SAPIENZA
UNIVERSITÀ DI ROMA

Elective in A.I. - Robot Programming

2014/2015 - Prof: Daniele Nardi

Perception with RGB-D sensors – Jacopo Serafin

Point Cloud Library (PCL), Surface Normals, Generalized ICP

Contact

Jacopo Serafin

Ph.D. Student in Engineering in Computer Science

Department of Computer, Control and Management Engineering "Antonio Ruberti", Sapienza University of Rome, Via Ariosto 25, 00185 Rome, Italy

Room B120 | Ro.Co.Co. Laboratory

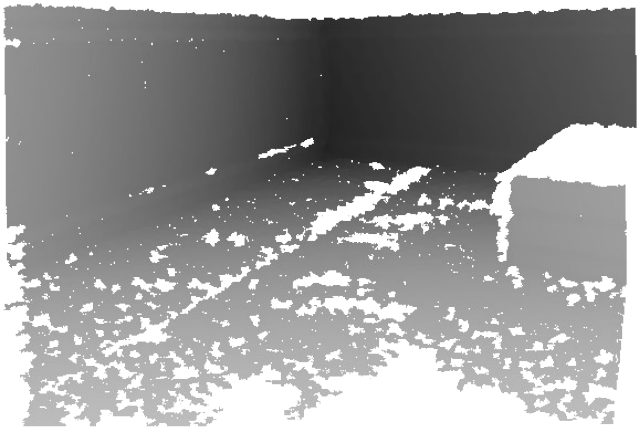
Email: serafin@dis.uniroma1.it

Phone: +39-06-77274157

Web Page: http://www.dis.uniroma1.it/~serafin

Organized Point Cloud

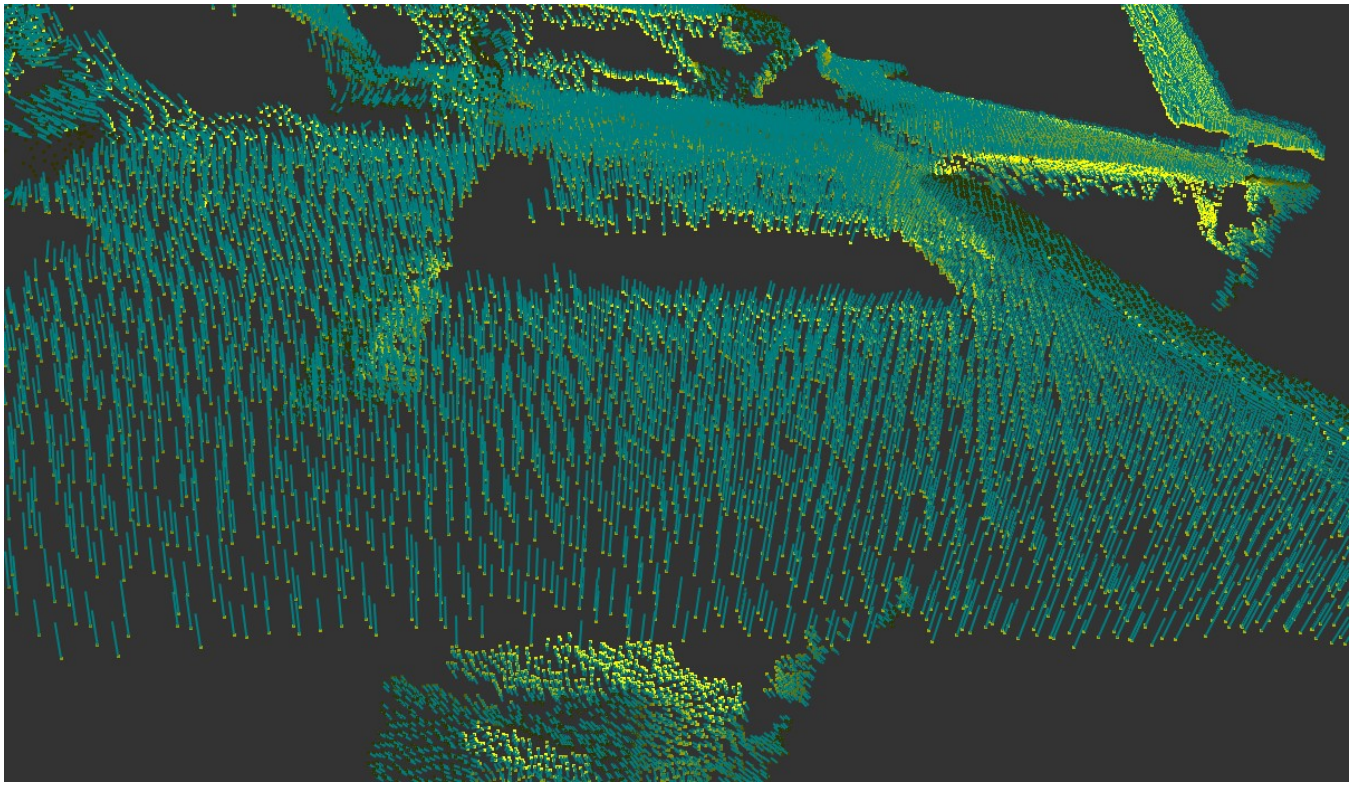
- Goal: maintain same structure (rows and columns) of the depth image
- Necessary for many algorithms that suppose to have an organized point cloud
- Depth images contain not valid values (e.g. zero pixels), thus the cloud is not dense



```
/* Image-like organized structure,
with 640 rows and 480 columns */

cloud.width = 640;

/* thus 640*480=307200 points total
in the dataset */

cloud.height = 480;
```

