Filtering, Normal Estimation, Segmentation

June 4, 2013
Introduction

- irregular density (2.5D)
- occlusions
- massive amount of data
- noise
Modifying the point cloud or point attributes.
Modifying the point cloud or point attributes.

- Removing Points:
  - Conditional Removal
  - Radius/Statistical Outlier Removal
  - Color Filtering
  - Passthrough
Modifying the point cloud or point attributes.

- **Removing Points:**
  - Conditional Removal
  - Radius/Statistical Outlier Removal
  - Color Filtering
  - Passthrough

- **Downsampling:**
  - Voxelgrid Filter
  - approximate Voxelgrid filtering
Modifying the point cloud or point attributes.

- **Removing Points:**
  - Conditional Removal
  - Radius/Statistical Outlier Removal
  - Color Filtering
  - Passthrough

- **Downsampling:**
  - Voxelgrid Filter
  - approximate Voxelgrid filtering

- **Modifying Other Point Attributes:**
  - Contrast
  - Bilateral Filtering
All filters are derived from the `Filter` base class with following interface:

```cpp
template<typename PointT> class Filter : public PCLBase<PointT> {
public:
  Filter (bool extract_removed_indices = false);
  inline IndicesConstPtr const getRemovedIndices ();
  inline void setFilterFieldName (const std::string &field_name);
  inline std::string const getFilterFieldName ();
  inline void setFilterLimits (const double &limit_min,
                              const double &limit_max);
  inline void getFilterLimits (double &limit_min,
                               double &limit_max);
  inline void setFilterLimitsNegative (const bool limit_negative);
  inline bool getFilterLimitsNegative ();
  inline void filter (PointCloud &output);
};
```
Removes points where values of selected field are out of range.

```cpp
// point cloud instance for the result
PointCloudPtr thresholded (new PointCloud);

// create passthrough filter instance
pcl::PassThrough<PointT> pass_through;

// set input cloud
pass_through.setInputCloud (input);

// set fieldname we want to filter over
pass_through.setFilterFieldName ("z");

// set range for selected field to 1.0 - 1.5 meters
pass_through.setFilterLimits (1.0, 1.5);

// do filtering
pass_through.filter (*thresholded);
```
Example: Passthrough Filter

original pointcloud: robot1.pcd

passthrough on the z axis 1.0 m - 1.5m
Divides the space into discrete cells (voxels) and replaces all points within a voxel by their centroids.

```c++
// point cloud instance for the result
PointCloudPtr downsampled (new PointCloud);

// create passthrough filter instance
pcl::VoxelGrid<PointT> voxel_grid;

// set input cloud
voxel_grid.setInputCloud (input);

// set cell/voxel size to 0.1 meters in each dimension
voxel_grid.setLeafSize (0.1, 0.1, 0.1);

// do filtering
voxel_grid.filter (*downsampled);
```
VoxelGrid Filter

voxelgrid with 0.1m voxel size in each dimension
Radius Outlier Removal

Removes all points with less than a given number of neighbors within a radius

```cpp
// point cloud instance for the result
PointCloudPtr cleaned (new PointCloud);

// create the radius outlier removal filter
pcl::RadiusOutlierRemoval<pcl::PointXYZRGB> radius_outlier_removal;

// set input cloud
radius_outlier_removal.setInputCloud (input);

// set radius for neighbor search
radius_outlier_removal.setRadiusSearch (0.05);

// set threshold for minimum required neighbors neighbors
radius_outlier_removal.setMinNeighborsInRadius (800);

// do filtering
radius_outlier_removal.filter (*cleaned);
```
Radius Outlier Removal
Radius Outlier Removal
How do we compute normals in practice?

- **Input:** point cloud $\mathcal{P}$ of 3D points $p = (x, y, z)^T$

- **Surface Normal Estimation:**
  1. Select a set of points $Q \subseteq \mathcal{P}$ from the neighborhood of $p$.
  2. Fit a local plane through $Q$.
  3. Compute the normal $\vec{n}$ of the plane.
Available Methods

- **Arbitrary Point Clouds:**
  - we can not make any assumptions about structure of the point cloud
  - use FLANN-based KdTree to find approx. nearest neighbors (pcl::NormalEstimation)

