UNIVERSITÀ DEGLI STUDI DI PADOVA

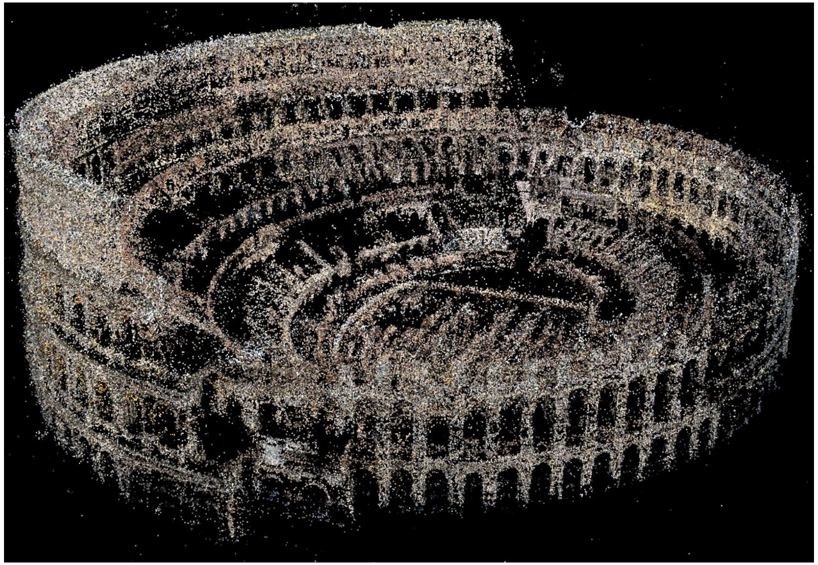
3D Data Processing

Introduction to PCL and Open3D
Basic Topics

Alberto Pretto

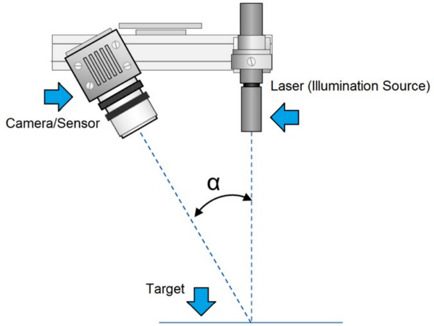
Point Cloud: a Definition

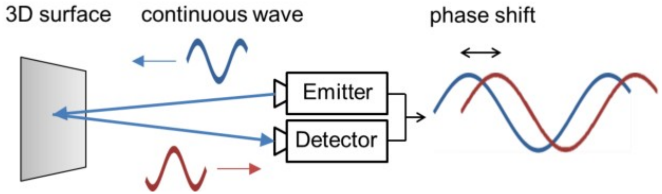
- A point cloud is a data structure used to represent a collection of multi-dimensional points and is commonly used to represent three-dimensional data.
- The points usually represent the X, Y, and Z geometric coordinates of a sampled surface.
- Each point can hold additional information: RGB colors, intensity values, etc...

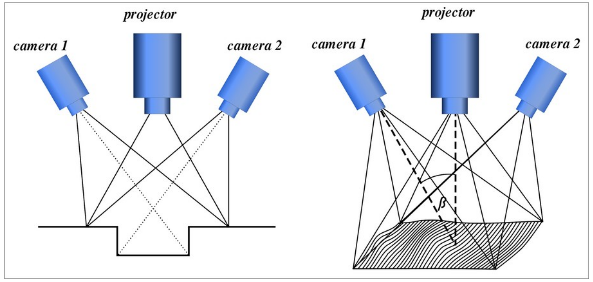


Where Do They Come From?

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







Point 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.
- Among others, PCL depends on Boost, Eigen, OpenMP,...

PCL PointCloud

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

- **width (int)**
  - 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)**
  - 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.

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.

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, ...

Point Cloud 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>...
```

- Functions to load/save point clouds:
  - pcl::io::loadPCDFile
  - 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);
```