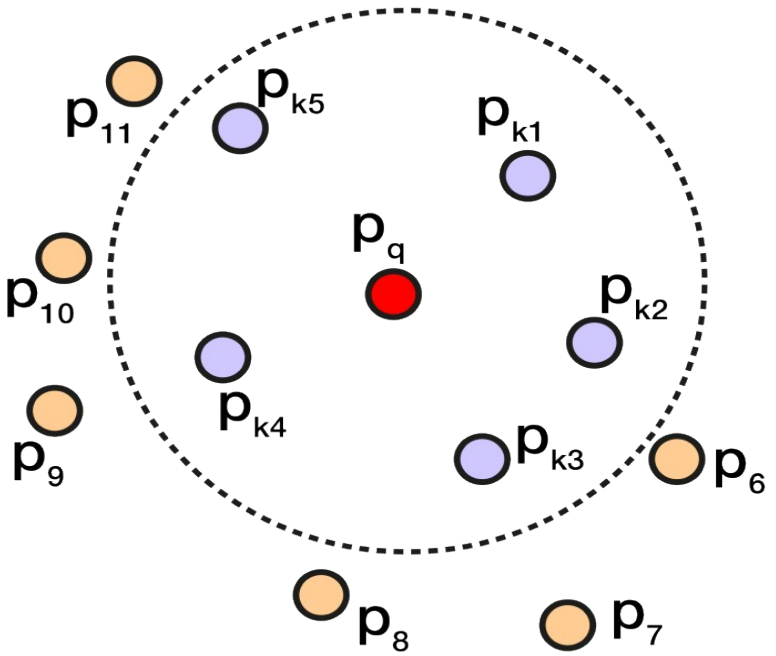
Surface Normal

- Vector normal to the surface where the point lies
- Provide additional information about the structure around the point



How to Compute the Surface Normals

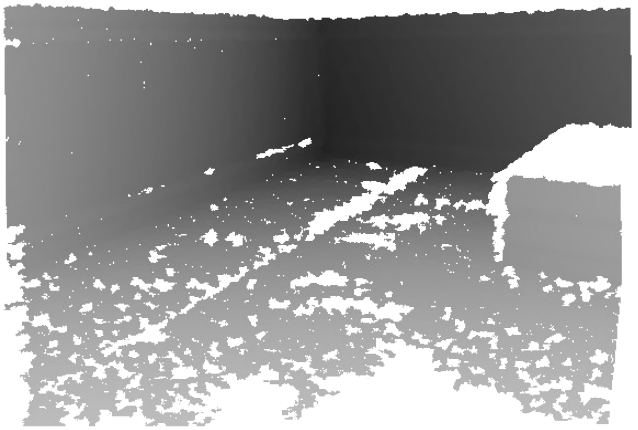
- Approximate the neighborhood of a point with a surface:
  - Compute the covariance matrix
  - Compute singular value decomposition
  - Take the normal as the eigenvector associated to the smallest eigenvalue

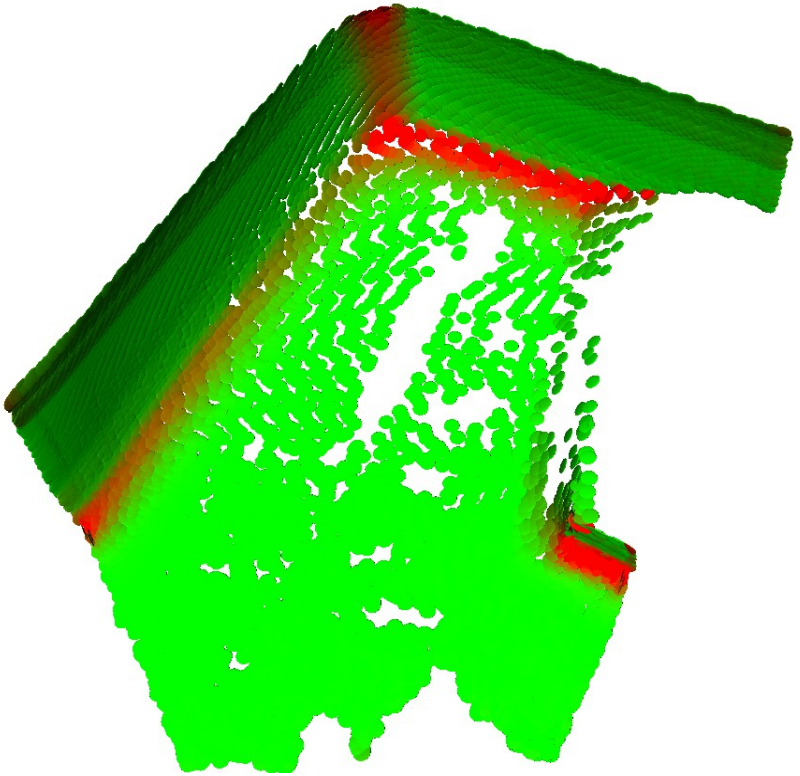


Curvature

- Can be computed as a function of the eigenvalues of the covariance matrix:

