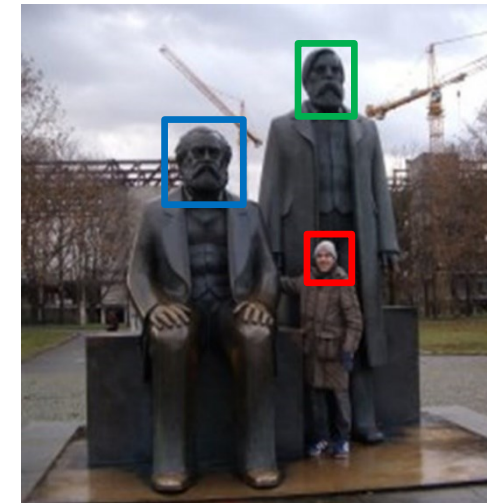
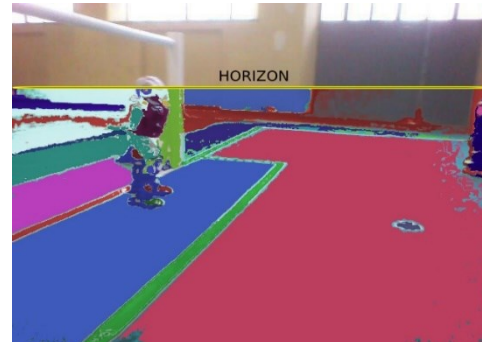
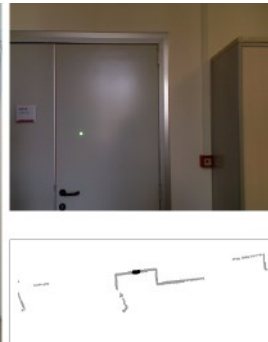
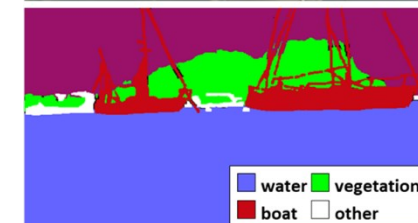
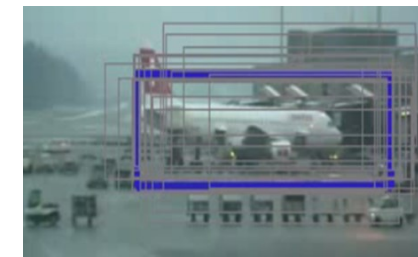


**UNIVERSITÀ DEGLI STUDI  
DELLA BASILICATA**

*Corso di Visione e Percezione*  
*A.A. 2019/2020*

Docente  
**Domenico Daniele Bloisi**

# Visualizzazione dati 3D



Maggio 2020

# References and Credits

---

Queste slide sono adattate 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](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 real-time computer vision