Example: Visualize a PC

```
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

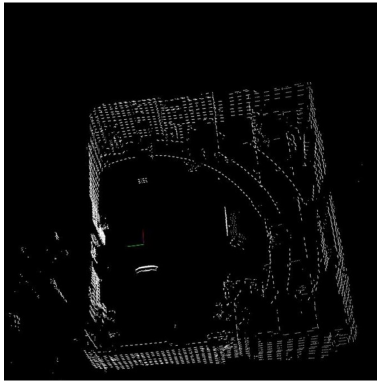
- Filters, Features, Segmentation all use the same basic usage interface:
- use **setInputCloud()** to give the input
- set some parameters
- call **compute()** or **filter()** or **align()** or ... to get the

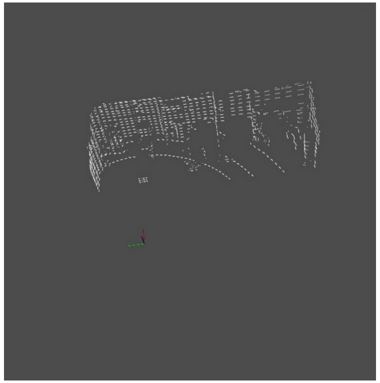
  output

PassThrough Filter

Filter out points outside a specified range in one dimension

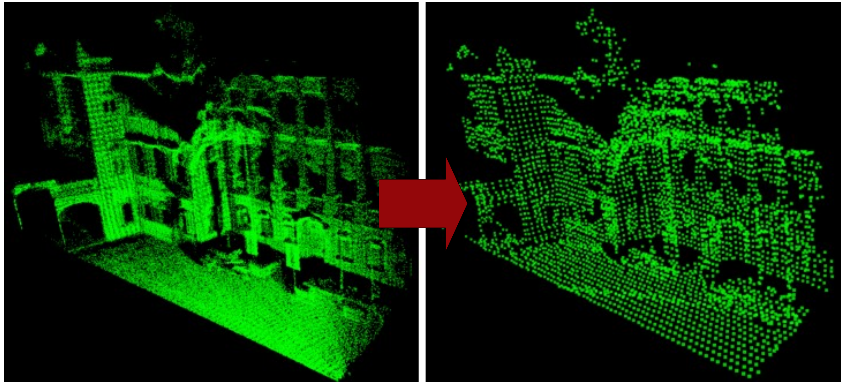
```
pcl::PassThrough<T> pass_through;
pass_through.setInputCloud (in_cloud);
pass_through.setFilterLimits (0.0, 0.5);
pass_through.setFilterFieldName ("z");
pass_through.filter( *cutted_cloud );
```





Downsampling

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



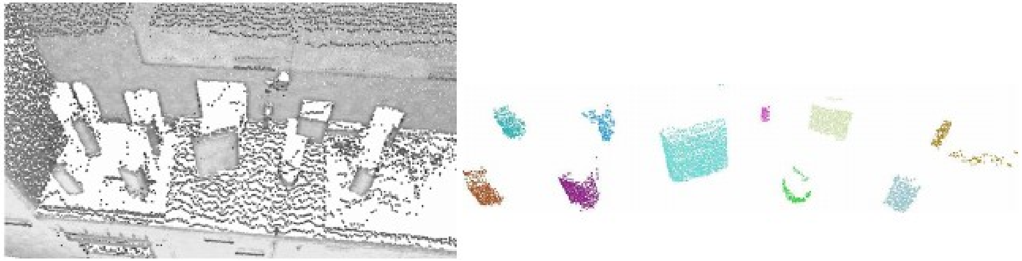
```
pcl::VoxelGrid<T> voxel_grid;
voxel_grid.setInputCloud (input_cloud);
voxel_grid.setLeafSize (0.01, 0.01, 0.01);
voxel_grid.filter ( *subsamp_cloud ) ;
```

Compute 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

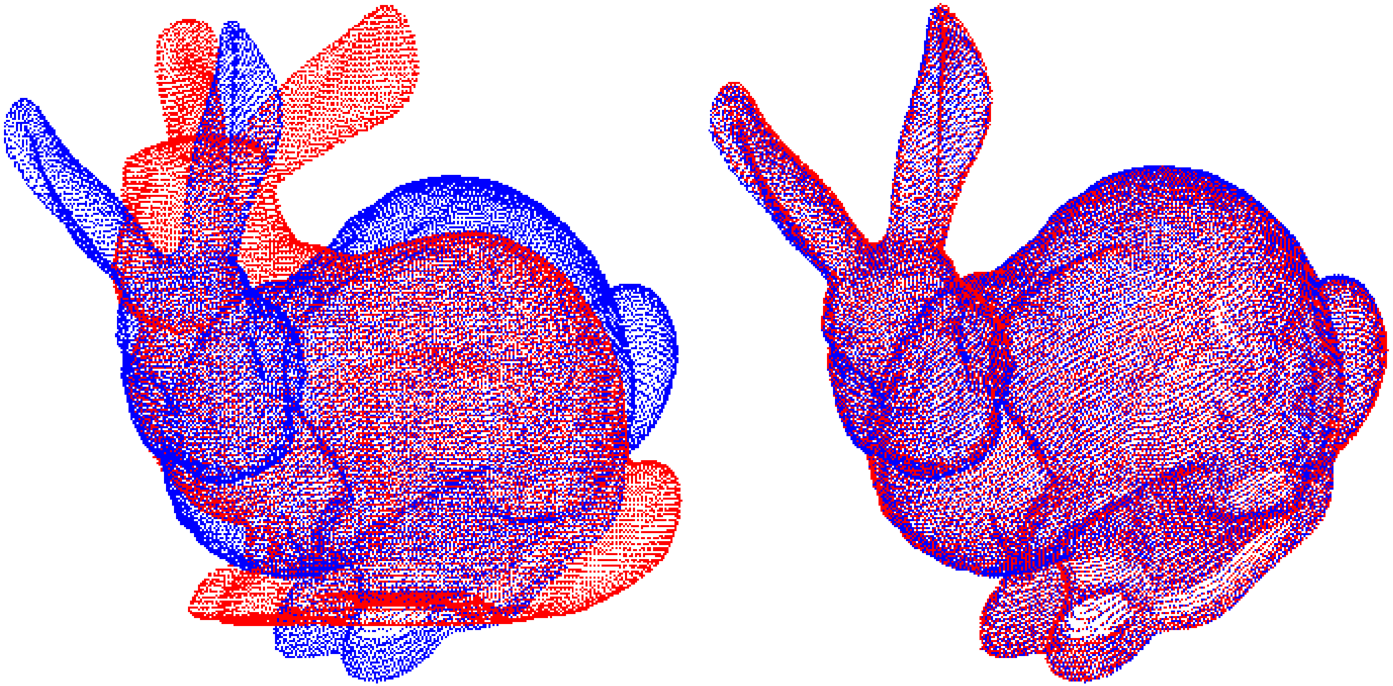
- A clustering method divides an unorganized point cloud into smaller, correlated, parts.
- **EuclideanClusterExtraction** uses a distance threshold to the nearest neighbors of each point to decide if the two points belong to the same cluster.