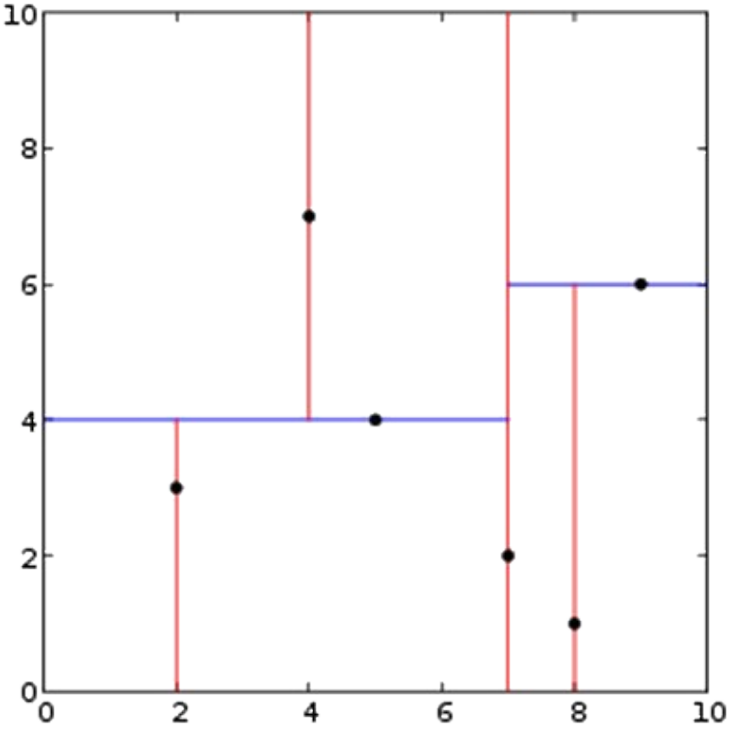
$$\sigma = \lambda_0 / (\lambda_0 + \lambda_1 + \lambda_2) \in [0,1]$$





KdTree

- Data structure to organize points in a space with k dimensions
- Very useful for range and nearest neighbor searches
- Cost for search one nearest neighbor is equal to O(log n)



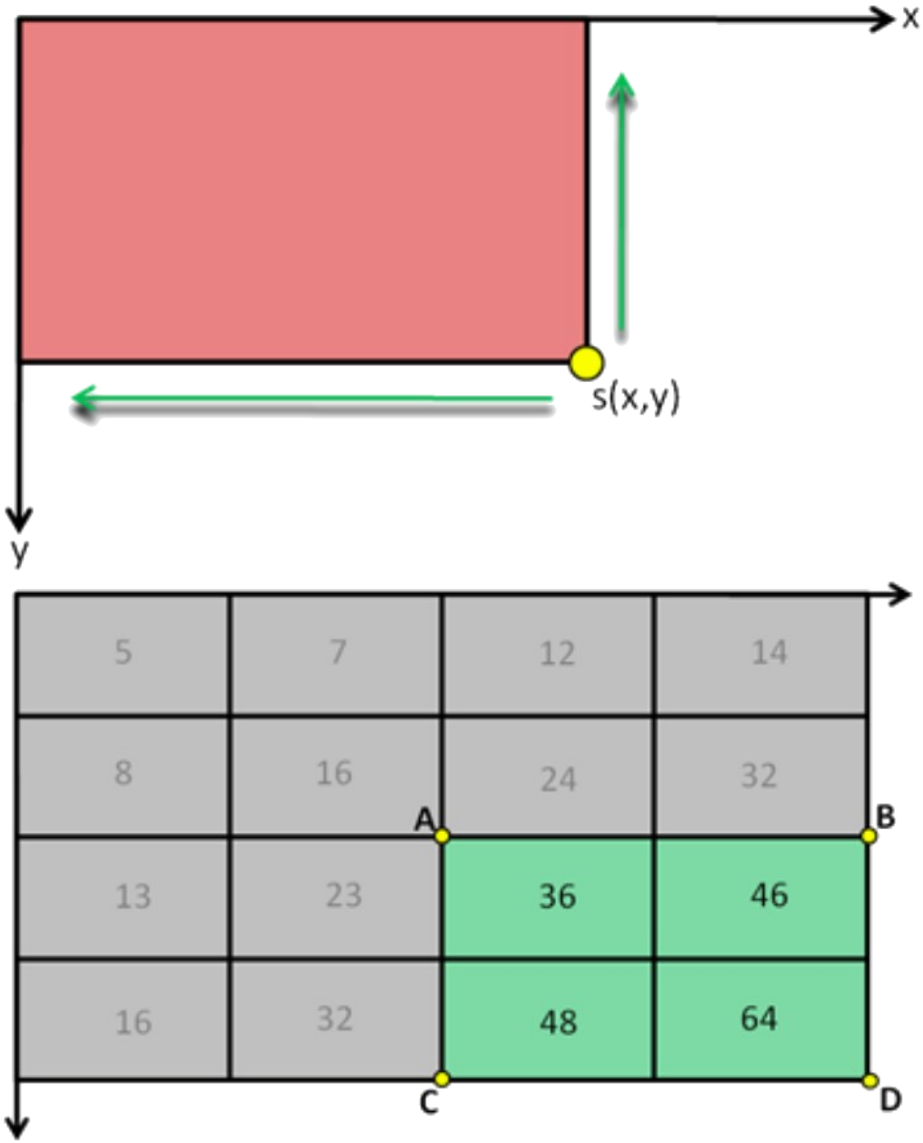

```
pcl::KdTreeFLANN<pcl::PointXYZ> kdtree;

kdtree.setInputCloud (cloud);

// K nearest neighbor search
kdtree.nearestKSearch (searchPoint, K,
                       pointIdxNKNSearch,
                       pointNKNSquaredDistance);

// Neighbors within radius search
kdtree.radiusSearch (searchPoint, radius,
                     pointIdxRadiusSearch,
                     pointRadiusSquaredDistance);
```

