Introduction to PCL: Point Cloud Library

SAPIENZA
UNIVERSITÀ DI ROMA

Roberto Capobianco

Thanks to Alberto Pretto, Radu Bogdan Rusu, Bastian Steder and Jeff Delmerico for some of the slides!

Dipartimento di Ingegneria Informatica, Automatica e Gestionale

Point clouds

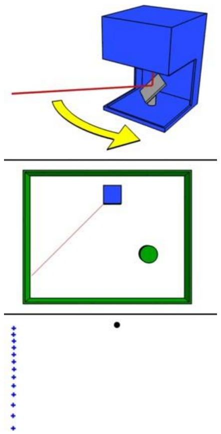
- collection of **multi-dimensional points**
- commonly used to represent three-dimensional data.

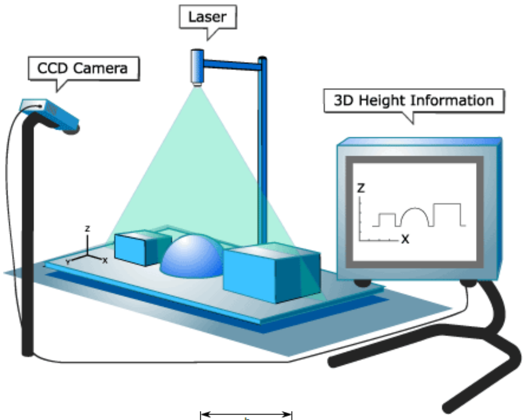
  the points usually represent X, Y, Z geometric coordinates of a sampled surface.

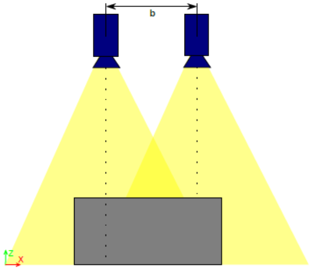
Each point can hold additional information: RGB colors, intensity values, etc.

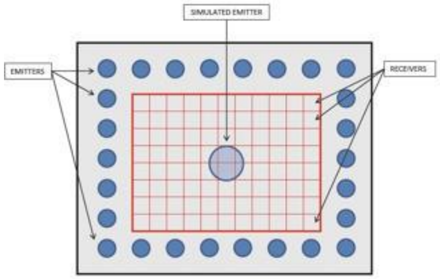
Where do they come from?

- 2/3D Laser scans
- Laser triangulation
- Stereo cameras
- RGB-D cameras
- Structured light cameras
- Time of flight cameras



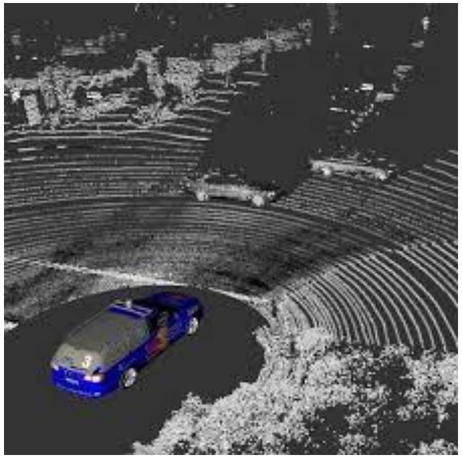


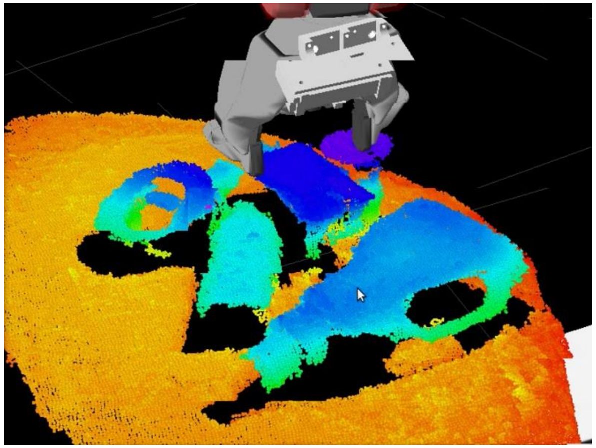




Point clouds in robotics

- Navigation / Obstacle avoidance
- Object recognition and registration
- Grasping and manipulation





Point Cloud Library

pointclouds.org

The Point Cloud Library (PCL) is a standalone, large scale, open source (C++) library 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.