```
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



Iterative Closest Point

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

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

```
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 (1e-8);
// 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 ();
```

Open3D

open3d.org

- Open3D is an open-source library that supports rapid development of software that deals with 3D data. The Open3D frontend exposes a set of carefully selected data structures and algorithms in both C++ and Python.
- Open3D is released under the MIT license. Its use is encouraged for both research and commercial purposes, as long as proper attribution is given

Open3D PointCloud

A Open3D PointCloud contains the following data fields:

- **points_** (std::vector< Eigen::Vector3d >) - Contains the 3D coordinates of the points
- **colors_** (std::vector< Eigen::Vector3d >) - Contains the RGB color for each point in points_
- **normals_** (std::vector< Eigen::Vector3d >) - Contains the surface normal for each point in points_

Open3D Modules

Open3D functions are gathered in different modules, some of them are:

- **geometry**: contains the class definition of point cloud and mesh structures
- **camera**: to store camera intrinsics and extrinsics parameters
- **visualization**: functions useful to display 3D structures
- **io**: I/O operations (e.g., writing to/reading from PLY, PCD,...)

Load and Visualize Multiple PC

```
open3d::geometry::PointCloud pc0;
open3d::geometry::PointCloud pc1;

open3d::io::ReadPointCloud(“path_to_pc0”, pc0);
open3d::io::ReadPointCloud(“path_to_pc1”, pc1);

auto pc0_pointer =
std::make_shared<open3d::geometry::PointCloud>(pc0);
auto pc1_pointer =
std::make_shared<open3d::geometry::PointCloud>(pc1);

open3d::visualization::DrawGeometries({pc0_pointer, pc1_pointer});
```

Downsample, Normals Estimation

```
//Downsample

double voxel_size = 0.007

std::shared_ptr<open3d::geometry::PointCloud>
&pcd_down_ptr = pcd.VoxelDownSample(voxel_size);


//Estimate point cloud normals

pcd_down_ptr->EstimateNormals(
  open3d::geometry::KDTreeSearchParamHybrid(
                          voxel_size*2, 30));
```

ICP Registration

```
Eigen::Matrix4d transformation = Eigen::Matrix4d::Identity();
double threshold = 0.02;
double relative_fitness=1e-6;
double relative_rmse1e-6;
int max_iteration=1000;


auto result =
  open3d::pipelines::registration::RegistrationICP(
     source, target, threshold, transformation,
     open3d::pipelines::registration::
       TransformationEstimationPointToPoint(),
     open3d::pipelines::registration::
       ICPConvergenceCriteria(relative_fitness,
                              relative_rmse, max_iteration)
  );
```