Surface Normals Computation: KdTree Based Code

```
// Create the normal estimation class, and give it the input dataset
pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> ne;
ne.setInputCloud (cloud);

// Create an empty kdtree, and pass it to the normal estimation object
pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new
pcl::search::KdTree<pcl::PointXYZ> ());
ne.setSearchMethod (tree);

// Output datasets
pcl::PointCloud<pcl::Normal>::Ptr cloud_normals (new
pcl::PointCloud<pcl::Normal>);

// Use all neighbors in a sphere of radius 3cm
ne.setRadiusSearch (0.03);

// Compute the features
ne.compute (*cloud_normals);
```

Integral Image

- It is a particular image where each pixel contains the sum of the pixels of the upper left part of the image
- Allows fast computation of the surface normals
- Once the integral image the cost for a surface normal computation is constant O(1)
- The main drawback is a loss of precision in the neighbors computation
- s(A, B, C, D) = s(A) + s(D) – s(B) - s(C)

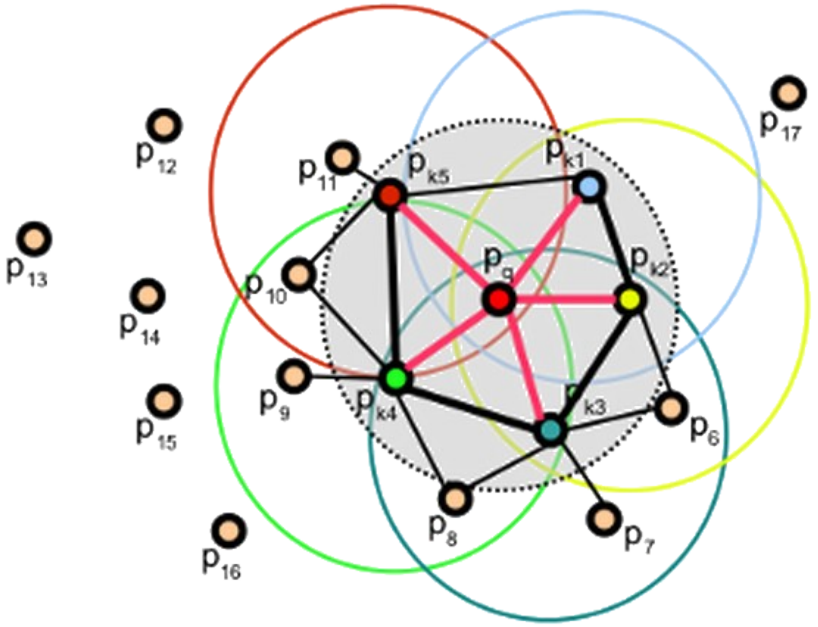


Surface Normals Computation: Integral Images Based Code

```
// estimate normals
pcl::PointCloud<pcl::Normal>::Ptr normals (new
pcl::PointCloud<pcl::Normal>);

// create a surface normals integral image estimator object
pcl::IntegralImageNormalEstimation<pcl::PointXYZ, pcl::Normal> ne;

// set some parameters
ne.setNormalEstimationMethod (ne.AVERAGE_3D_GRADIENT);
ne.setMaxDepthChangeFactor(0.02f);
ne.setInputCloud(cloud);

// compute the surface normals
ne.compute(*normals);
```

Fast Point Features Histograms (FPFH)

- Encode the point's k-neighborhood geometrical properties
- Fast computation allowing real-time execution