PCL provides the 3D processing pipeline for ROS, so you can also get the perception pcl stack and still use PCL standalone. Among others, PCL depends on Boost, Eigen, OpenMP, ...

PCL Basic Structures: PointCloud

A PointCloud is a templated C++ class which basically contains the following data fields:

- **width (int)** - specifies the width of the point cloud dataset in the number of points.
  →the total number of points in the cloud (equal with the number of elements in points) for *unorganized* datasets
  →the width (total number of points in a row) of an *organized* point cloud dataset
- **height (int)** - Specifies the height of the point cloud dataset in the number of points
  →set to 1 for unorganized point clouds
  →the height (total number of rows) of an organized point cloud dataset
- **points (std::vector<PointT>)** - Contains the data array where all the points of type PointT are stored.

PointCloud vs. PointCloud2

We distinguish between two data formats for the point clouds:

- **PointCloud<PointType>** with a specific data type (for actual usage in the code)
- **PointCloud2** as a general representation containing a header defining the point cloud structure (e.g., for loading, saving or sending as a ROS message)

Conversion between the two frameworks is easy:

- **pcl::fromROSMsg** and **pcl::toROSMsg**

Important: clouds are often handled using smart pointers, e.g.:

- PointCloud<PointType>::Ptr cloud_ptr;

Point Types

PointXYZ - float x, y, z

PointXYZI - float x, y, z, intensity

PointXYZRGB - float x, y, z, rgb

PointXYZRGBA - float x, y, z, uint32 t rgba

Normal - float normal[3], curvature

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

→See pcl/include/pcl/point_types.h for more examples.

Building PCL Standalone Projects

```
# CMakeLists.txt
project(pcl_test)
cmake_minimum_required(VERSION 2.8)
cmake_policy(SET CMP0015 NEW)

find_package(PCL 1.8 REQUIRED)
add_definitions(${PCL_DEFINITIONS})

include_directories(... ${PCL_INCLUDE_DIRS})
link_directories(... ${PCL_LIBRARY_DIRS})

add_executable(pcl_test pcl_test.cpp ...)
target_link_libraries(
pcl_test${PCL_LIBRARIES})
```

PCL structure

PCL is a collection of smaller, modular C++ libraries:

- **libpcl_features**: many 3D features (e.g., normals and curvatures, boundary points, moment invariants, principal curvatures, Point Feature Histograms (PFH), Fast PFH, ...)
- **libpcl_surface**: surface reconstruction techniques (e.g., meshing, convex hulls, Moving Least Squares, ...)
- **libpcl_filters**: point cloud data filters (e.g., downsampling, outlier removal, indices extraction, projections, ...)
- **libpcl_io**: I/O operations (e.g., writing to/reading from PCD (Point Cloud Data) and BAG files)
- **libpcl_segmentation**: segmentation operations (e.g.,cluster extraction, Sample Consensus model fitting, polygonal prism extraction, ...)
- **libpcl_registration**: point cloud registration methods (e.g., Iterative Closest Point (ICP), non linear optimizations, ...)
- **libpcl_range_image**: range image class with specialized methods

It provides unit tests, examples, tutorials, ...

PointCloud File Format

Point clouds can be stored to disk as files, into the **PCD (Point Cloud Data) format**:

```
– # Point Cloud Data (PCD) file format v. 5
  FIELDS x y z rgba
  SIZE 4 4 4 4
  TYPE F F F U
  WIDTH 307200
  HEIGHT 1
  POINTS 307200
  DATA binary
  ...<data>...
```

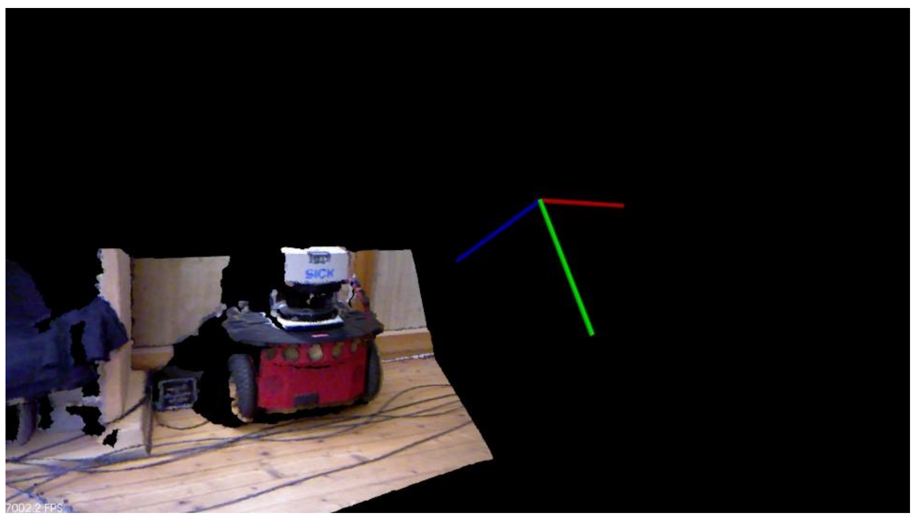
Funtions: **pcl::io::loadPCDFile** and **pcl::io::savePCDFile**