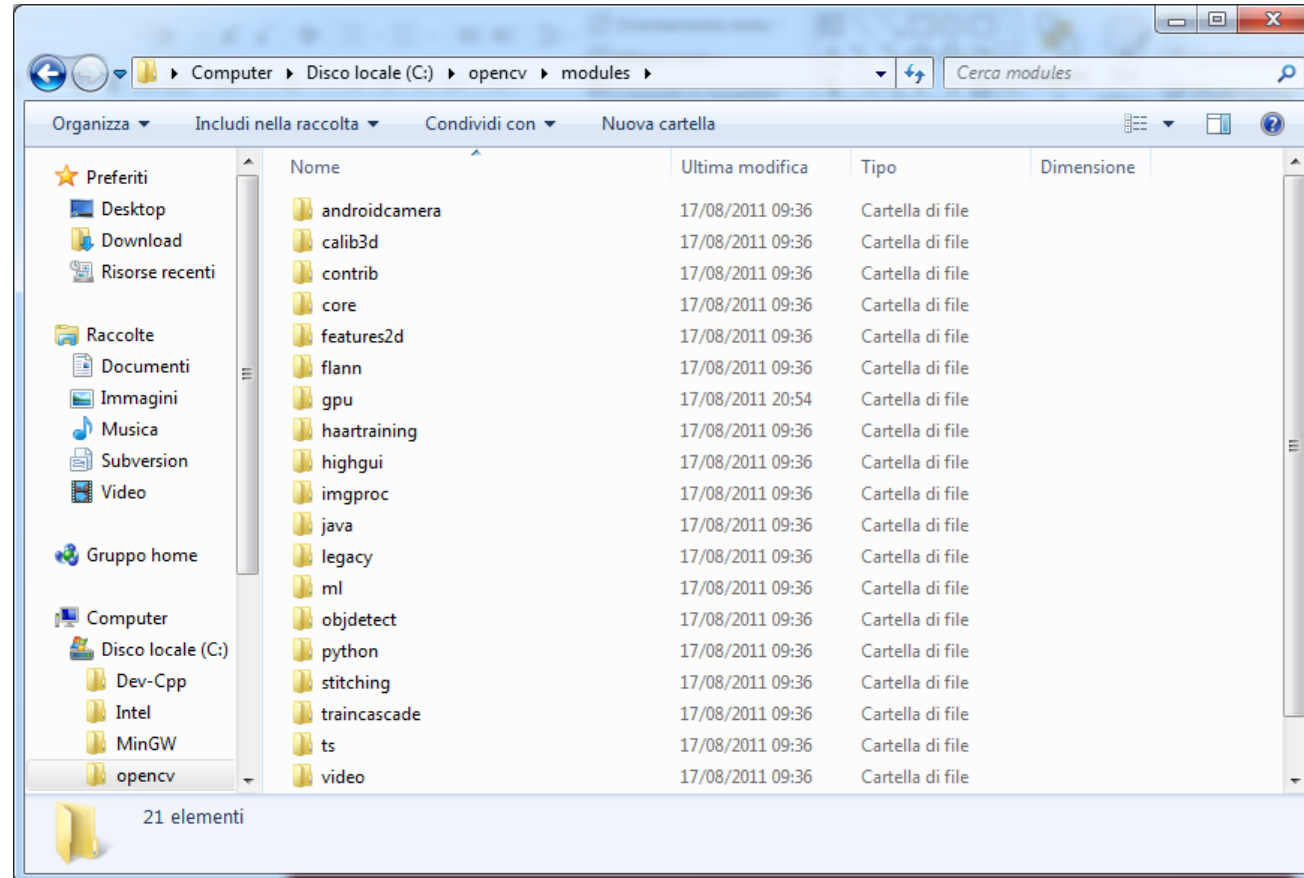


- BSD Licensed - free for commercial use
- C++, C, Python and Java (Android) interfaces
- Supports Windows, Linux, Android, iOS and Mac OS
- More than 2500 optimized algorithms

# 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.

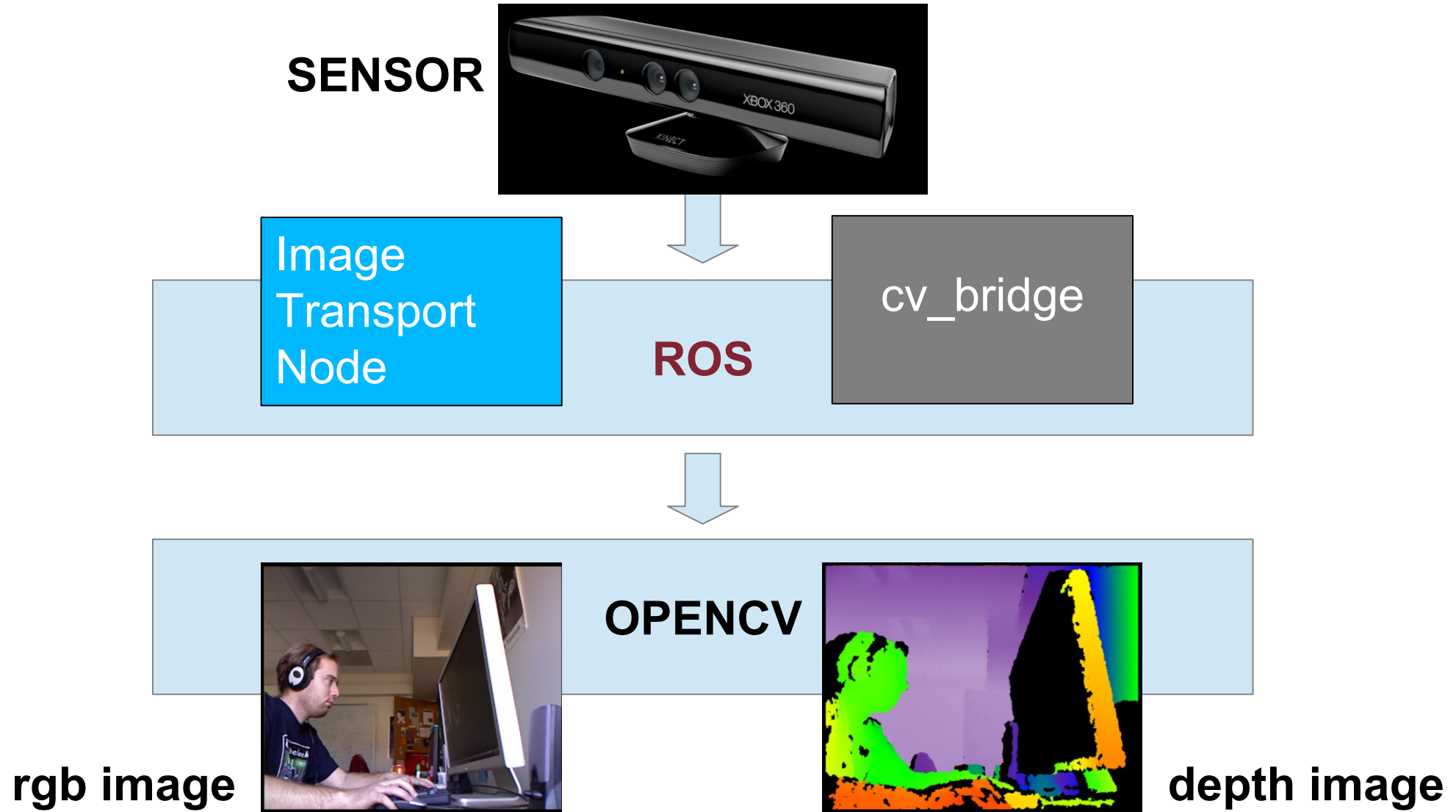
**features2d** - salient feature detectors, descriptors, and descriptor matchers.

