UNIVERSITÀ di VERONA
Dipartimento di INFORMATICA

Corso di Laboratorio Ciberfisico
Modulo di Robot Programming with ROS

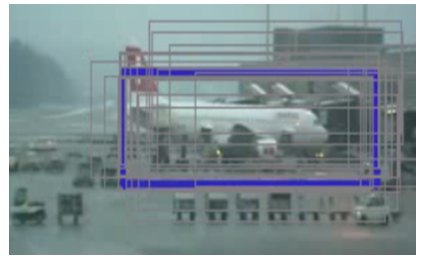
Acquisizione e gestione dati 3D

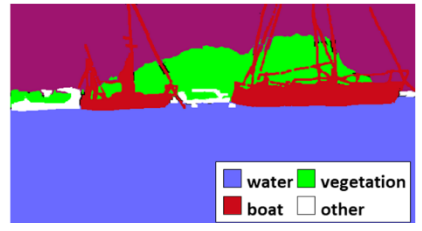
pcl

Docente:
Domenico Daniele Bloisi

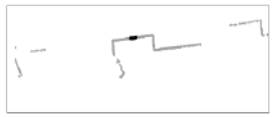
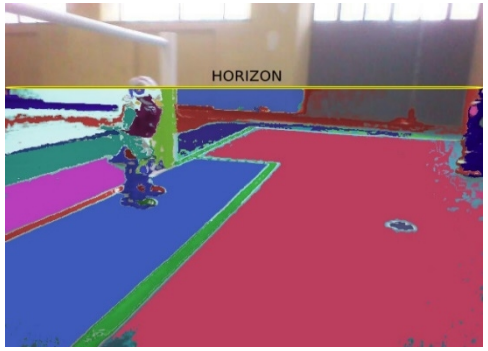


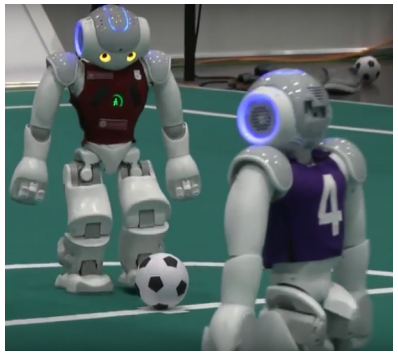











Maggio 2018

References and Credits

Questo materiale deriva da:

Alberto Pretto – Sapienza Università di Roma
Introduction to PCL: The Point Cloud Library
Basic topics

http://www.dis.uniroma1.it/~pretto/download/pcl_intro.pdf

Gestione dati 2D

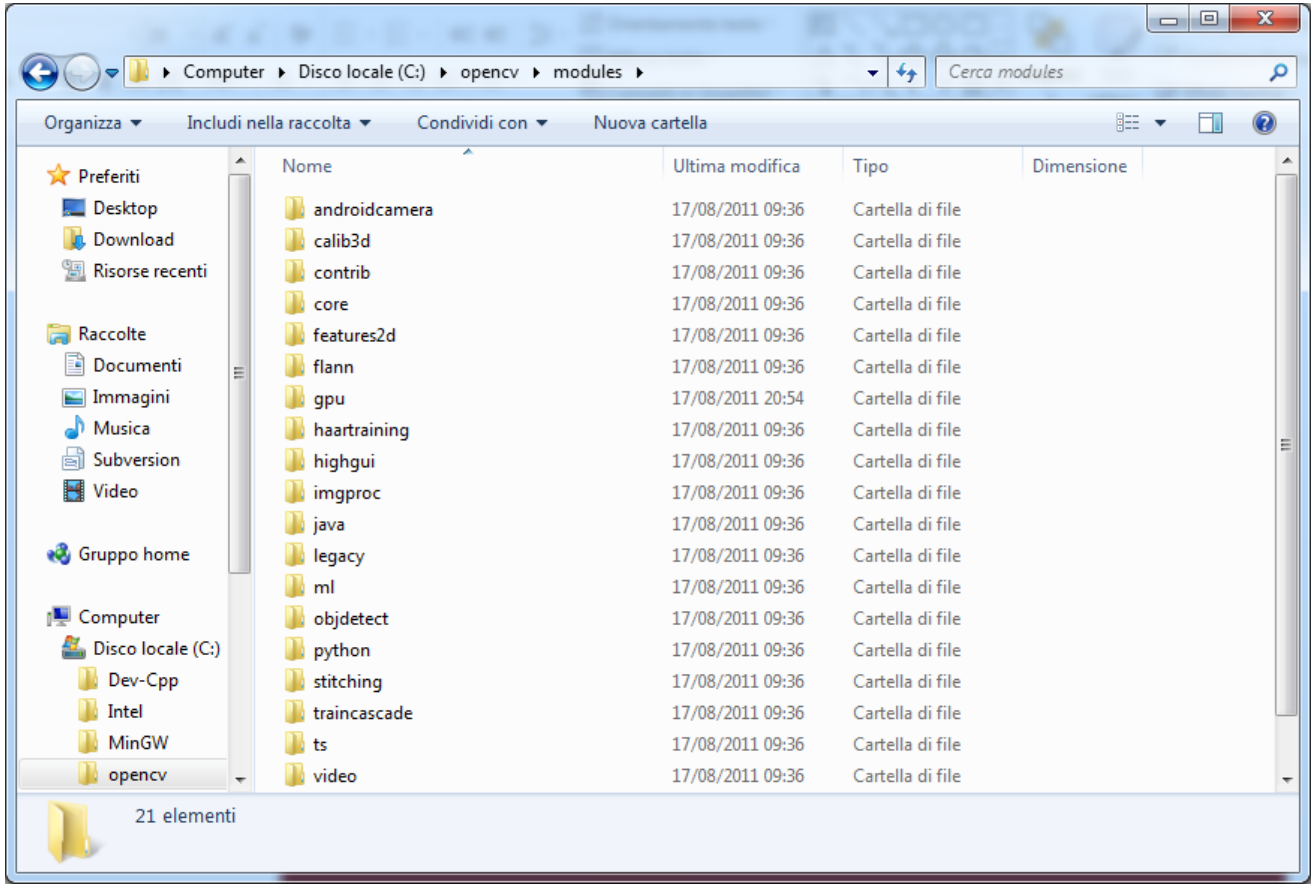
OpenCV (Open Source Computer Vision) is a library of programming functions for realtime computer vision

- BSD Licensed - free for commercial use
- C++, C, Python and Java (Android) interfaces
- Supports Windows, Linux, Android, iOS and Mac OS
- More than 2500 optimized algorithms

Moduli OpenCV

OpenCV has a modular structure

- core
- imgproc
- video
- calib3d
- features2d
- objdetect
- highgui
- gpu
- ...



Processamento delle immagini

core - a compact module defining basic data structures, including the dense multi-dimensional array Mat and basic functions used by all other modules.

imgproc - an image processing module that includes linear and non-linear image filtering, geometrical image transformations (resize, affine and perspective warping, generic table-based remapping), color space conversion, histograms, and so on.

Processamento delle immagini

features2d - salient feature detectors, descriptors, and descriptor matchers.

highgui - an easy-to-use interface to video capturing, image and video codecs, as well as simple UI capabilities.

objdetect - detection of objects and instances of the predefined classes (for example, faces, eyes, mugs, people, cars, and so on).