Example: create and save a PC

```
#include<pcl/io/pcd_io.h>
#include<pcl/point_types.h>

//....
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_ptr (new pcl::PointCloud<pcl::PointXYZ>);

cloud>width = 50;
cloud>height = 1;
cloud>isdense = false;
cloud>points.resize(cloud.width*cloud.height);

for(size_t i = 0; I < cloud.points.size(); ++i){
  cloud>points[i].x = 1024*rand()/(RANDMAX + 1.0f);
  cloud>points[i].y = 1024*rand()/(RANDMAX + 1.0f);
  cloud>points[i].z = 1024*rand()/(RANDMAX + 1.0f);
}

pcl::io::savePCDFileASCII("testpcd.pcd", *cloud);
```

Visualize a Cloud

```
boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer(new
                                    pcl::visualization::PCLVisualizer("3D Viewer"));
viewer>setBackgroundColor(0, 0, 0);  viewer-
>addPointCloud<pcl::PointXYZ>(in_cloud, cloud_color, "Input cloud");
viewer>initCameraParameters();
viewer→addCoordinateSystem(1.0);

while (!viewer>wasStopped()) viewer>spinOnce(1);
```



Basic Module Interface

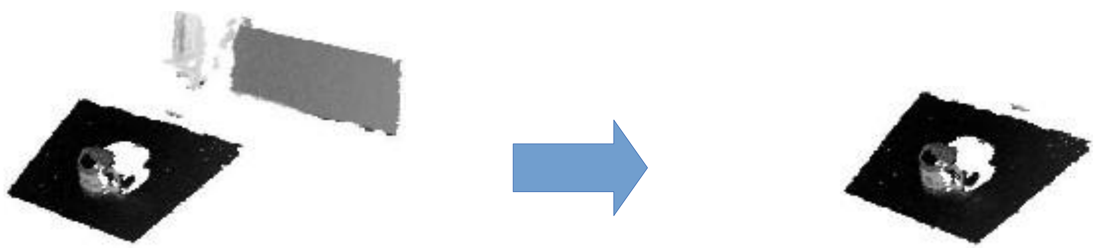
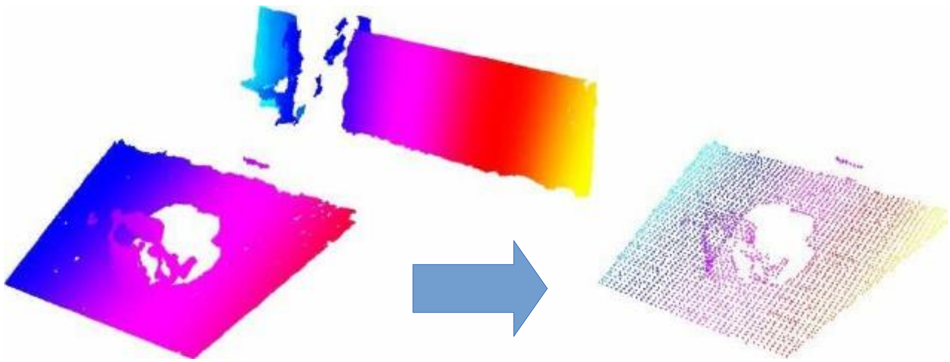
Filters, Features, Segmentation all use the same basic usage interface:

- use **setInputCloud()** to give the input
- set some parameters
- call **compute()** or **filter()** or **align()** or **...** to get the output

PassThrough Filter

Filter out points outside a specified range in one dimension.

- ```
  pcl::PassThrough<T> pass_through;
  pass_through.setInputCloud(in_cloud);
  pass_through.setFilterLimits(0.0, 0.5);
  pass_through.setFilterFieldName("z");
  pass_through.filter(*cut_cloud);
  ```


  ```

Downsampling

Voxelize the cloud to a 3D grid. Each occupied voxel is approximated by the centroid of the points inside it.