**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

---



# Gestione dati 3D

---

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

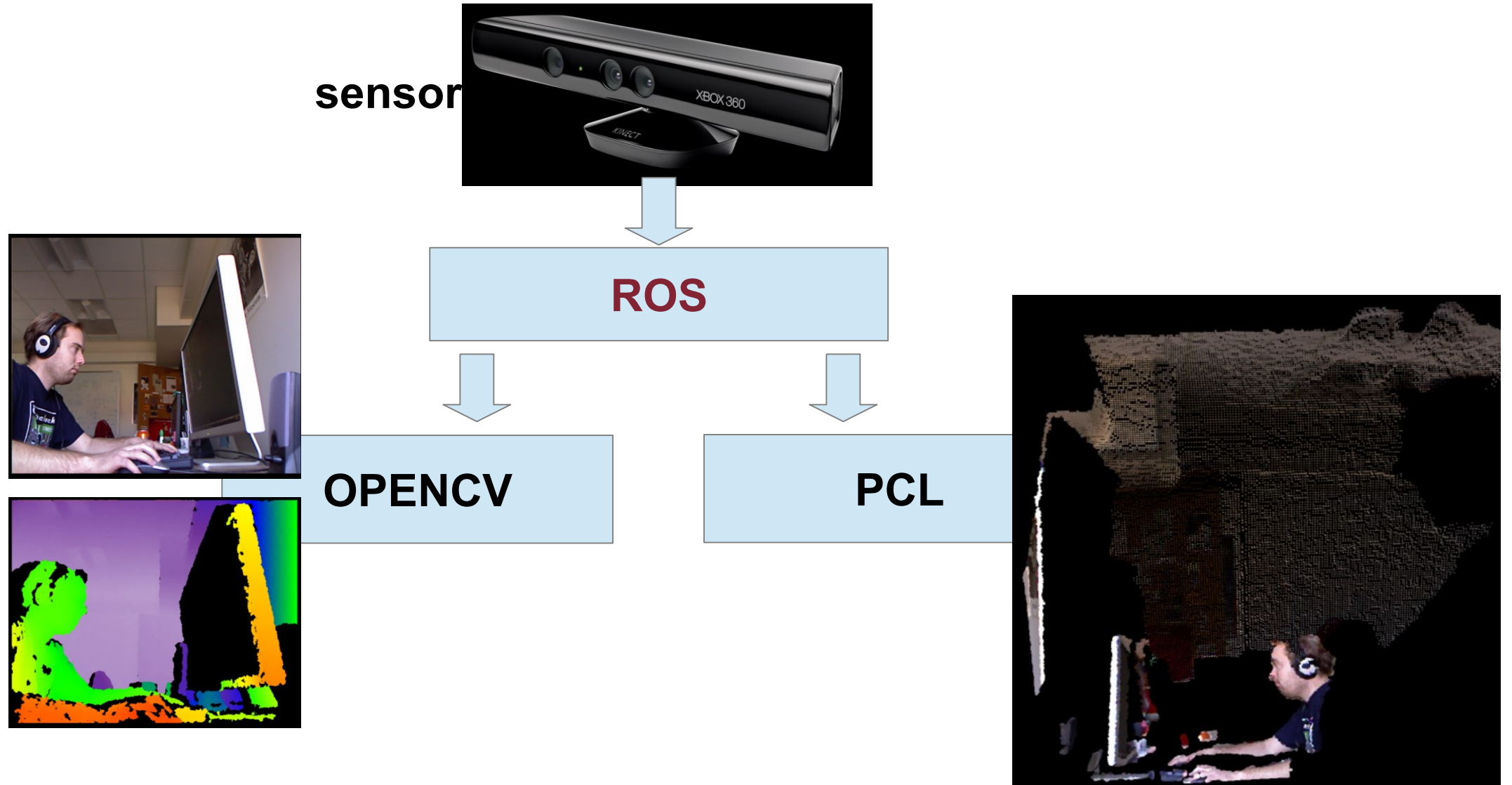


- Collection of Libraries focused on Point Cloud processing
- More than 450 developers/contributors
- Over 60 Tutorials and many examples
- BSD Licensed - free for commercial use