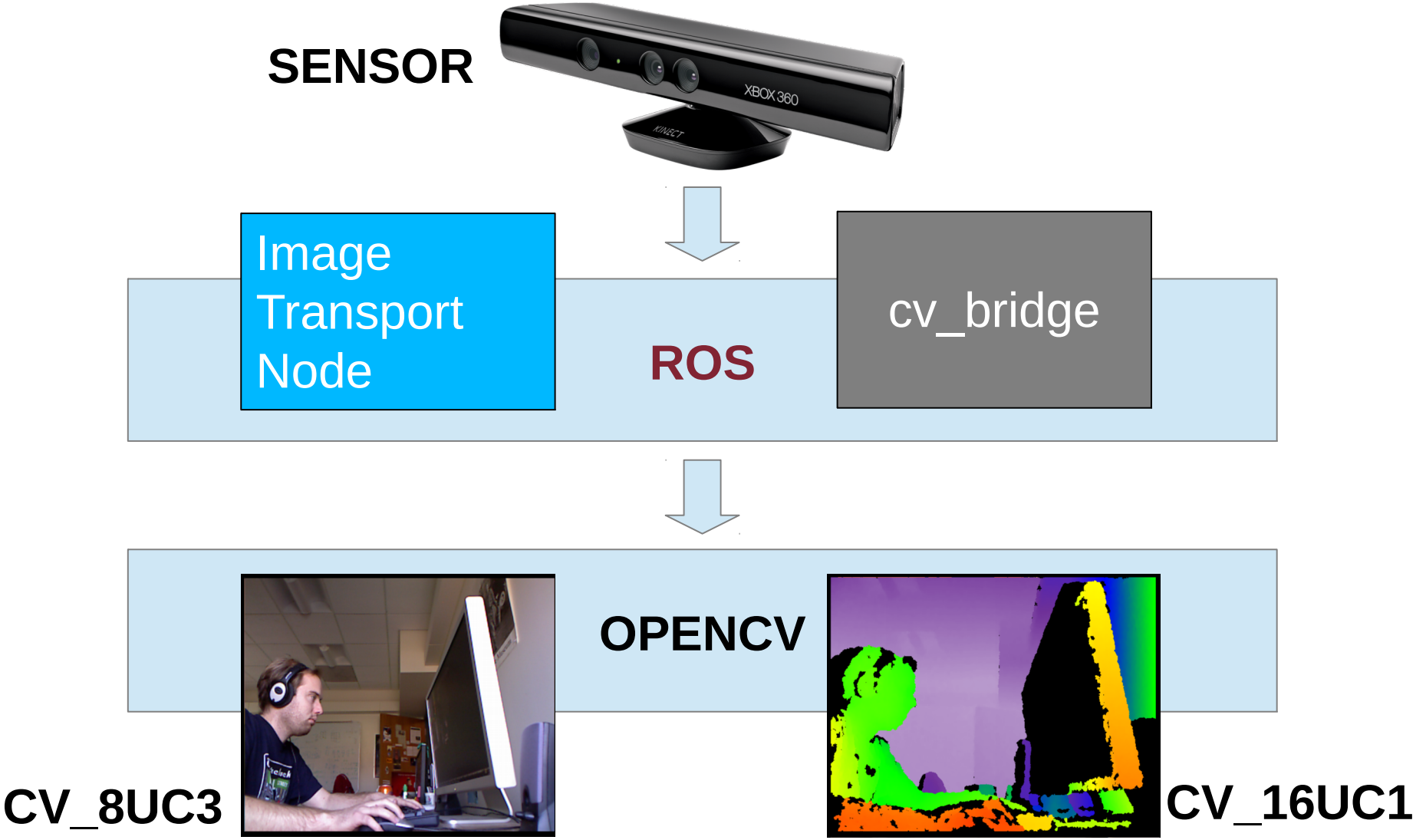
Processamento delle immagini

features2d - salient feature detectors, descriptors, and descriptor matchers.

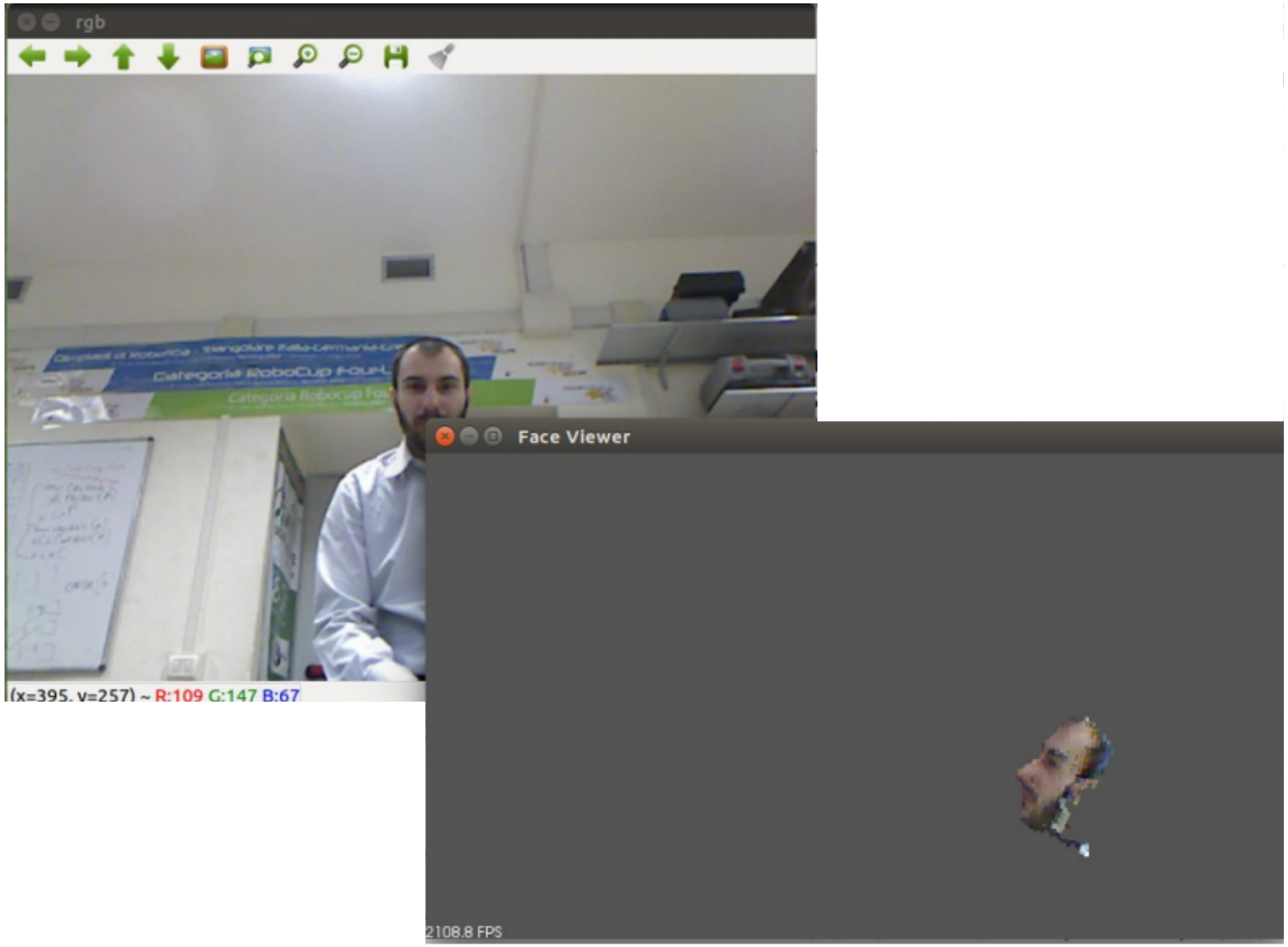
highgui - an easy-to-use interface to video capturing, image and video codecs, as well as simple UI capabilities.

objdetect - detection of objects and instances of the predefined classes (for example, faces, eyes, mugs, people, cars, and so on).

OpenCV e ROS


SENSOR
XBOX 360
KINECT
Image Transport Node
ROS
cv_bridge
OPENCV
CV_8UC3
CV_16UC1

Face Visualizer 3D



CMakeLists.txt

CMakeLists.txt (~/catkin_ws/src/face_visualizer_3d) - gedit

Open Save

```
1 cmake_minimum_required(VERSION 2.8.3)
2 project(face_visualizer_3d)
3
4 ## Find catkin macros and libraries
5 ## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
6 ## is used, also find other catkin packages
7 find_package(catkin REQUIRED COMPONENTS
8   cv_bridge
9   pcl_conversions
10  pcl_ros
11  roscpp
12  sensor_msgs
13  std_msgs
14 )
15
16 find_package(OpenCV REQUIRED
17   COMPONENTS
18   opencv_highgui
19   opencv_objdetect
20   CONFIG
21 )
22
23 ## System dependencies are found with CMake's conventions
24 # find_package(Boost REQUIRED COMPONENTS system)
25
26
27 ## Uncomment this if the package has a setup.py. This macro ensures
28 ## modules and global scripts declared therein get installed
29 ## See http://ros.org/doc/api/catkin/html/user_guide/setup_dot_py.html
30 # catkin_python_setup()
31
```

CMake Tab Width: 8 Ln 109, Col 2 INS

CMakeLists.txt

CMakeLists.txt (~/catkin_ws/src/face_visualizer_3d) - gedit

Open Save

```
 99 ############
100 ## Build ##
101 ############
102
103 ## Specify additional locations of header files
104 ## Your package locations should be listed before other locations
105 # include_directories(include)
106 include_directories(
107   include
108   ${catkin_INCLUDE_DIRS}
109 )
110
111 ## Declare a cpp library
112 # add_library(face_visualizer_3d
113 #   src/${PROJECT_NAME}/face_visualizer_3d.cpp
114 # )
115
116 ## Declare a cpp executable
117 add_executable(face_visualizer_3d_node src/face_visualizer_3d_node.cpp)
118
119 ## Add cmake target dependencies of the executable/library
120 ## as an example, message headers may need to be generated before nodes
121 # add_dependencies(face_visualizer_3d_node face_visualizer_3d_generate_messages_cpp)
122
123 ## Specify libraries to link a library or executable target against
124 target_link_libraries(face_visualizer_3d_node
125   ##vtkCommon
126   ##vtkFiltering
127   ##vtkRendering
128   ${catkin_LIBRARIES}
129   ${OpenCV_LIBRARIES}
130 )
```

