Semi-automatic point cloud segmentation

Sergey Alexandrov

PCL Tutorial – IAS 2014

pointcloudlibrary

- Completely automatic segmentation is hard
- In some applications it is acceptable to ask the user for hints
  - Number of segments
  - Seed locations
  - Segment contours
- Semi-automatic (“interactive”) segmentation algorithms leverage the hints to produce output matching with user expectations

Random Walker segmentation papers

- Random Walks for Image Segmentation

  Grady, L. (2006)

  IEEE Transactions on Pattern Analysis and Machine Intelligence

- Rapid and Effective Segmentation of 3D Models using Random Walks

  Lai, Y.-K., Hu, S.-M., Martin, R. R., & Rosin, P. (2009)

  Computer Aided Geometric Design

PCL implementation of Random Walker

```
#include <pcl/segmentation/random_walker_segmentation.h>

pcl::segmentation::RandomWalkerSegmentation<pcl::PointXYZ>
```

- Developed as a part of Toyota Code Sprint “Segmentation/Clustering of Objects in Cluttered Environments”
- The code is on its way to upstream PCL

TOYOTA

pcl::graph module

- New experimental `pcl::graph` module
  - Point cloud graph structure
  - Graph construction
  - Edge weight computation
  - Subgraph manipulation
  - Other miscellaneous functions

- Main design goal is seamless integration of
  - Boost.Graph data structures/algorithms
  - PCL data structures/algorithms

- Point cloud graph structure

  `pcl::graph::point_cloud_graph<pcl::PointXYZ>`

  - **Is a** Boost graph, so may be used as-is in Boost.Graph algorithms
  - Can be **viewed as** a PCL point cloud, so may be used in PCL algorithms

Point cloud graph structure usage

```
// Create a point cloud with 10 points
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::PointXYZ> (10, 1));
// Create a graph based on the cloud
pcl::graph::point_cloud_graph<pcl::PointXYZ> graph (cloud);
// The graph will have a vertex for each point of the original cloud
assert (10 == boost::num_vertices (graph));
// The points may be accessed using operator[]
graph[1].x = 14;
// The graph shares data with the original point cloud, so modifying a bundled point
// changes the corresponding point in the original cloud
assert (14 == cloud->points[1].x);
```

Point cloud graph structure usage

```
// Compute connected components using Boost.Graph algorithm on point cloud graph

std::vector<int> component (boost::num_vertices (graph));

size_t num_components = boost::connected_components (graph, &component[0]);

// Retrieve the bundled data as a point cloud

pcl::PointCloud<pcl::PointXYZ>::Ptr data = pcl::graph::point_cloud (graph);

// Continue to work with the data

pcl::io::savePCDFile ("output.pcd", *data);
```

- Graph construction from a point cloud
  - Create vertex set
  - Create edge set
- `pcl::graph::VoxelGridGraphBuilder<GraphT>`
  - Downsample
  - Use voxel neighborhood structure to establish edge set
- `pcl::graph::NearestNeighborsGraphBuilder<GraphT>`
  - Use fixed number of nearest neighbors to establish edge set

Edge weight computation

- Edge weight usually depends on the data in end vertices
- `pcl::graph::EdgeWeightComputer<GraphT>`
  - Iterates over edge set
  - Computes weights using per-configured function
- Weighting function has a fixed form: product of independent terms
  - XYZ (Euclidean distance)
  - Normal (Angular distance)
  - Curvature (curvature product)
  - RGB (Euclidean distance in RGB space)

Edge weight computation

- Contributions of terms are combined using an arbitrary balancing function
- Terms may be normalized
  - Globally
  - Locally
- Terms may be disabled when an edge is convex

Egde weight computation example

```
using namespace pcl::graph;

// Create edge weight computer

EdgeWeightComputer<Graph> computer;

// Append terms

computer.addTerm<terms::XYZ> (3.0f);

computer.addTerm<terms::Normal> (0.01ff);

computer.addTerm<terms::Curvature> (0.0001f);

// Compute edge weights

computer.compute (graph);
```

Random Walker segmentation

```
pcl::segmentation::RandomWalkerSegmentation<pcl::PointXYZ>
```

- Idea:
  - Each vertex in the graph emits a random walker
  - Edge weights represent probabilities of hop
  - Seeds are “destinations”
  - A vertex is assigned to the label of most probable destination
- Algorithm:
  - Construct graph Laplacian
  - Build and solve a system of linear equations