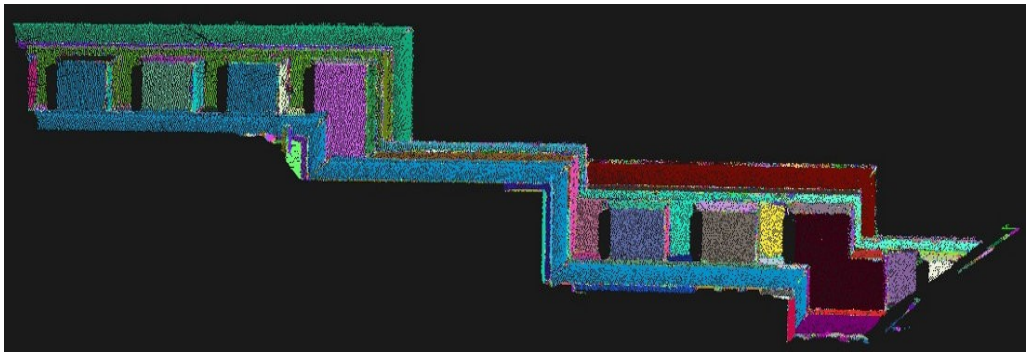


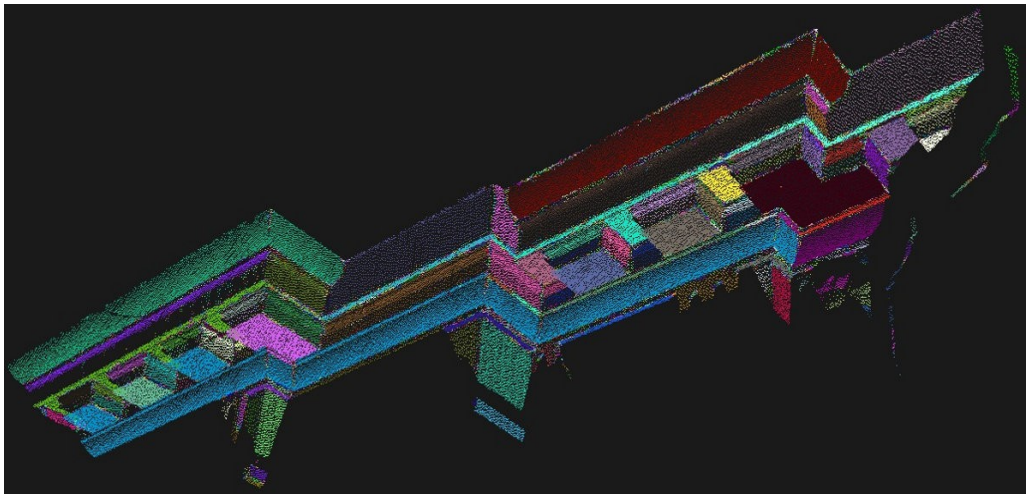
Fast Point Features Histograms (FPFH): Code

```
// Create the FPFH estimation class, and pass the input dataset + normals to it
pcl::FPFHEstimation<pcl::PointXYZ, pcl::Normal, pcl::FPFHSignature33> fpfh;
fpfh.setInputCloud (cloud);
fpfh.setInputNormals (normals);

// alternatively, if cloud is of tpe PointNormal, do fpfh.setInputNormals (cloud);
// Create an empty kdtree representation, and pass it to the FPFH estimation object.
// Its content will be filled inside the object, based on the given input dataset (as
// no other search surface is given).
pcl::search::KdTree<PointXYZ>::Ptr tree (new pcl::search::KdTree<PointXYZ>);

fpfh.setSearchMethod (tree);

// Output datasets
pcl::PointCloud<pcl::FPFHSignature33>::Ptr fpfhs (new
pcl::PointCloud<pcl::FPFHSignature33> ());

// Use all neighbors in a sphere of radius 5cm
// IMPORTANT: the radius used here has to be larger than the radius used to estimate
// the surface normals!!!
fpfh.setRadiusSearch (0.05);

// Compute the features
fpfh.compute (*fpfhs);
```

Region Growing Segmentation Based on Surface Normals

- Merge the points that are close enough in terms of smoothness constraints
- Until all the points not in a region are parsed:
  - Select a point as seed
  - Check recursively if its neighbors satisfy the constraints (angle difference between the surface normals):
    - If at least one, or more neighbors are good, add them to the region
    - If not, generate a new seed and create a new region



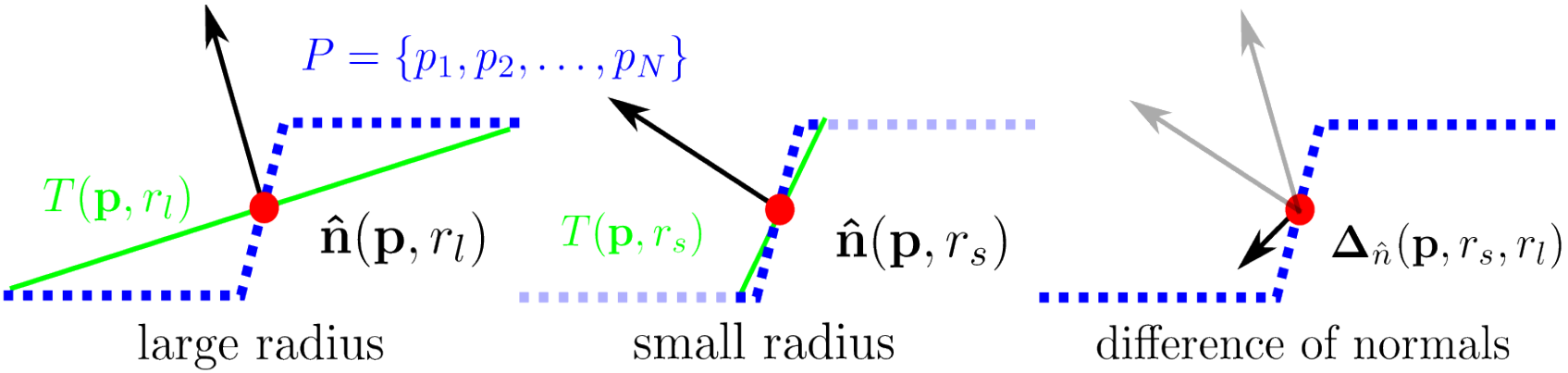


Region Growing Segmentation Based on Surface Normals: Code

```
// create the object that implements the surface normals region growing algorithm
pcl::RegionGrowing<pcl::PointXYZ, pcl::Normal> reg;

// Set some parameters
reg.setMinClusterSize (50);
reg.setMaxClusterSize (1000000);
reg.setSearchMethod (tree); // e.g. KdTree
reg.setNumberOfNeighbours (30);
reg.setInputCloud (cloud);
reg.setInputNormals (normals);
reg.setSmoothnessThreshold (3.0 / 180.0 * M_PI);
reg.setCurvatureThreshold (1.0);

// Perform the segmentation
std::vector <pcl::PointIndices> clusters;
reg.extract (clusters);

// Print some information
std::cout << "Number of clusters is equal to " << clusters.size () << std::endl;
std::cout << "First cluster has " << clusters[0].indices.size () << " points." << endl;
```

Difference of Normals for Segmentation

- Estimate the normals for every point using a large support radius r_l
- Estimate the normals for every point using a small support radius r_s
- Compute the normalized difference of normals for every point, as shown in the image
- Filter the resulting vector field to isolate points belonging to the scale/region of interest.