CMake Tab Width: 8 Ln 109, Col 2 INS

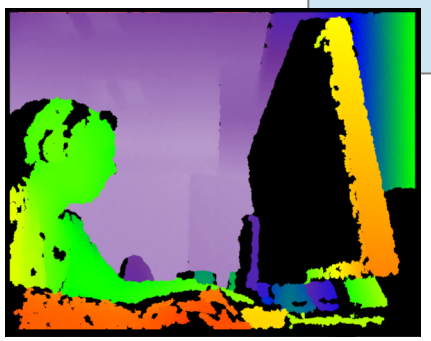
Gestione dati 3D

The Point Cloud Library (PCL) is a standalone, large scale, open project for 2D/3D image and point cloud processing

- Collection of Libraries focused on Point Cloud processing
- More than 450 developers/contributors
- Over 60 Tutorials and many examples
- BSD Licensed - free for commercial use

PCL e ROS

SENSOR



ROS

OPENCV

PCL

Point cloud: a definition

- A point cloud is a data structure used to represent a collection of multi-dimensional points
- It is commonly used to represent three-dimensional data

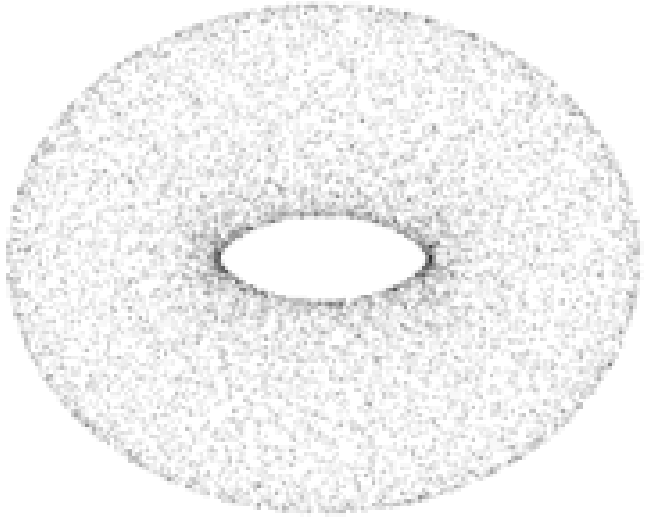


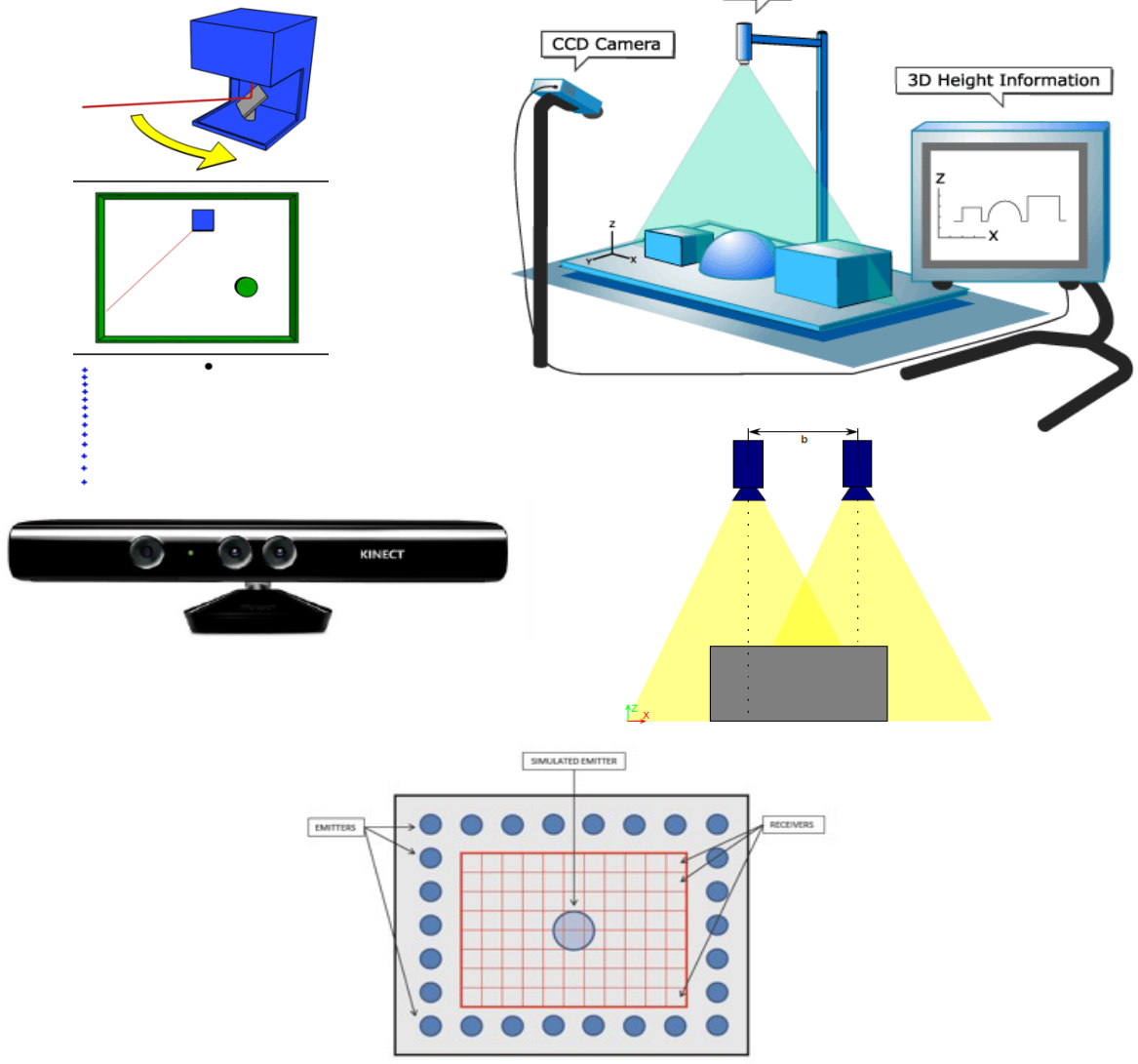
image from
https://en.wikipedia.org/wiki/Point_cloud

Point cloud: a definition

- The points in the point cloud usually represent the X, Y, and Z geometric coordinates of a sampled surface
- Each point can hold additional information: RGB colors, intensity values, etc...

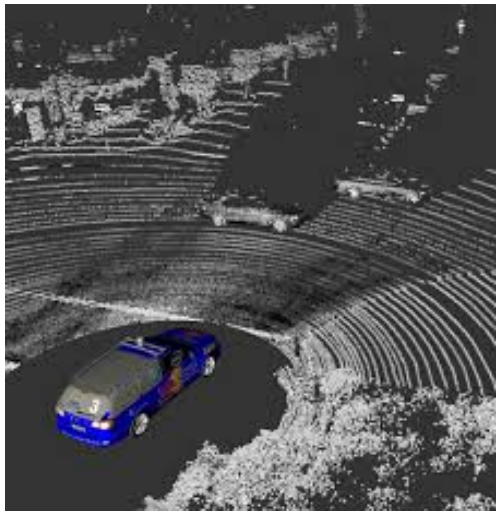
Where do they come from?

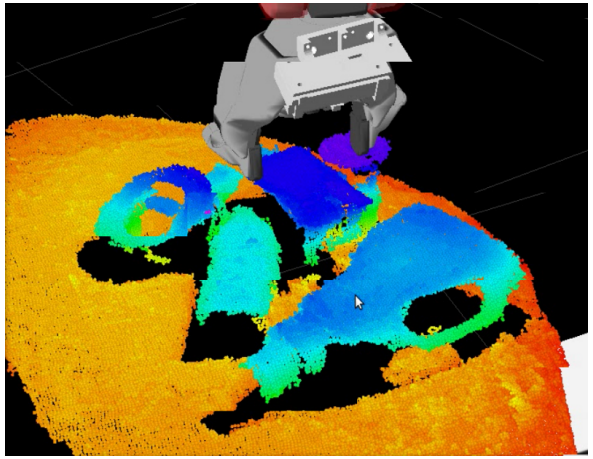
- 2/3D Laser scans
- Laser triangulation
- Stereo cameras
- RGB-D cameras
- Structured light cameras
- Time of flight cameras

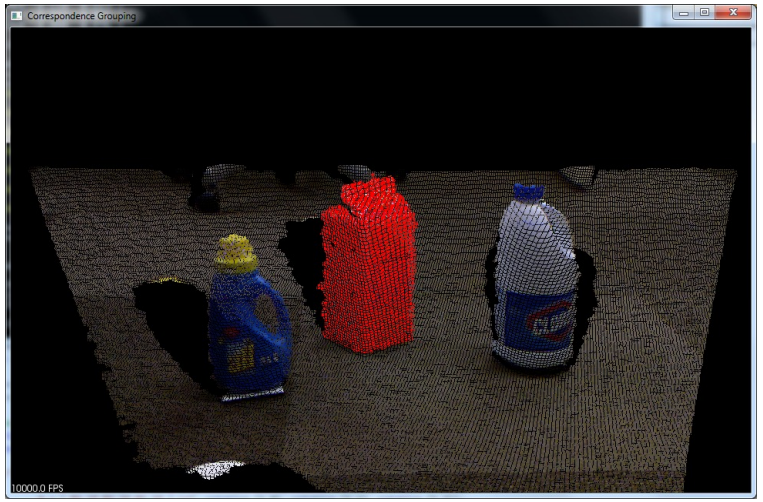


Point clouds in robotics

- 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

Point Cloud Library

- 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 that 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

