNormals->size() == normals->GetNumberOfTuples());

for (size_t i = 0; i < cloudNormals->size(); ++i)
{
    normals->SetTuple(i, cloudNormals->points[i].normal);
}
...

```

---

**Listing 11:** RequestData routine of the `vtkPCLNormalEstimation` filter (Part II).