# PCL e ROS

---





# 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

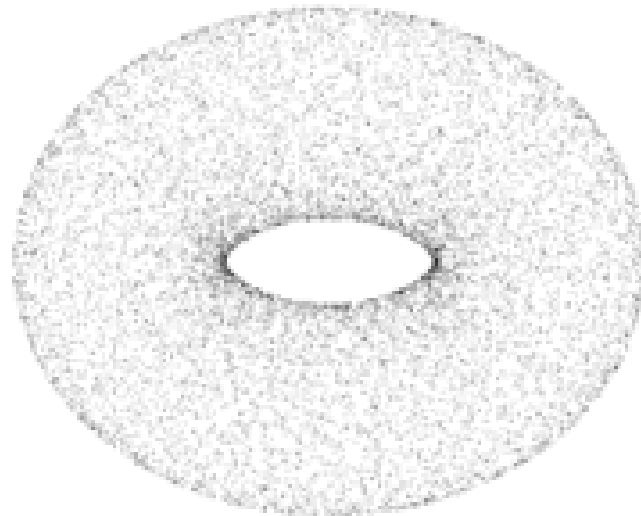


image from  
[https://en.wikipedia.org/wiki/Point\\_cloud](https://en.wikipedia.org/wiki/Point_cloud)

# Point cloud: a definition

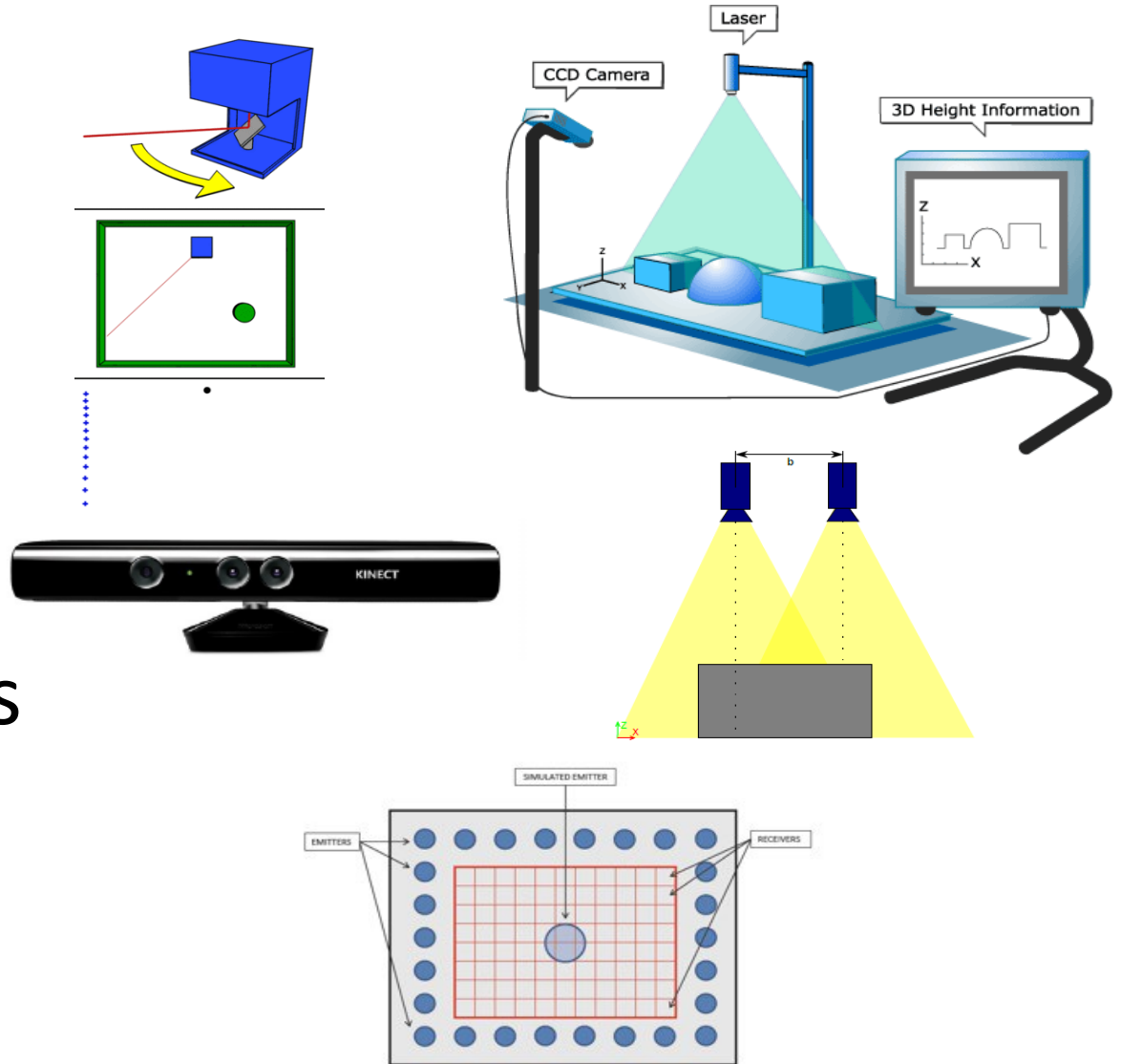
---

- 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...