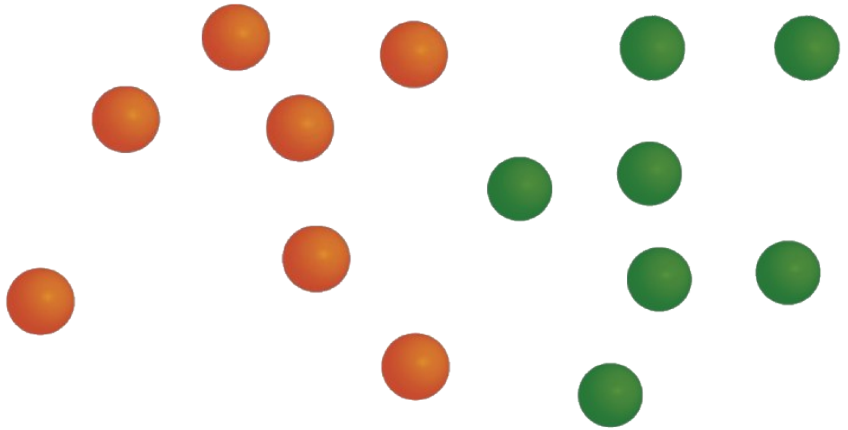


Difference of Normals for Segmentation: Code

```
// Create Difference of Normal operator
pcl::DifferenceOfNormalsEstimation<PointXYZRGB, PointNormal, PointNormal> don;

don.setInputCloud (cloud);
don.setNormalScaleLarge (normals_large_radius);
don.setNormalScaleSmall (normals_small_radius);

// Check possible failures
if (!don.initCompute ())
{
     std::cerr << "Error: Could not intialize DoN feature operator" << std::endl;
     exit (EXIT_FAILURE);
}

// Compute Difference of Normals
don.computeFeature (*doncloud);

// Filter by magnitude
...
```

Point Cloud Registration

We want to find the translation and the rotation that maximize the overlap between two point clouds



Point Cloud Registration

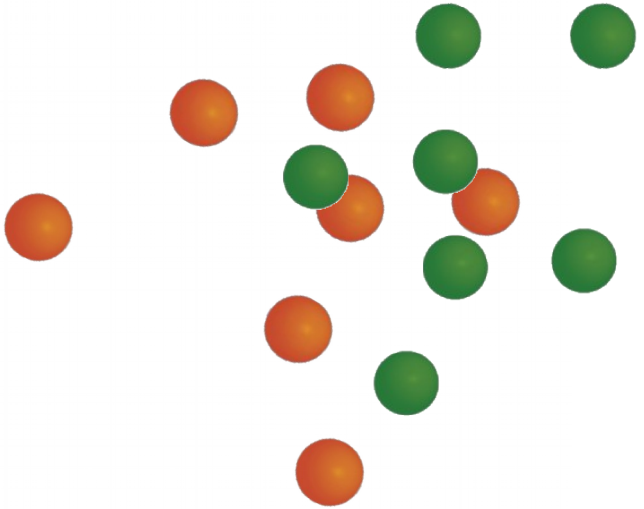
We want to find the translation and the rotation that maximize the overlap between two point clouds

Point Cloud Registration

We want to find the translation and the rotation that maximize the overlap between two point clouds

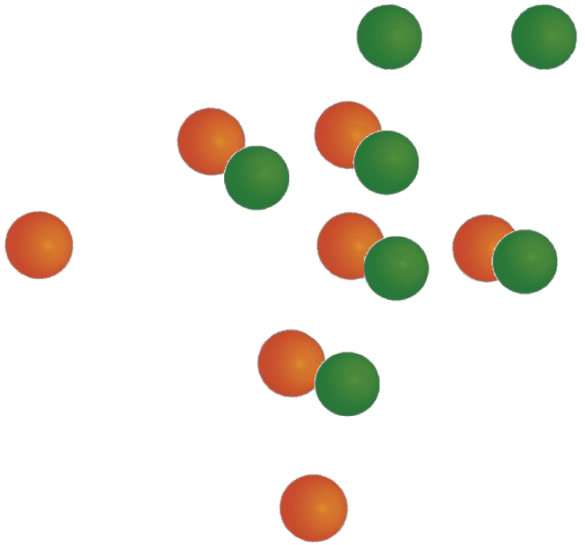
Point Cloud Registration

We want to find the translation and the rotation that maximize the overlap between two point clouds



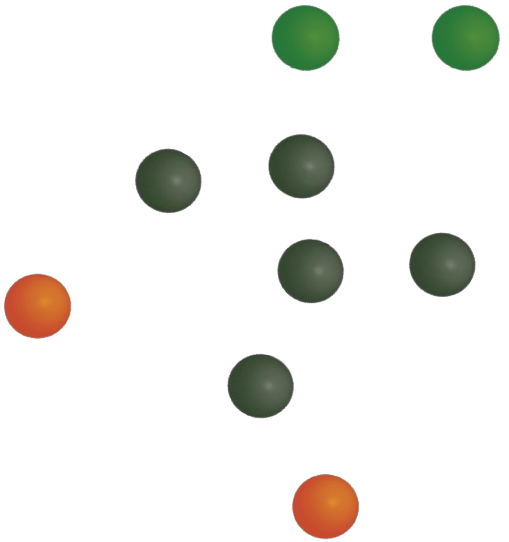
Point Cloud Registration

We want to find the translation and the rotation that maximize the overlap between two point clouds