- ```
  pcl::VoxelGrid<T> voxel_grid;
  voxel_grid.setInputCloud(input_cloud);
  voxel_grid.setLeafSize(0.01, 0.01, 0.01);
  voxel_grid.filter(*subsamp_cloud );
  ```


  ```

Feature Example: Normals

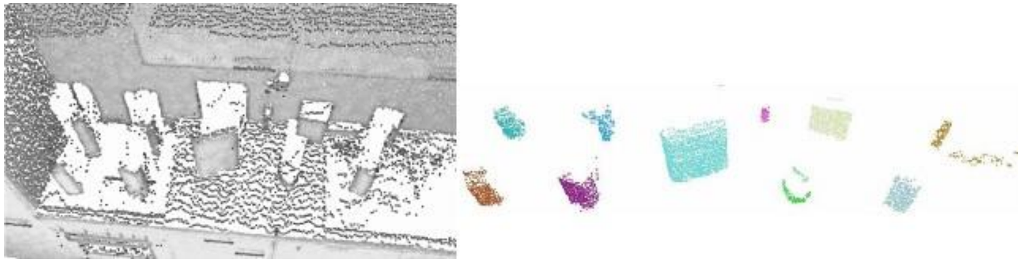
```
pcl::NormalEstimation<T, pcl::Normal> ne;
ne.setInputCloud(in_cloud);
pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new
              pcl::search::KdTree<pcl::PointXYZ>());
ne.setSearchMethod(tree);
ne.setRadiusSearch(0.03);
ne.compute(*cloud_normals);
```

Segmentation Example

A clustering method divides an unorganized point cloud into smaller, correlated, parts.

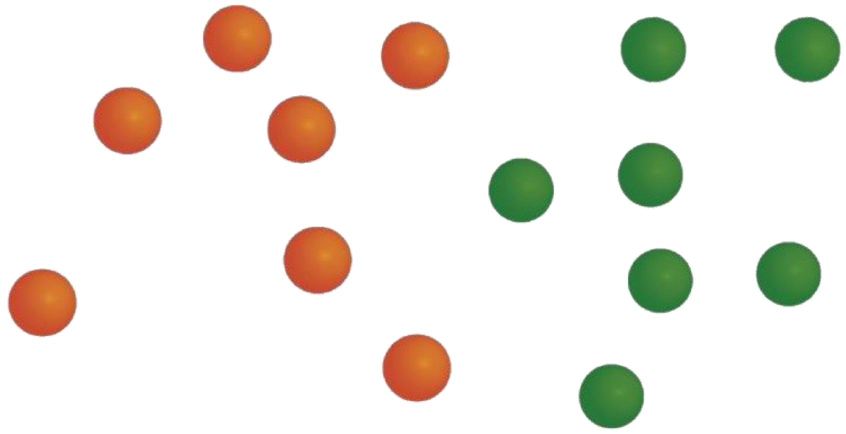
EuclideanClusterExtraction uses a distance threshold to the nearest neighbors of each point to decide if the two points belong to the same cluster.

```
pcl::EuclideanClusterExtraction<T> ec;
ec.setInputCloud(in_cloud);
ec.setMinClusterSize(100);
ec.setClusterTolerance(0.05); // distance threshold
ec.extract(cluster_indices);
```