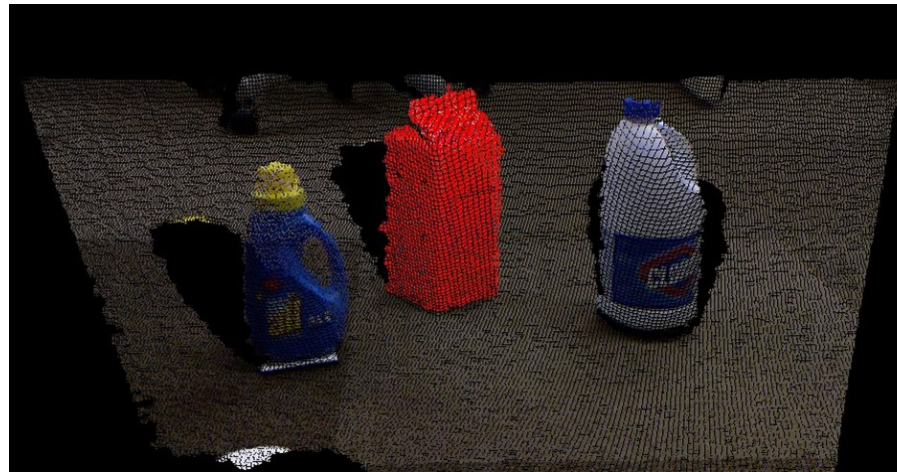
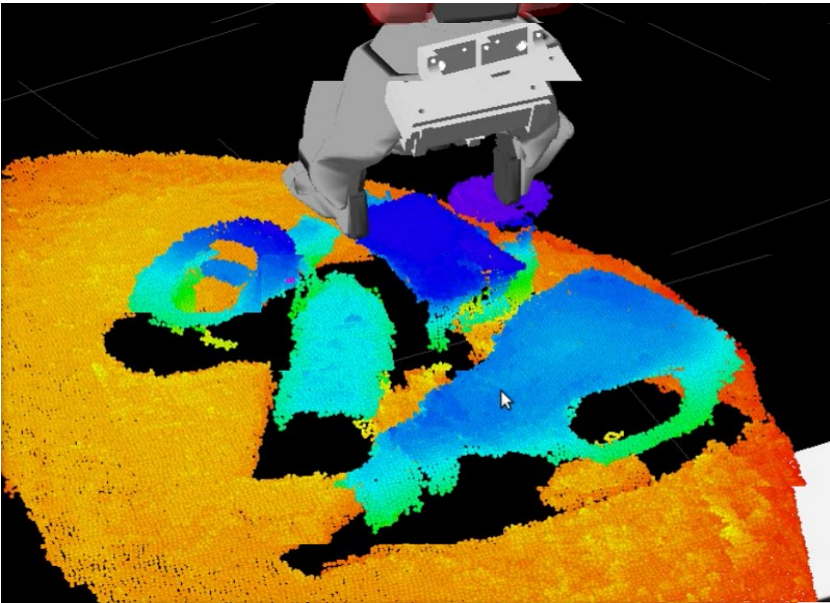
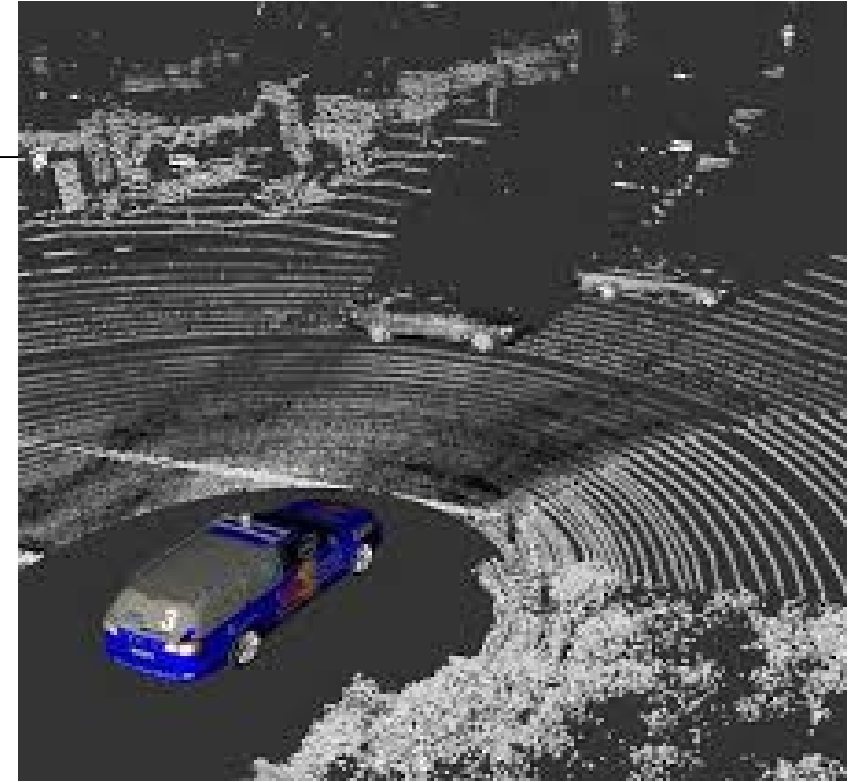
# Where do they come from?