Point Cloud Registration

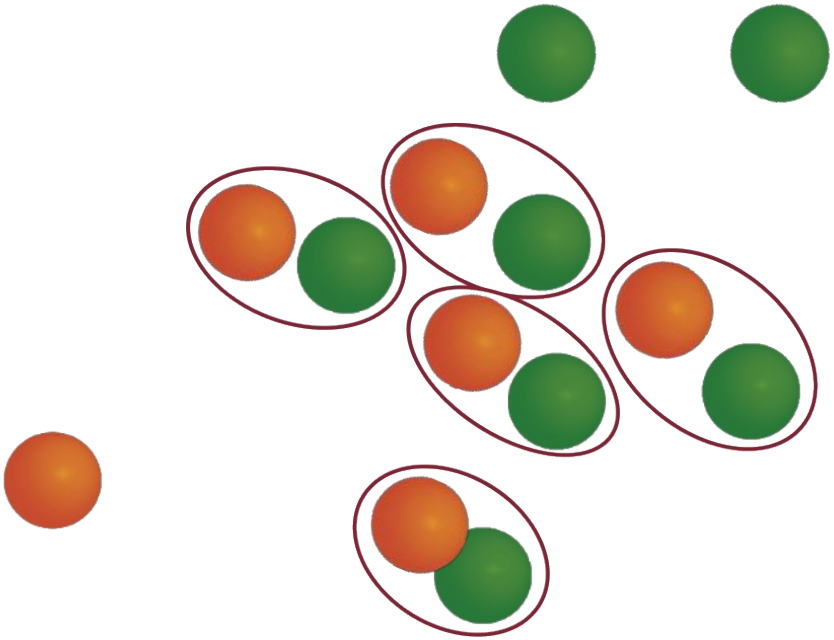
We want to find the translation and the rotation that maximize the overlap between two point clouds



Iterative Closest Point Algorithm

ICP iteratively refine an initial transformation **T** by alternating:

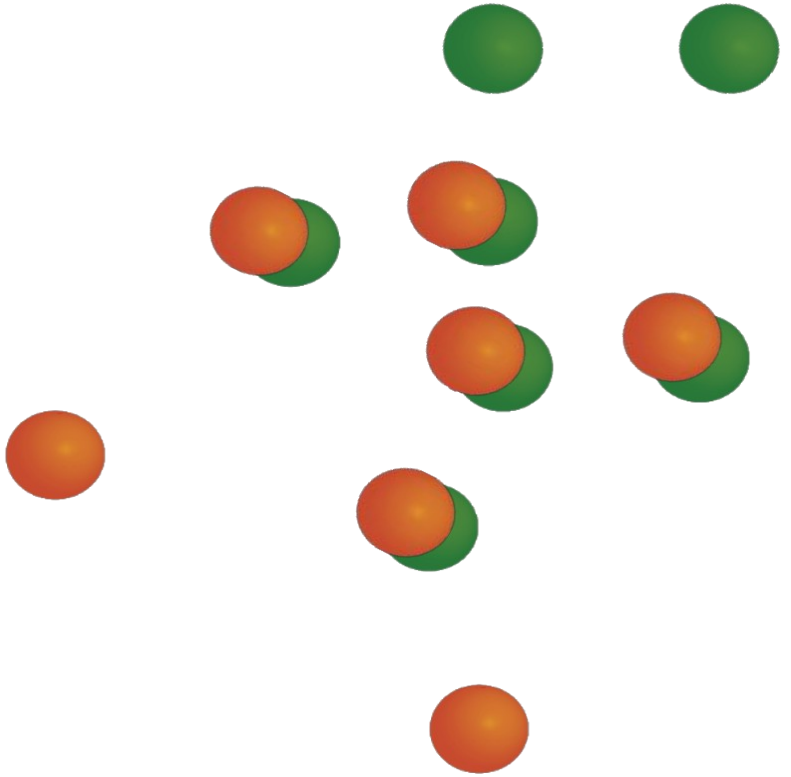
- **search of correspondences between the two point clouds…**
- and the optimization step to update the current transformation



Iterative Closest Point Algorithm

ICP iteratively refine an initial transformation **T** by alternating:

- search of correspondences between the two point clouds…
- **and the optimization step to update the current transformation**