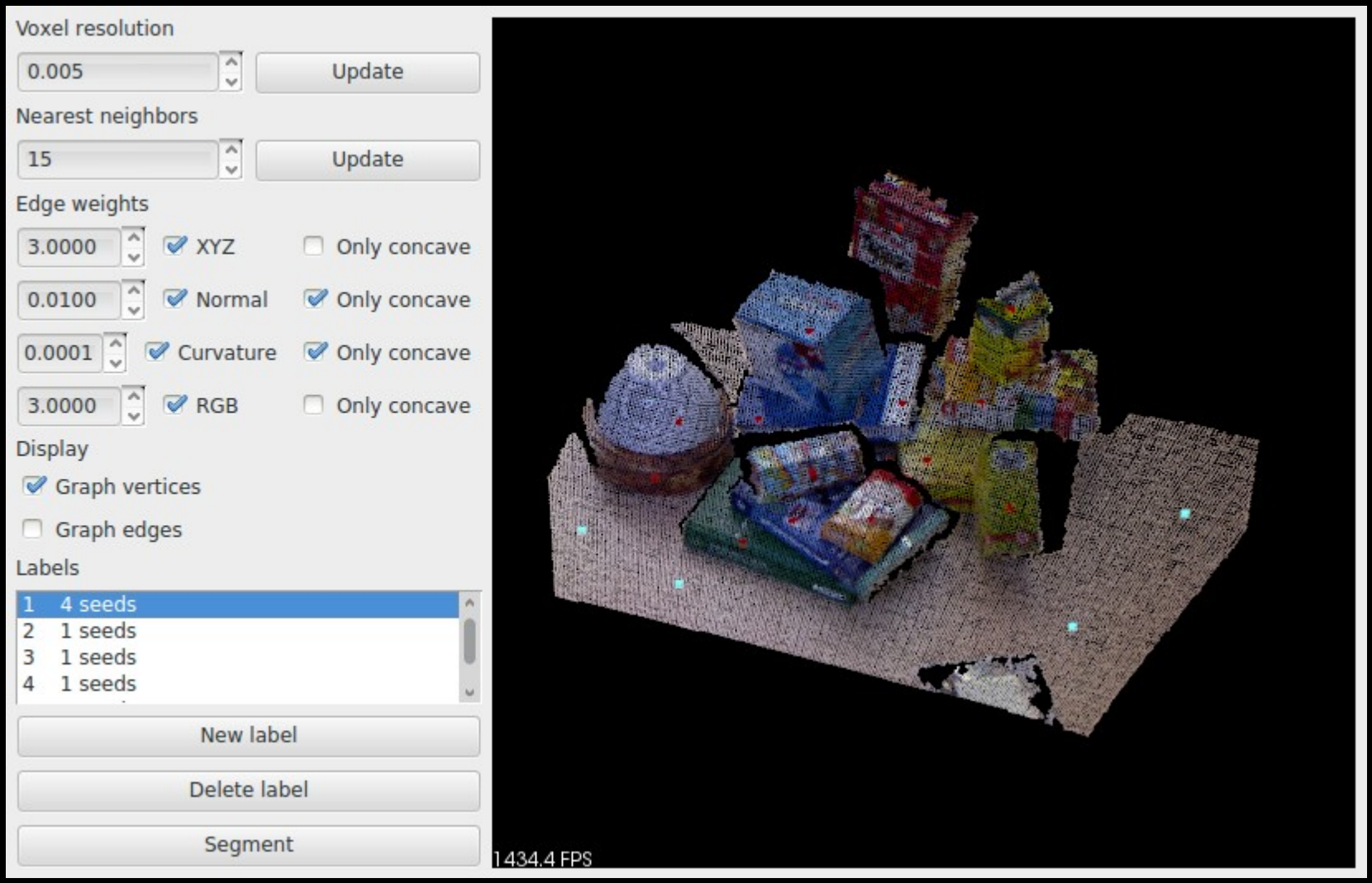
- Properties of Random Walker algorithm
  - Robust with respect to
    - Noise
    - Weak boundaries
  - Handles arbitrary number of segments
  - Fast
  - Avoids trivial solutions

- Basic usage
  - User supplies point cloud and seeds
  - Graph is created transparently for the user
    - Default parameters are fit for Kinect-style point clouds
- Advanced usage
  - User supplies point cloud graph and seeds

Random Walker basic usage example

```
// Create random walker segmentation object
RandomWalkerSegmentation<pcl::PointXYZRGB> rws;
rws.setInputCloud (cloud);
rws.preComputeGraph ();
auto graph = rws.getGraph ();
// Visualize graph or graph point cloud
// Ask the user to select seeds
// ...
rws.setSeeds (seeds);
rws.segment (clusters);
```

Questions?


Voxel resolution
0.005
Update
Nearest neighbors
15
Update
Edge weights
3.0000
XYZ
Only concave
0.0100
Normal
Only concave
0.0001
Curvature
Only concave
3.0000
RGB
Only concave
Display
Graph vertices
Graph edges
Labels
1 4 seeds
2 1 seeds
3 1 seeds
4 1 seeds
New label
Delete label
Segment
1434.4 FPS