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



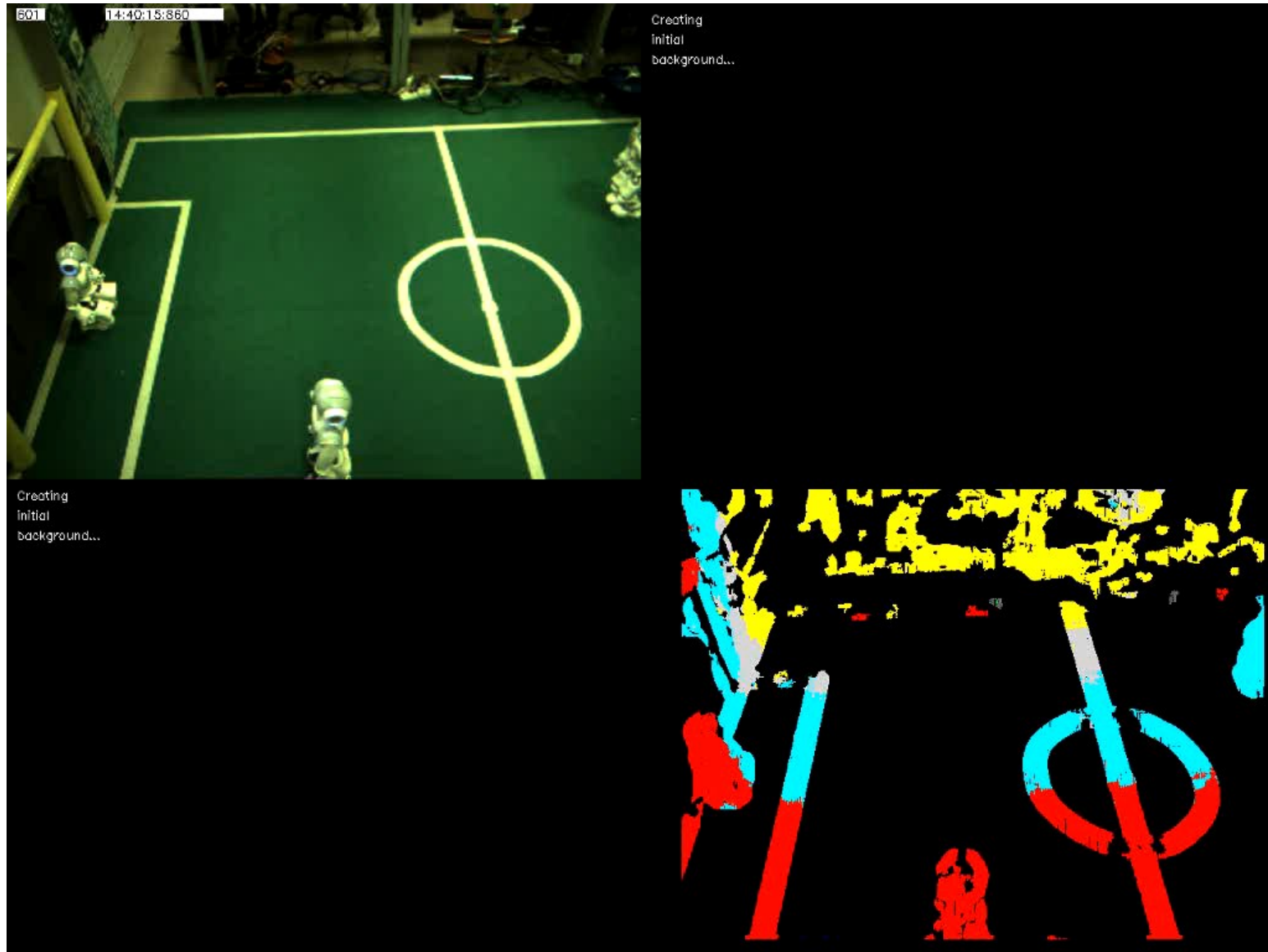
# Point clouds in robotics

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



# Offside detection

---



# Grasping

---



<https://youtu.be/HIMIEOdSttU>



# Point Cloud Library

---

→ [pointclouds.org](http://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 + ROS