CMakeLists.txt

```
project( pcl_test )
cmake_minimum_required (VERSION 2.8)
cmake_policy(SET CMP0015 NEW)

find_package(PCL 1.7 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, ...

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>...
```

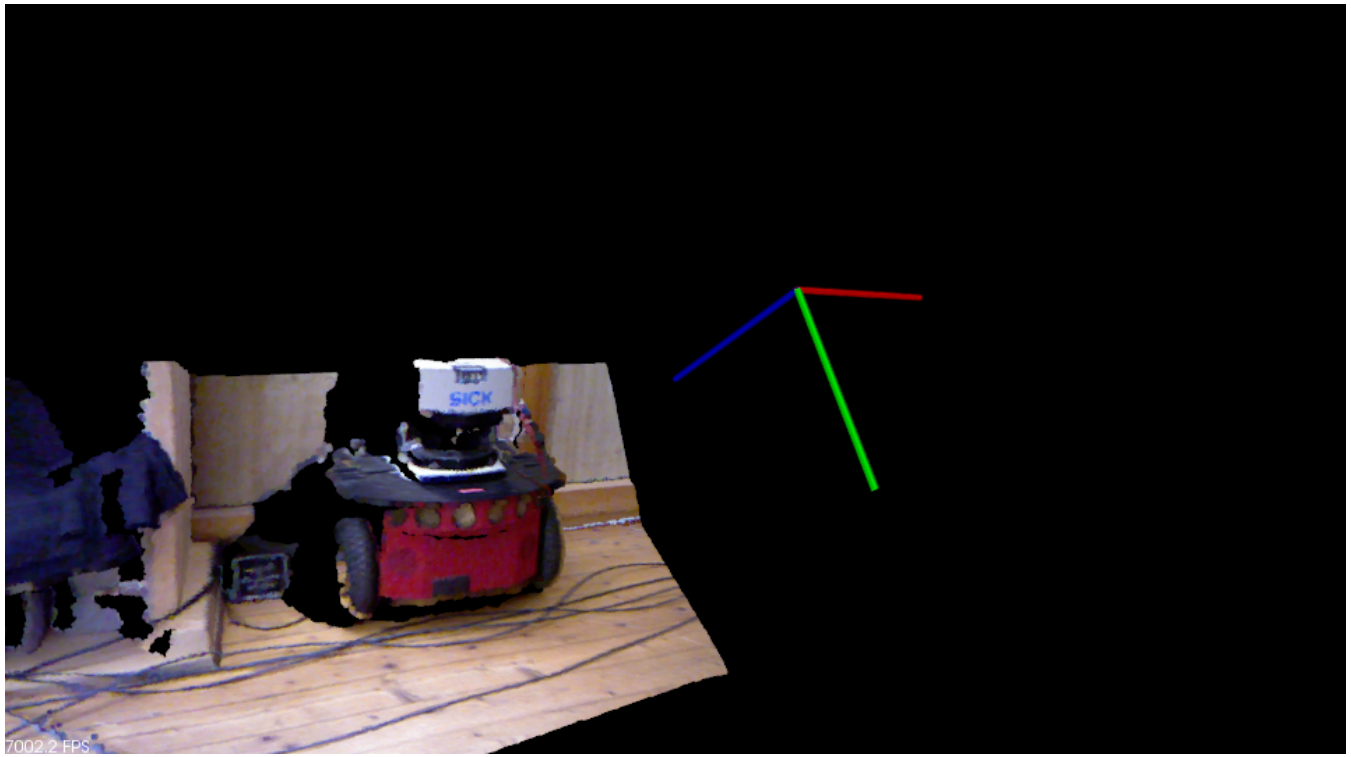
Funtions: pcl::io::loadPCDFile and pcl::io::savePCDFile

Example: create and save a PC

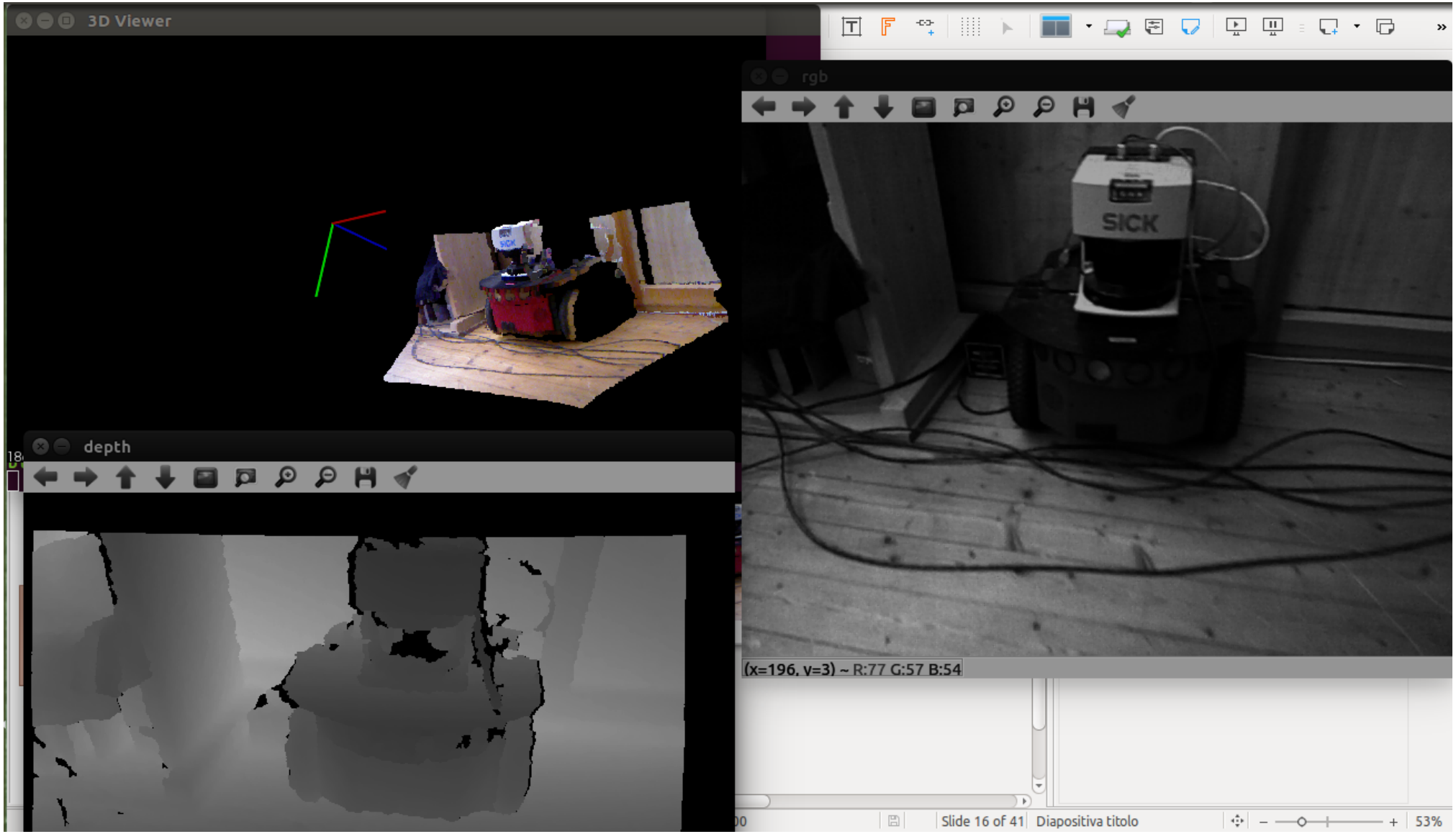
```
#include<pcl/io/pcd_io.h>
#include<pcl/point_types.h>
  //....
  pcl::PointCloud<pcl::PointXYZ>:: Ptr cloud_ptr (new pcl::PointCloud<pcl::PointXYZ>);
  cloud->width=50;
  cloud->height=1;
  cloud->isdense=false;
  cloud->points.resize(cloud.width*cloud.height);
  for(size_t i=0; i<cloud.points.size(); 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 );
```



depth2cloud.cpp



Basic Module Interface

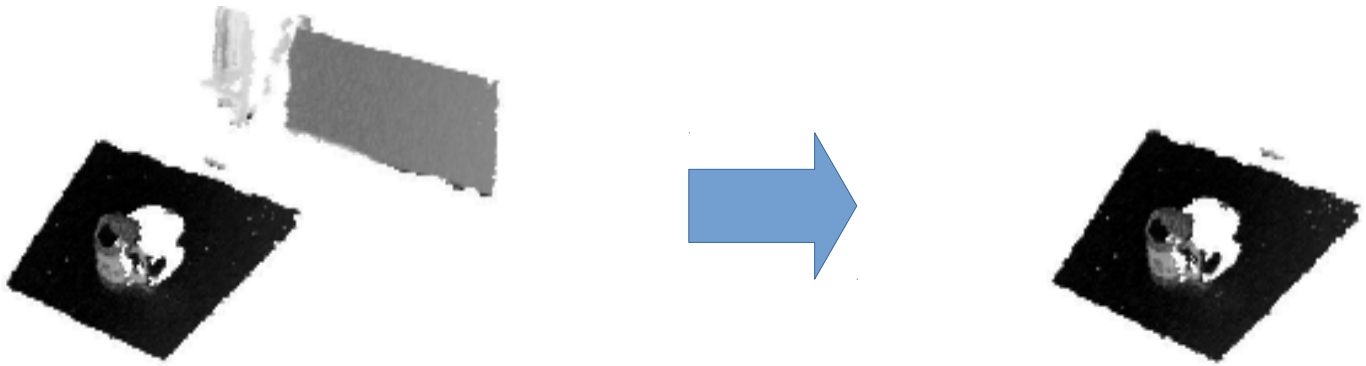
Filters, Features, Segmentation all use the same basic usage interface:

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

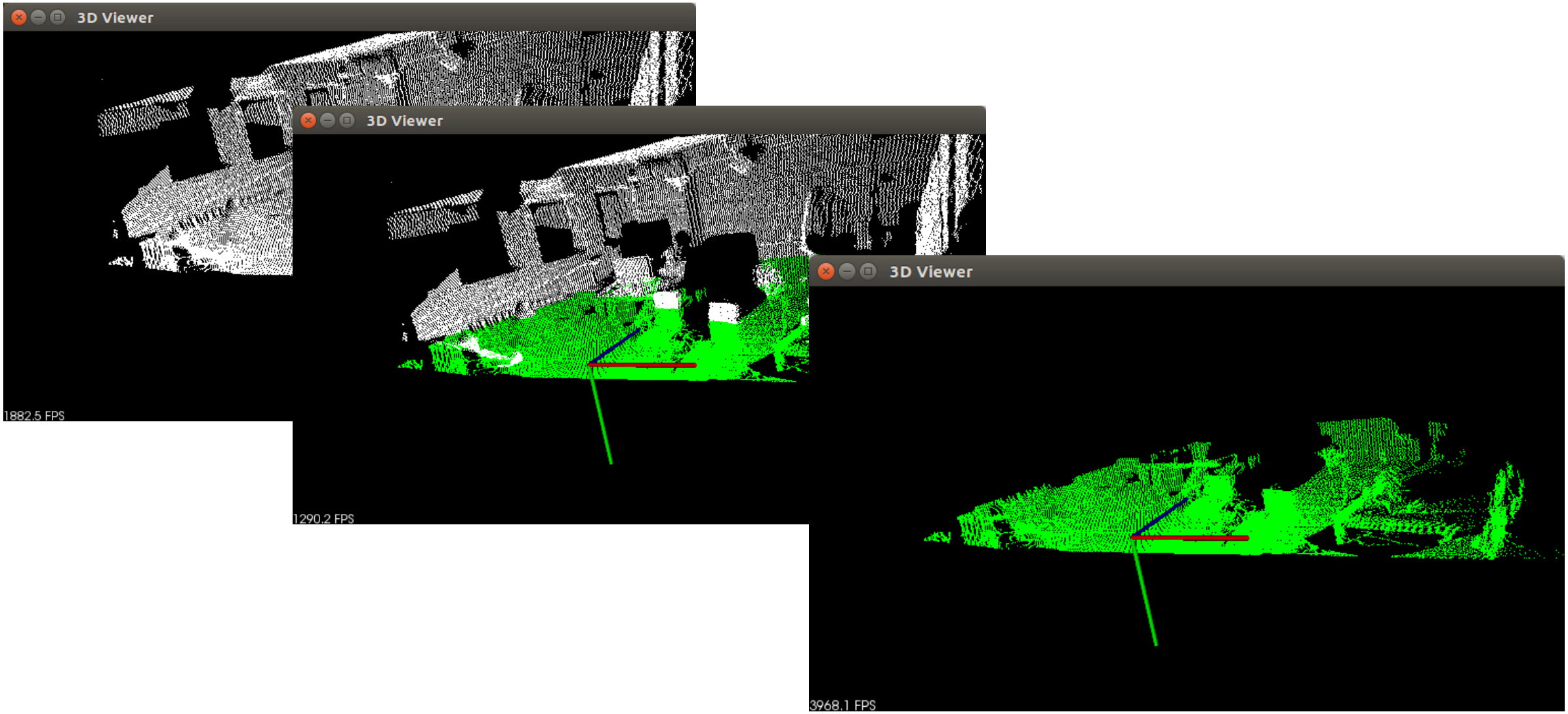
PassThrough Filter

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 );
```



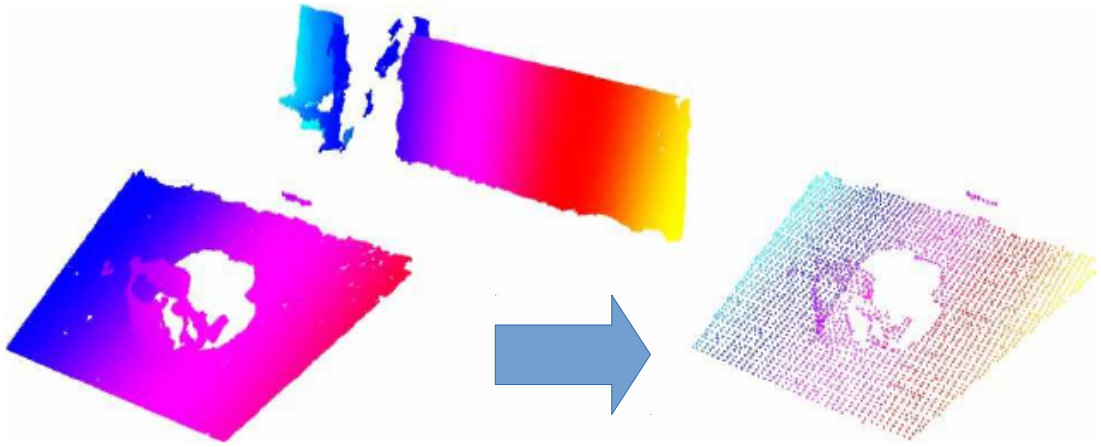
cloud_filters.cpp



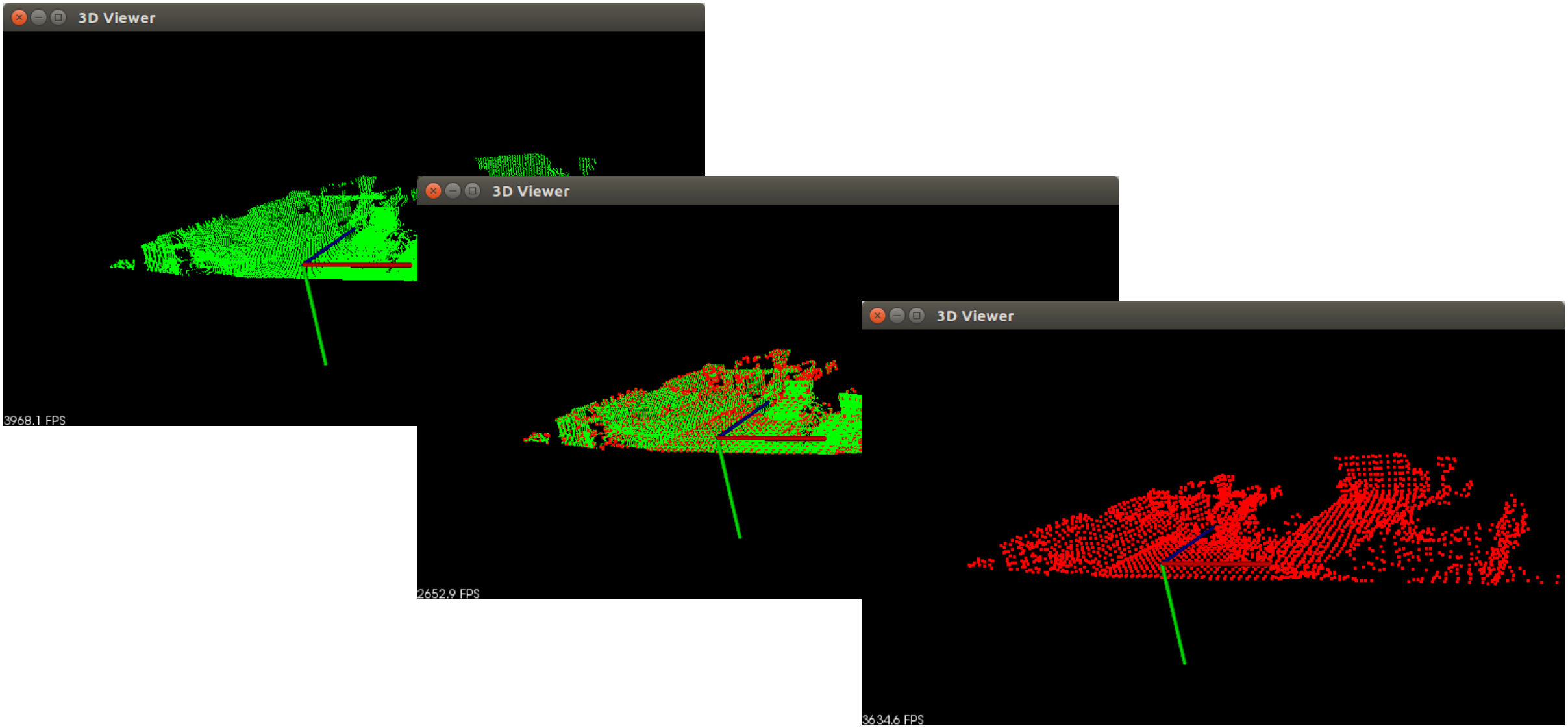
Downsampling

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 ) ;
```