- **Organized Point Clouds:**
  - regular grid of points (width $w \times$ height $h$)
  - but, not all points in the regular grid have to be valid
  - we can use:
    - FLANN-based KdTree to find approx. nearest neighbors (pcl::NormalEstimation)
    - or faster: an Integral Image based approach (pcl::IntegralImageNormalEstimation)
Normal Estimation using pcl::NormalEstimation

```cpp
pcl::PointCloud<pcl::Normal>::Ptr normals_out
  (new pcl::PointCloud<pcl::Normal>);

pcl::NormalEstimation<pcl::PointXYZRGB, pcl::Normal> norm_est;

// Use a FLANN-based KdTree to perform neighborhood searches
norm_est.setSearchMethod
  (pcl::KdTreeFLANN<pcl::PointXYZRGB>::Ptr
   (new pcl::KdTreeFLANN<pcl::PointXYZRGB>));

// Specify the size of the local neighborhood to use when
// computing the surface normals
norm_est.setRadiusSearch (normal_radius);

// Set the input points
norm_est.setInputCloud (points);

// Set the search surface (i.e., the points that will be used
// when search for the input points' neighbors)
norm_est.setSearchSurface (points);

// Estimate the surface normals and store the result in "normals_out"
norm_est.compute (*normals_out);
```
Normal Estimation using pcl::NormalEstimation
Normal Estimation using `pcl::IntegralImageNormalEstimation`

```cpp
cpcl::PointCloud<pcl::Normal>::Ptr normals_out
    (new pcl::PointCloud<pcl::Normal>);

cpcl::IntegralImageNormalEstimation<pcl::PointXYZRGB, pcl::Normal> norm_est;

    // Specify method for normal estimation
    norm_est.setNormalEstimationMethod (ne.AVERAGE_3D_GRADIENT);

    // Specify max depth change factor
    norm_est.setMaxDepthChangeFactor(0.02f);

    // Specify smoothing area size
    norm_est.setNormalSmoothingSize(10.0f);

    // Set the input points
    norm_est.setInputCloud (points);

    // Estimate the surface normals and
    // store the result in "normals_out"
    norm_est.compute (*normals_out);
```
There are three ways of computing surface normals using integral images in PCL:

1. **COVARIANCE MATRIX**
   - Compute surface normal as eigenvector corresponding to smallest eigenvalue of covariance matrix
   - Needs 9 integral images

2. **AVERAGE_3D_GRADIENT**
   - Compute average horizontal and vertical 3D difference vectors between neighbors
   - Needs 6 integral images

3. **AVERAGEDEPTHCHANGE**
   - Compute horizontal and vertical 3D difference vectors from averaged neighbors
   - Needs 1 integral images
In presence of a dominant plane, we can find
- objects standing on tables or shelves
- protruding objects such as door handles

by
- computing the dominant plane of a scene
- segmenting the objects lying on (or detached from) this plane
One simple method is to perform segmentation via Euclidean clustering in the 3D space. If paired with RANSAC plane estimation, it can also be applied to the presence of a dominant plane on which to extract objects.
The basic idea is to use a region growing approach that cannot “grow” / connect two points with a high distance, therefore merging locally dense areas and splitting separate clusters.

```cpp
// Create EuclideanClusterExtraction and set parameters
pcl::EuclideanClusterExtraction<PointT> ec;
ec.setClusterTolerance (cluster_tolerance);
ec.setMinClusterSize (min_cluster_size);
ec.setMaxClusterSize (max_cluster_size);

// set input cloud and let it run
ec.setInputCloud (input);
ec.extract (cluster_indices_out);
```

`pcl::ConditionalEuclideanClustering` allows user-defined clustering strategies (e.g. on color space, etc..)
Recent method proposed by Alex Trevor et al. to detect multiple dominant planes

- for organized data (exploits the 2D lattice, real-time)
- based on detection of connected components among depth values
- extension to exploit color (full RGB-D)
Filtering, Normal Estimation, Segmentation

```cpp
pcl::OrganizedMultiPlaneSegmentation<PointT, pcl::Normal, pcl::Label> mps;
mps.setMinInliers(10000);
mps.setAngularThreshold (0.035); //2 degrees
mps.setDistanceThreshold(0.02);
mps.setInputNormals (normal_cloud);
mps.setInputCloud ( cloud);
std::vector<pcl::PlanarRegion<PointT> > regions;
mps.segmentAndRefine (regions);
```

Once planes are detected, objects can be found via Euclidean clustering. E.g., class `pcl::OrganizedConnectedComponentSegmentation` can follow using the extracted planes as a mask.
See plane segmentation tutorial at: 
http://www.pointclouds.org/documentation/tutorials/planar_segmentation.php

See normal estimation tutorials at: 
See projecting points using parametric model tutorial at:
http://www.pointclouds.org/documentation/tutorials/project_inliers.php

See convex/concave hull tutorial at:
http://www.pointclouds.org/documentation/tutorials/hull_2d.php

See euclidean clustering tutorial at:
http://www.pointclouds.org/documentation/tutorials/cluster_extraction.php