---

- 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

# 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:** `pcl::io::loadPCDFile` and `pcl::io::savePCDFile`

# Create and save a PC

---

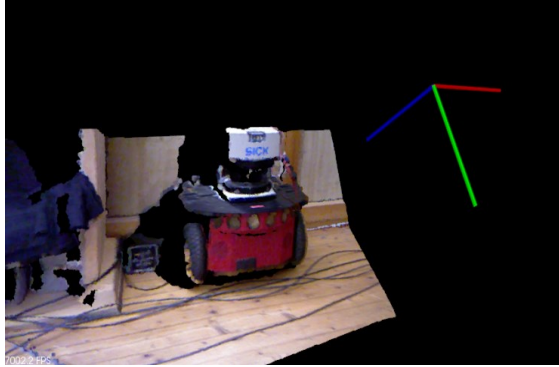
```
#include<pcl/io/pcd_io.h>
#include<pcl/point_types.h>

//....

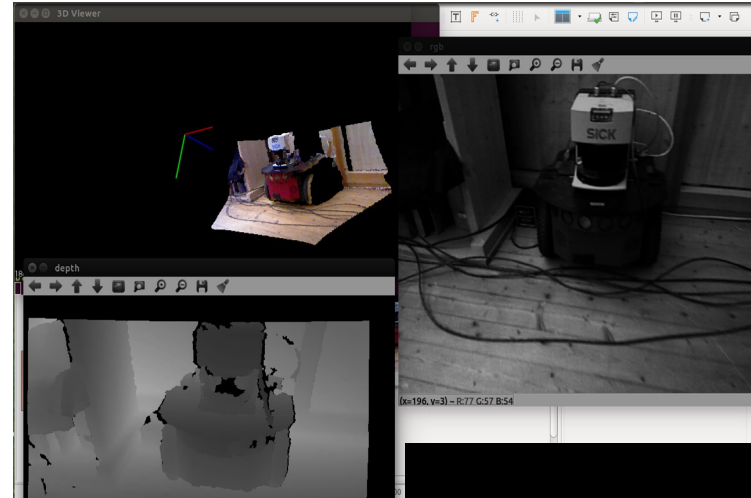
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_ptr(new pcl::PointCloud<pcl::PointXYZ>);
cloud->width = 50;
cloud->height = 1;
cloud->isdense = false;
cloud->points.resize(cloud.width*cloud.height);
for(size_t i = 0; i < cloud.points.size(); i++){
    cloud->points[i].x = 1024*rand() / (RANDMAX+1.0f);
    cloud->points[i].y = 1024*rand() / (RANDMAX+1.0f);
    cloud->points[i].z = 1024*rand() / (RANDMAX+1.0f);
}
pcl::io::savePCDFileASCII("testpcd.pcd", *cloud);

//....
```

# pcl\_examples

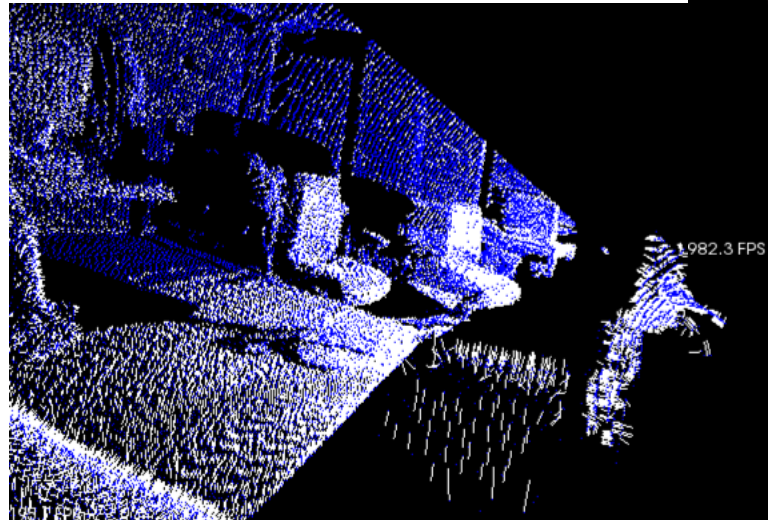
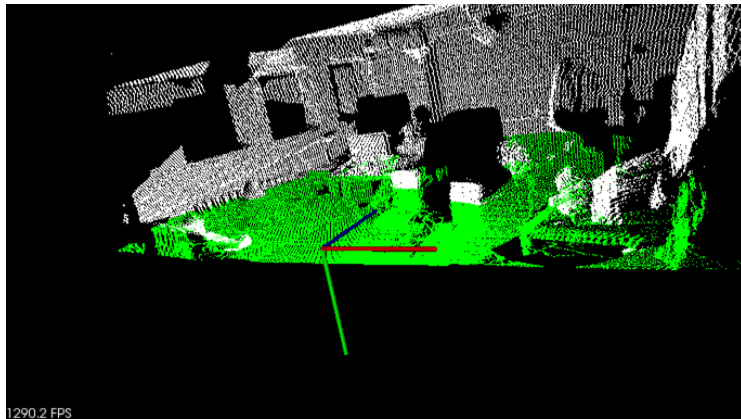