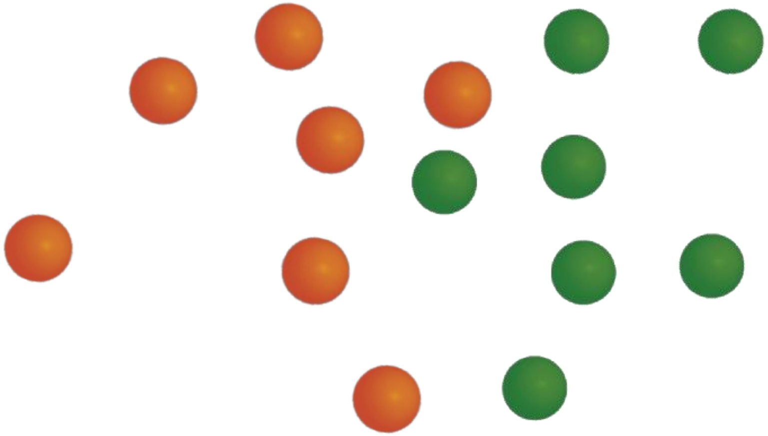
Point Cloud Registration

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

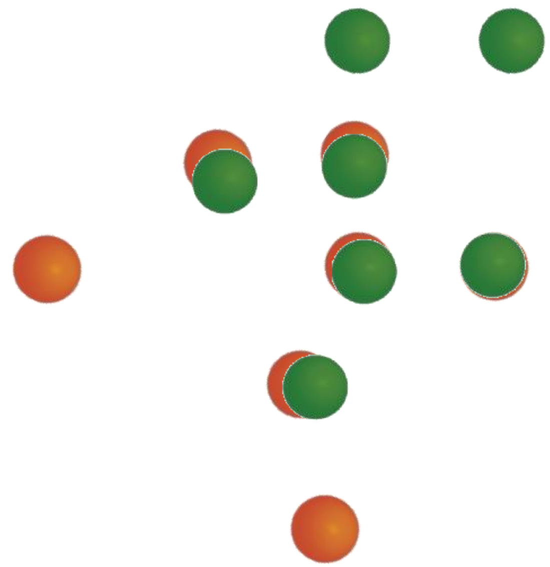
We want to find the translation and the rotation that maximize the overlap between two point clouds

Point Cloud Registration

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

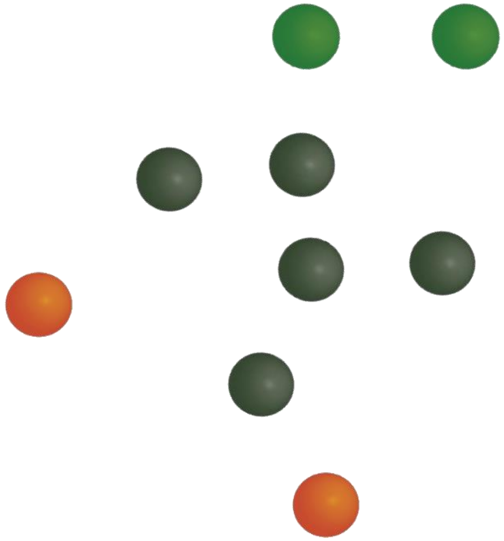
Point Cloud Registration

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



Point Cloud Registration

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



Iterative Closest Point - 1

ICP iteratively revises the transformation (translation,rotation) needed to minimize the distance between the points of two raw scans.

- **Inputs**: points from two raw scans, initial estimation of the transformation, criteria for stopping the iteration.
- **Output**: refined transformation.

Iterative Closest Point - 2

The algorithm steps are :

1) associate points of the two cloud using the nearest neighbor criteria;
2) estimate transformation parameters using a mean square cost function;
3) transform the points using the estimated parameters;
4) iterate (re-associate the points and so on);