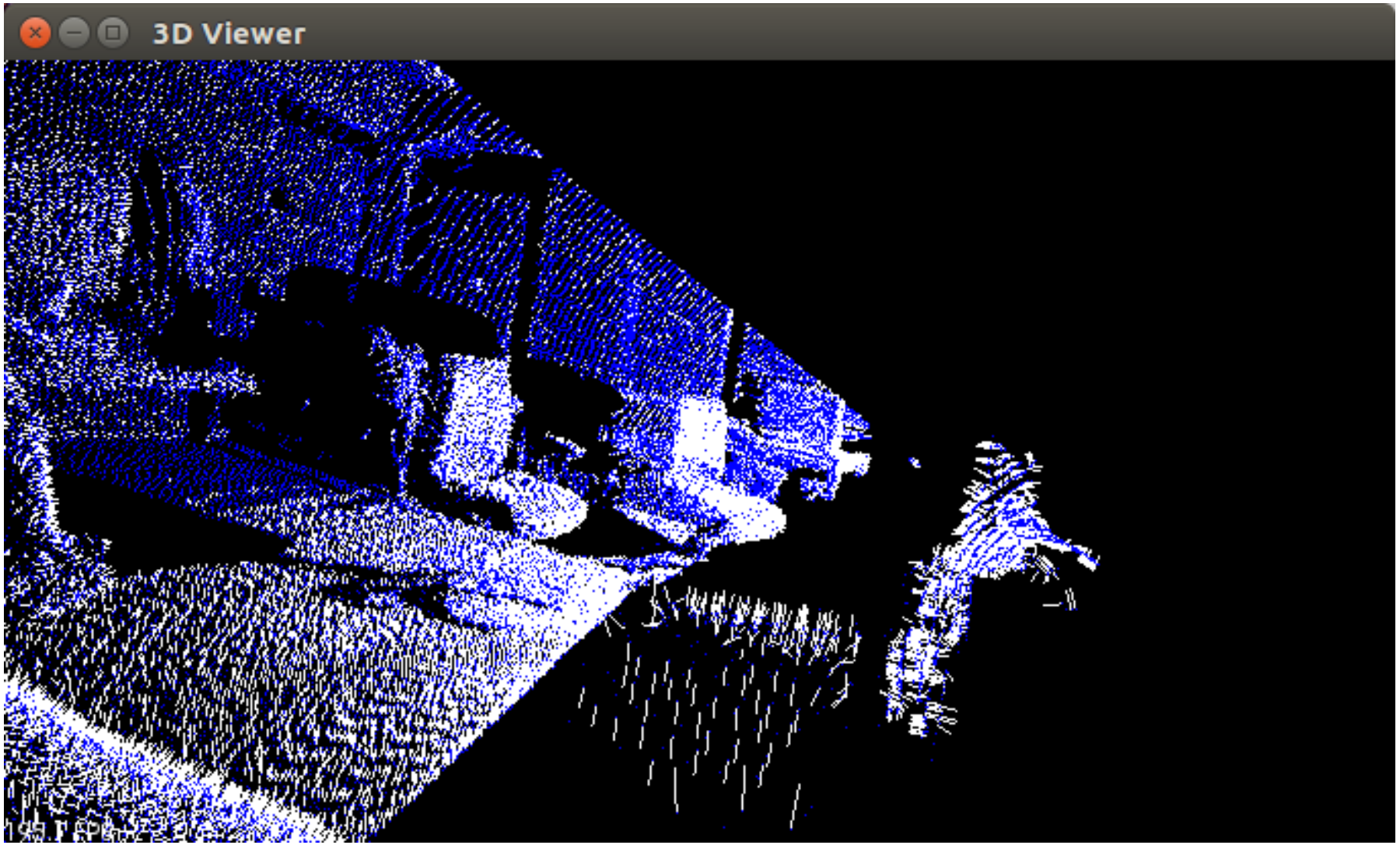
cloud_filters.cpp



Features 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 );
```

cloud_normals.cpp



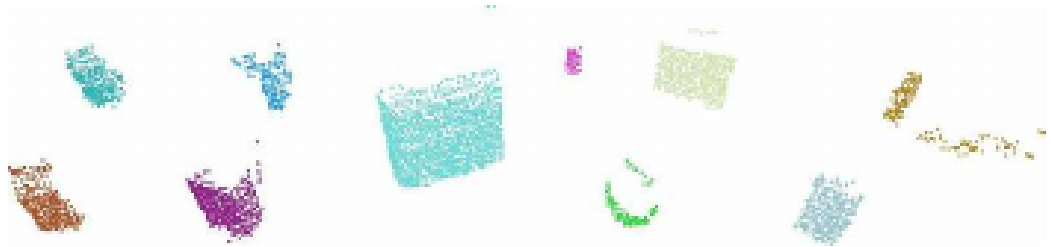
Segmentation example

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

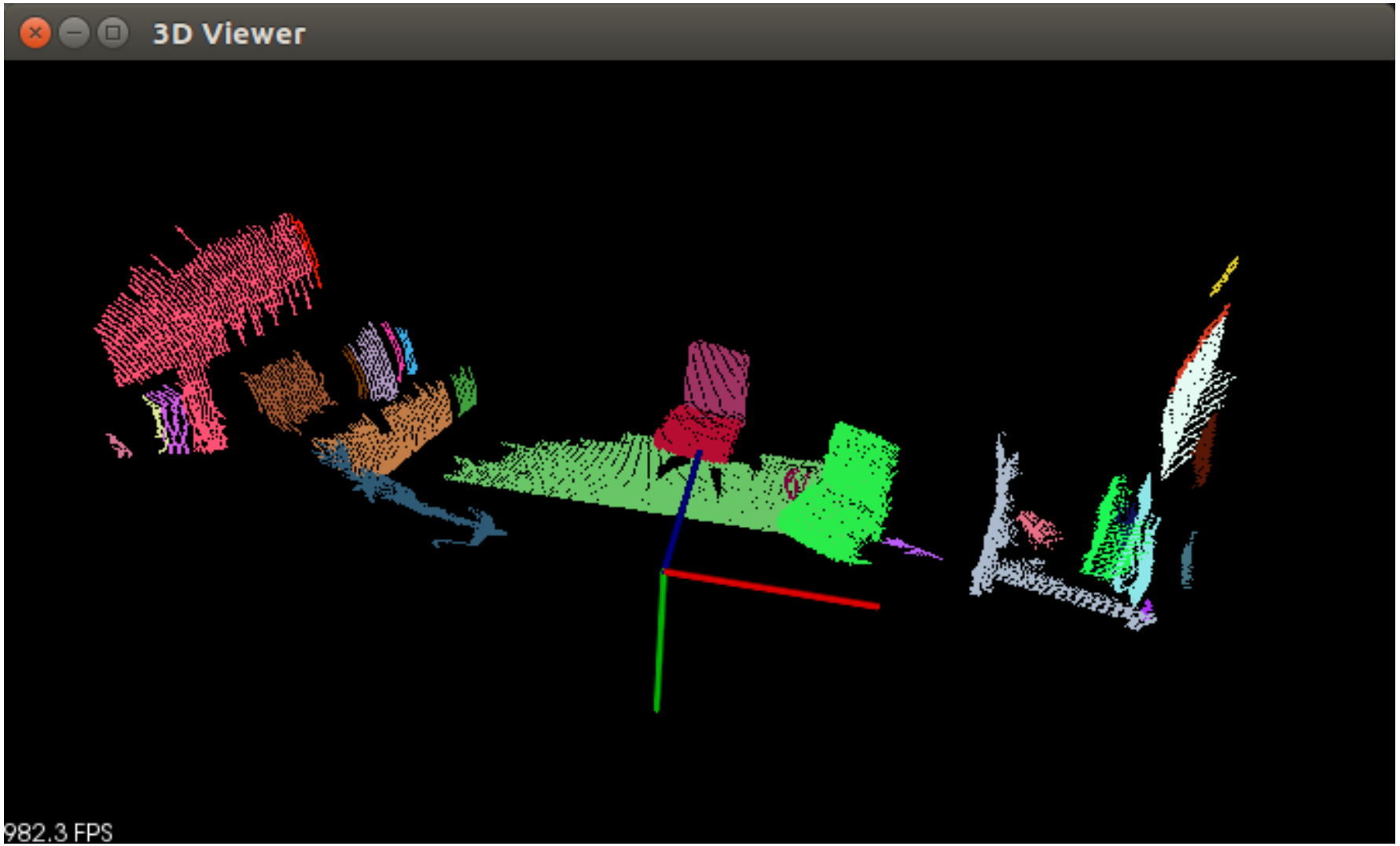
EuclideanClusterExtraction uses a distance threshold to the nearest neighbors of each point to decide if the two points belong to the same cluster.

Segmentation example

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

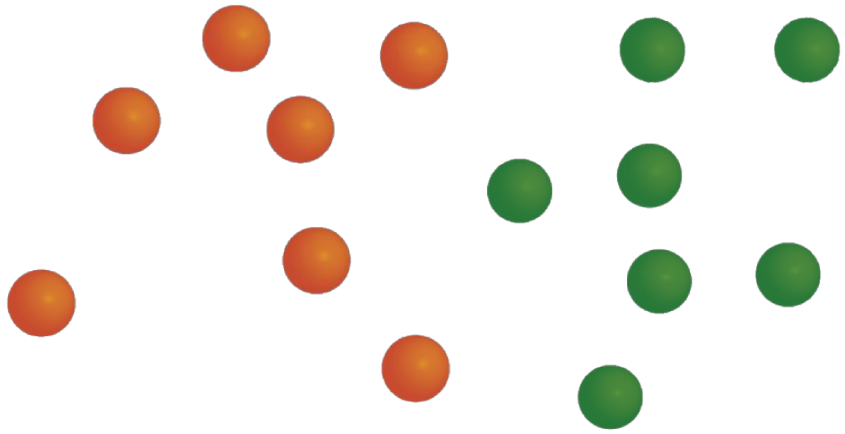


clustering.cpp



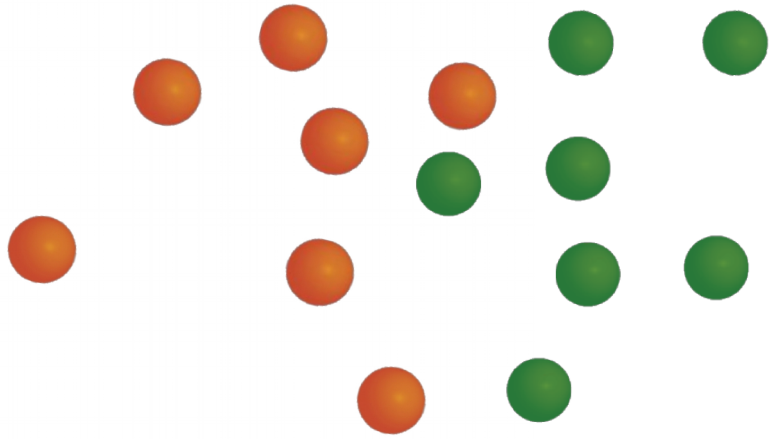
Point Cloud Registration

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