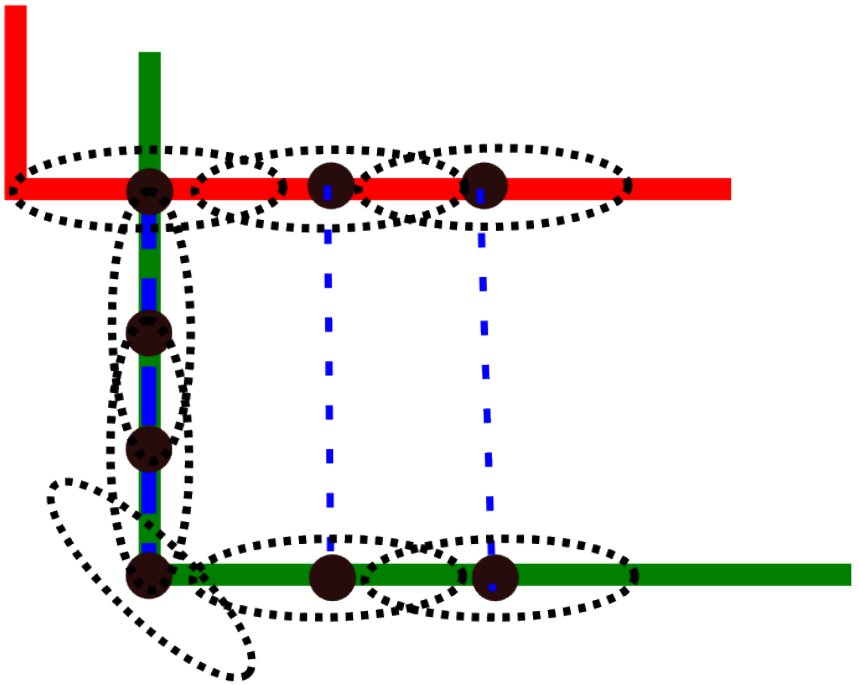


Point Cloud Registration: ICP Based Code

```
// create the object implementing ICP algorithm
pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ> icp;

// set the input point cloud to align
icp.setInputCloud(cloud_in);
// set the input reference point cloud
icp.setInputTarget(cloud_out);

// compte the point cloud registration
pcl::PointCloud<pcl::PointXYZ> Final;
icp.align(Final);

// print fitness score
std::cout << "has converged:" << icp.hasConverged()
          << " score: "
          << icp.getFitnessScore() << std::endl;

// print the output transformation
std::cout << icp.getFinalTransformation() << std::endl;
```

Generalized ICP (GICP)

- Variant of ICP
- Assumes that points are sampled from a locally continuous and smooth surfaces
- Since two points are not the same it is better to align patches of surfaces instead of the points



Point Cloud Registration: GICP Based Code

```
// create the object implementing ICP algorithm
pcl::GeneralizedIterativeClosestPoint<pcl::PointXYZRGBNormal,
                                      pcl::PointXYZRGBNormal> gicp;

// set the input point cloud to align
gicp.setInputCloud(cloud_in);
// set the input reference point cloud
gicp.setInputTarget(cloud_out);

// compte the point cloud registration
pcl::PointCloud<pcl::PointXYZRGBNormal> Final;
gicp.align(Final);

// print if it the algorithm converged and its fitness score
std::cout << "has converged:" << gicp.hasConverged()
          << " score: "
          << gicp.getFitnessScore() << std::endl;

// print the output transformation
std::cout << gicp.getFinalTransformation() << std::endl;
```

Homework 1/3

Read a sequence of ordered pairs of images (RGB + Depth images) and save the associated point cloud with colors and surface normals on .pcd files (e.g. cloud_005.pcd)

- Download one of the datasets (e.g. desk_1.tar) at :

  http://rgbd-dataset.cs.washington.edu/dataset/rgbd-scenes/

Homework 2/3

After, for each file .pcd read sequentially:

- Align the current point cloud with the previous one by using Generalized ICP
- Save the cloud with its global transformation (either transforming directly the cloud or using the `sensor_origin` and `sensor_orientation` parameter provided in the point cloud object)

Homework 3/3

Apply a voxelization to the totoal point cloud (necessary to reduce the dimension in terms of bytes) and visualize it so that the entire scene reconstructed is shown

Hints

- Warning: the depth images are stored with 16 bit depth, so in this case calling the `cv::imread()` function you should specify the flag `CV_LOAD_IMAGE_ANYDEPTH` (or `-1`)
- WARNING: the input depth image should be scaled by a 0.001 factor in order to obtain distances in meters. You could use the opencv function:

```
input_depth_img.convertTo(scaled_depth_img, CV_32F, 0.001);
```

- As camera matrix, use the following default matrix:

```
float fx = 512, fy = 512, cx = 320, cy = 240;

Eigen::Matrix3f camera_matrix;

camera_matrix << fx, 0.0f, cx, 0.0f, fy, cy, 0.0f, 0.0f, 1.0f;
```

- As re-projection matrix, use the following matrix:

```
Eigen::Matrix4f t_mat;t_mat.setIdentity();

t_mat.block<3, 3>(0, 0) = camera_matrix.inverse();
```

Hints

- For each pixel (x,y) with depth d, obtain the corresponding 3D point as:

  ```
  Eigen::Vector4f point = t_mat * Eigen::Vector4f(x*d, y*d, d, 1.0);
  ```
  (the last coordinate of point can be ignored)

- WARNING: Since We are working with organized point clouds, also points with depth equal to 0 that are not valid, should be added to the computed cloud as `NaN`, i.e. in pseudocode:

  ```
  const float bad_point = std::numeric_limits<float>::quiet_NaN();
  if( depth(x, y) == 0) { p.x = p.y = p.z = bad_point; }
  ```

- To get the global transform of the current cloud just perform the following multiplication after you computed the registration:

  ```
  Eigen::Matrix4f globalTransform = previousGlobalTransform *
  alignmentTransform;
  ```

  `previousGlobalTransform` is the global transformation found for the previous point cloud

  `alignmentTransform` is the local transform computed using Generalized ICP

- WARNING: the first global transform has to be initialized to the identity matrix