Iterative Closest Point - 3

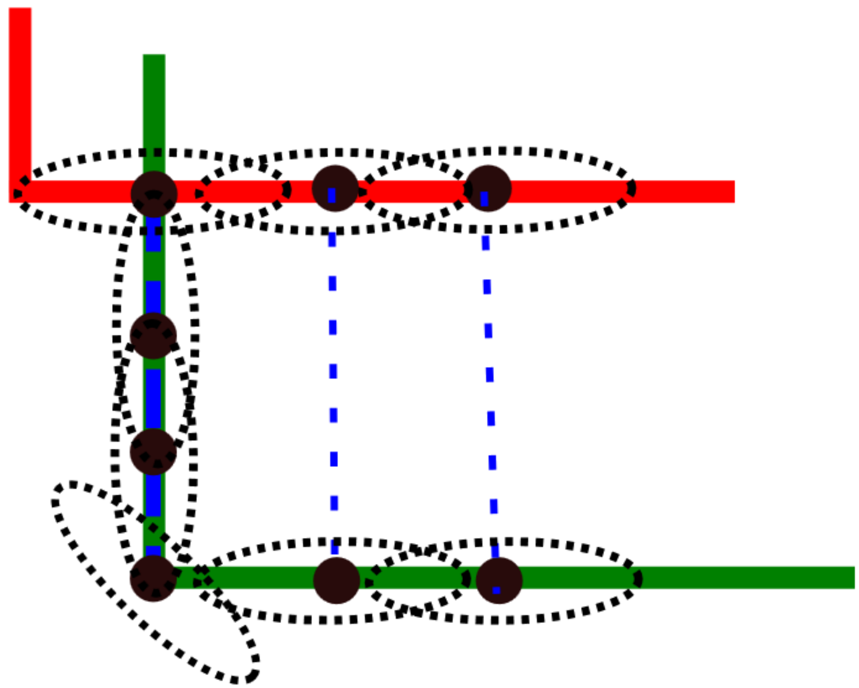
```
IterativeClosestPoint<PointXYZ, PointXYZ> icp;
// Set the input source and target
icp.setInputCloud(cloud_source);
icp.setInputTarget(cloud_target);
// Set the max correspondence distance to 5cm
icp.setMaxCorrespondenceDistance(0.05);
// Set the maximum number of iterations (criterion 1)
icp.setMaximumIterations(50);
// Set the transformation epsilon (criterion 2)
icp.setTransformationEpsilon(1e8);
// Set the euclidean distance difference epsilon (criterion 3)
icp.setEuclideanFitnessEpsilon(1);
// Perform the alignment
icp.align(cloud_source_registered);
// Obtain the transformation that aligned cloud_source to
cloud_source_registered
Eigen::Matrix4f transformation = icp.getFinalTransformation ();
```

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



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

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 individual scene datasets at :

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

Each student should use a different dataset

Homework

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

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

Apply an additional filter of your choice to the point cloud

Homework

Acquire a dataset (using rosbag) using a Kinect/Xtion (in lab) of your face for both depth and RGB. Alternatively use the ROSbag at

http://www.dis.uniroma1.it/~bloisi/didattica/RobotProgramming/face.bag

Visualize the bag in 3D in the PCL viewer

http://wiki.ros.org/ROS/Tutorials/Recording%20and%20playing%20back%20data

Kinect: http://wiki.ros.org/freenect_stack

Hints (1/3)

Warning:thedepthimagesarestoredwith16bitdepth,sointhiscasecallingthe `cv::imread()` functionyoushouldspecifytheflagcv:IMREAD_ANYDEPTH :
cv:Matinput_depth=cv:imread("test_depth.png",cv:IMREAD_ANYDEPTH);

WARNING:theinputdepthimageshouldbescaledbya0.0 1factorinorderto obtaindistancesinmeters.Youcouldusetheopencvfunction:

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

Ascameramatrix,usethefolowingdefaultmatrix:

```
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;
```

Asre-projectionmatrix,usethefollowingmatrix:

```
Eigen::Matrix4f t_mat;
t_mat.setIdentity();
t_mat.block<3, 3>(0, 0) = camera_matrix.inverse();
```

Hints (2/3)

Foreachpixel(x,y)withdepthd,obtainthecoresponding3D pointas:

```
Eigen::Vector4f point = t_mat * Eigen::Vector4f(x*d, y*d, d, 1.0);
```

(thelastcoordinateofpointcanbeignored)

WARNING:SinceWeareworkingwithorganizedpointclouds,alsopointswithdepthequalto 0thatarenotvalid,shouldbead edtothecomputedcloudas`NaN`,i.e.inpseudocode:

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

Togettheglobaltransform ofthecurentcloudjustperform thefolowingmultiplication afteryoucomputedtheregistration:

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

`previousGlobalTransform` istheglobaltransformationfoundfortheprevious point cloud

`alignmentTransform` isthelocaltransform computedusingGeneralizedICP

WARNING:thefirstglobaltransform hastobeinitializedtotheidentitymatrix

Hints (3/3)

Kinecttopics:

"/camera/depth_registered/image_rect_raw"

"/camera/rgb/image_rect"

Topic subscription, synchronization and callback registration

```
#include <message_filters/subscriber.h>
#include
<message_filters/synchronizer.h>
#include <message_filters/sync_policies/approximate_time.h>
ros::NodeHandle nh;
message_filters::Subscriber<Image> depth_sub(nh, "topic1", 1);
message_filters::Subscriber<Image> rgb_sub(nh, "topic2", 1);
typedef sync_policies::ApproximateTime<Image, Image> syncPolicy;
Synchronizer<syncPolicy> sync(syncPolicy(10), depth_sub, rgb_sub);
sync.registerCallback(boost::bind(&callback, _1, _2));
```