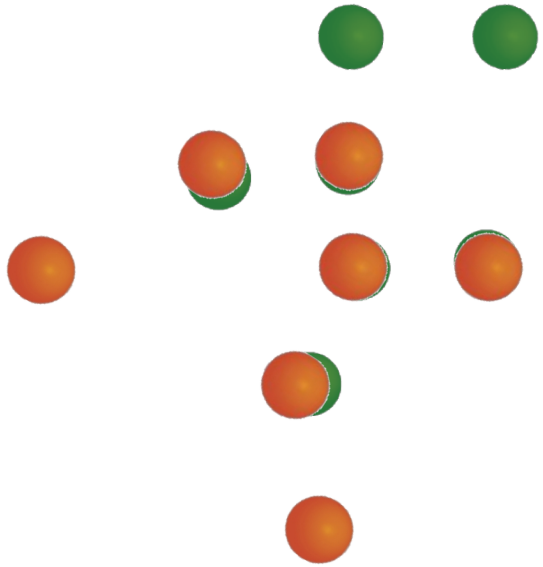


Point Cloud Registration

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

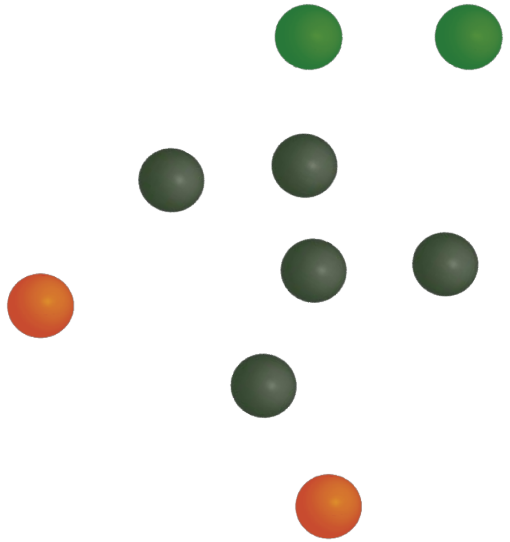
Point Cloud Registration

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



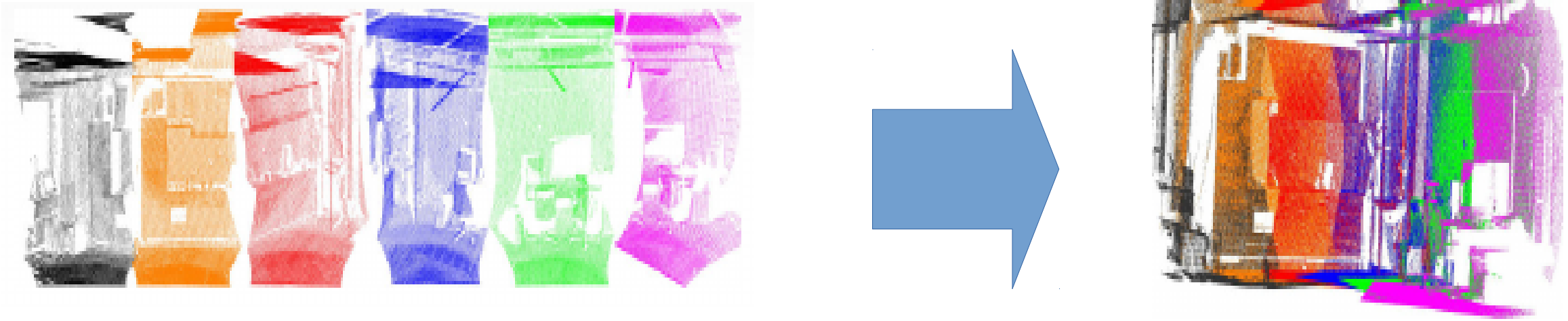
Iterative Closest Point

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

Input: points from two raw scans, initial estimation of the transformation, criteria for stopping the iteration

Output: refined transformation

Iterative Closest Point: Example



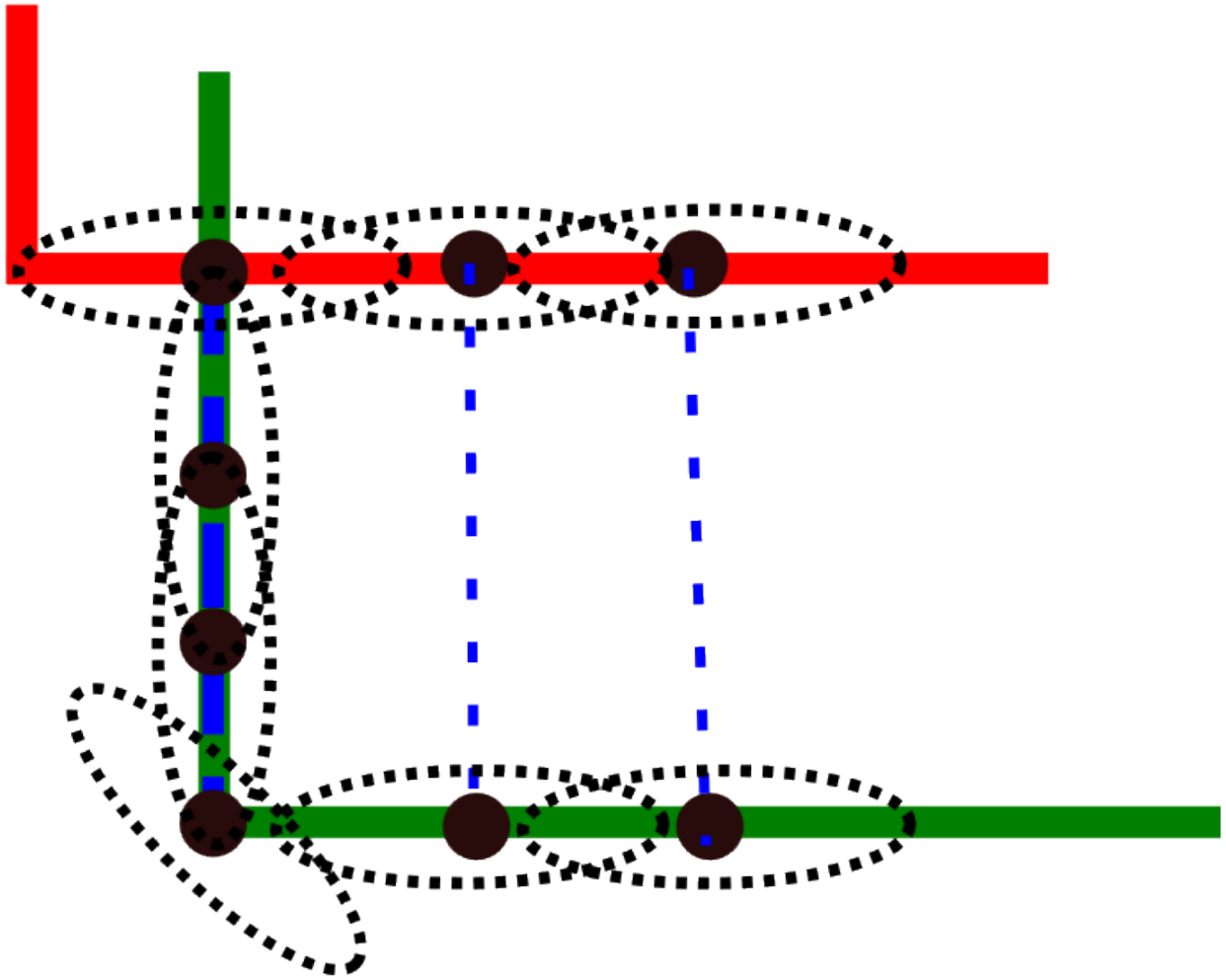
Iterative Closest Point: Algorithm

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: Code

```
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 ();
```

Plane-to-plane



All of the points along the vertical section of the green scan are incorrectly associated with a single point in the red scan

Generalized ICP

- 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

Generalized ICP: 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;
```

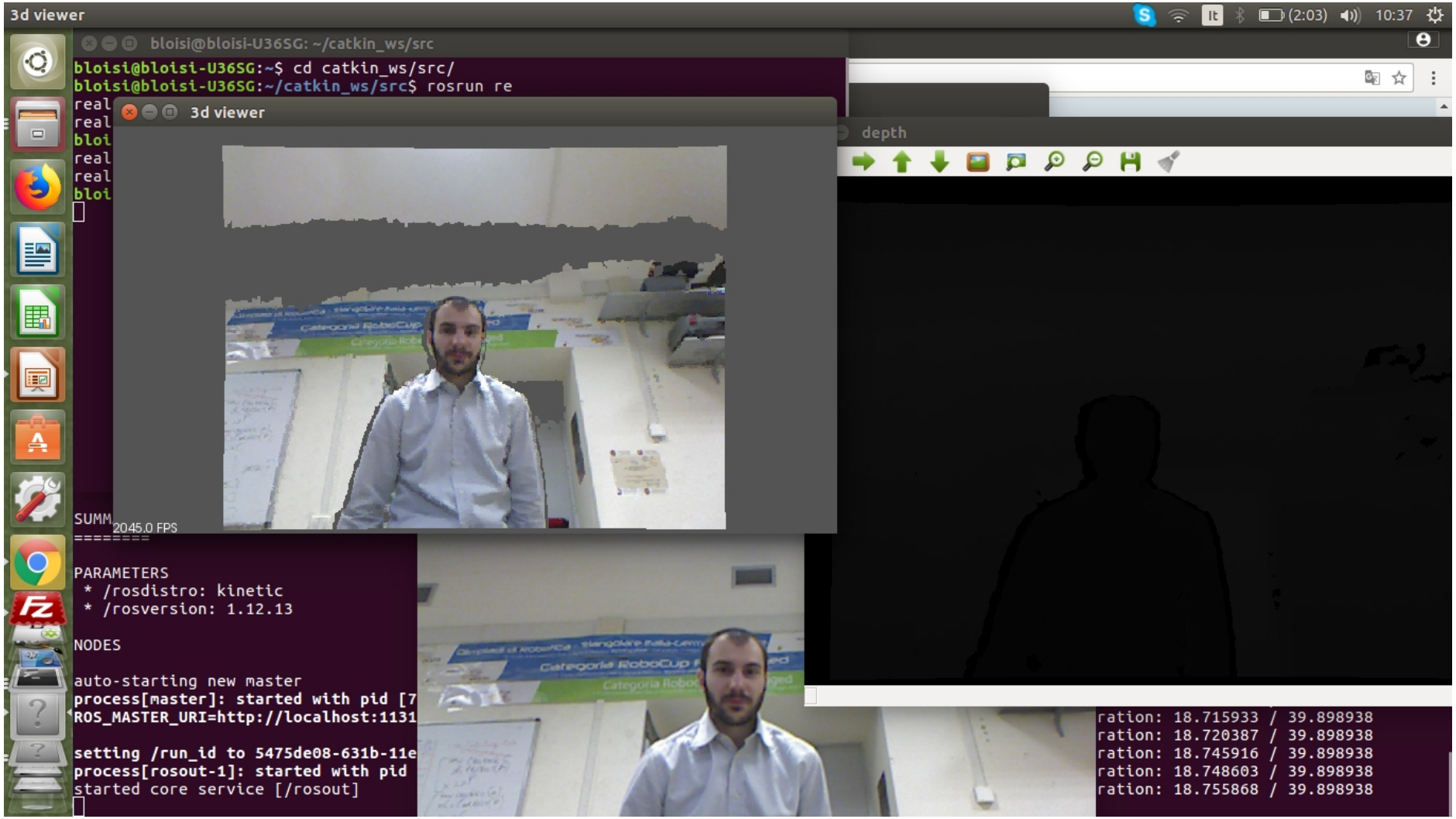
Esercizio 1

- Modificare il codice di

  http://profs.scienze.univr.it/~bloisi/corsi/lezioni/face_visualizer_3d.zip

  mostrando la point cloud completa e non soltanto i punti relativi al volto

Esercizio 1 - esecuzione



Esercizio 2 (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/

Esercitazione (2/3)

Then, 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)

Esercitazione (3/3)

- 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

Suggerimenti (1/2)

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::IMREAD_ANYDEPTH :
cv::Mat input_depth = cv::imread("test_depth.png", cv::IMREAD_ANYDEPTH);

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();
```

Suggerimenti (2/2)

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 pointcloud

`alignmentTransform` is the local transform computed using Generalized ICP

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

UNIVERSITÀ di VERONA
Dipartimento di INFORMATICA

Corso di Laboratorio Ciberfisico
Modulo di Robot Programming with ROS

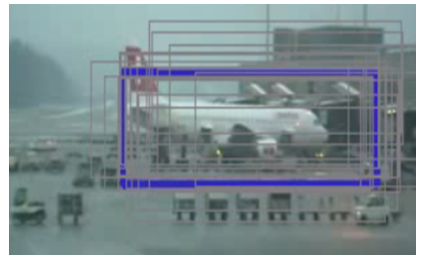
Acquisizione e gestione dati 3D

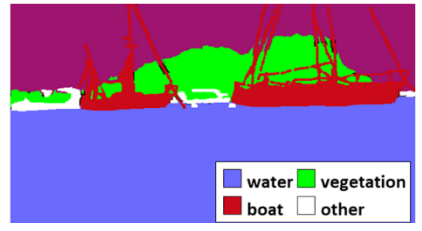
pcl

Docente:
Domenico Daniele Bloisi

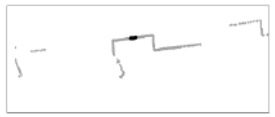
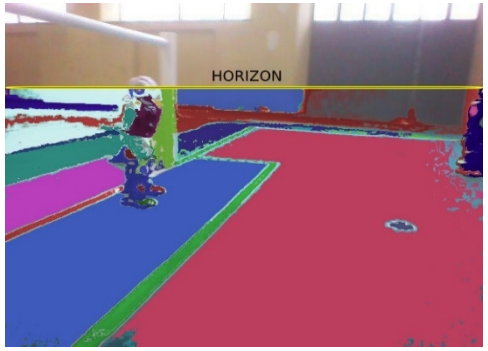


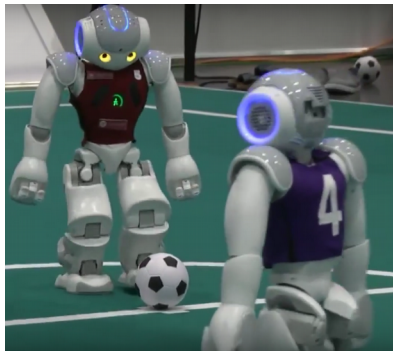











Maggio 2018