Viewer

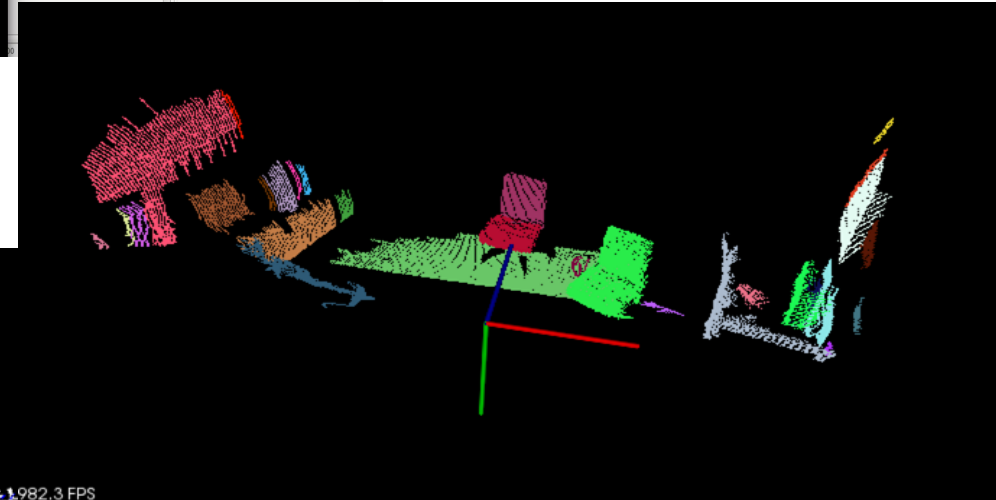


Depth2cloud

Cloud\_filters



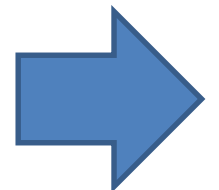
Cloud\_normals



Clustering

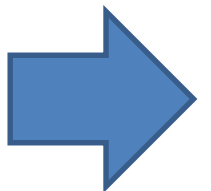
# Compile pcl\_examples

```
bloisi@bloisi-U36SG: ~/workspace/pcl_examples/build
File Edit View Search Terminal Help
bloisi@bloisi-U36SG:~$ cd workspace/pcl_examples/
bloisi@bloisi-U36SG:~/workspace/pcl_examples$ mkdir build
bloisi@bloisi-U36SG:~/workspace/pcl_examples$ cd build/
bloisi@bloisi-U36SG:~/workspace/pcl_examples/build$ cmake ..
-- The C compiler identification is GNU 7.5.0
-- The CXX compiler identification is GNU 7.5.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Checking for module 'eigen3'
-- Found eigen3, version 3.3.4
-- Found eigen: /usr/include/eigen3
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
```



# Compile pcl\_examples

```
bloisi@bloisi-U36SG: ~/workspace/pcl_examples/build
File Edit View Search Terminal Help
-- Configuring done
-- Generating done
-- Build files have been written to: /home/bloisi/workspace/pcl_examples/build
bloisi@bloisi-U36SG:~/workspace/pcl_examples/build$ ls
CMakeCache.txt CMakeFiles cmake_install.cmake Makefile
bloisi@bloisi-U36SG:~/workspace/pcl_examples/build$ make
Scanning dependencies of target cloud_filters
[ 8%] Building CXX object CMakeFiles/cloud_filters.dir/cloud_filters.cpp.o
[ 16%] Linking CXX executable ../bin/cloud_filters
[ 16%] Built target cloud_filters
Scanning dependencies of target depth2cloud
[ 25%] Building CXX object CMakeFiles/depth2cloud.dir/depth2cloud.cpp.o
[ 33%] Linking CXX executable ../bin/depth2cloud
[ 33%] Built target depth2cloud
Scanning dependencies of target cloud_normals
[ 41%] Building CXX object CMakeFiles/cloud_normals.dir/cloud_normals.cpp.o
[ 50%] Linking CXX executable ../bin/cloud_normals
[ 50%] Built target cloud_normals
Scanning dependencies of target viewer
[ 58%] Building CXX object CMakeFiles/viewer.dir/viewer.cpp.o
[ 66%] Linking CXX executable ../bin/viewer
[ 66%] Built target viewer
Scanning dependencies of target clustering
[ 75%] Building CXX object CMakeFiles/clustering.dir/clustering.cpp.o
[ 83%] Linking CXX executable ../bin/clustering
[ 83%] Built target clustering
Scanning dependencies of target icp
[ 91%] Building CXX object CMakeFiles/icp.dir/icp.cpp.o
/home/bloisi/workspace/pcl_examples/icp.cpp: In function 'int main(int, char**)'
:
/home/bloisi/workspace/pcl_examples/icp.cpp:35:28: warning: 'void pcl::Registrat
ion::PrintSource, PrintTarget, GetSource, GetTargetCloud, GetSourceCloud, GetTargetCloud'
```





# Compile pcl\_examples

---

bloisi@bloisi-U36SG: ~/workspace/pcl\_examples/bin

File Edit View Search Terminal Help

```
bloisi@bloisi-U36SG:~/workspace/pcl_examples/build$ cd ..
```

```
bloisi@bloisi-U36SG:~/workspace/pcl_examples$ cd bin
```

```
bloisi@bloisi-U36SG:~/workspace/pcl_examples/bin$ ls
```

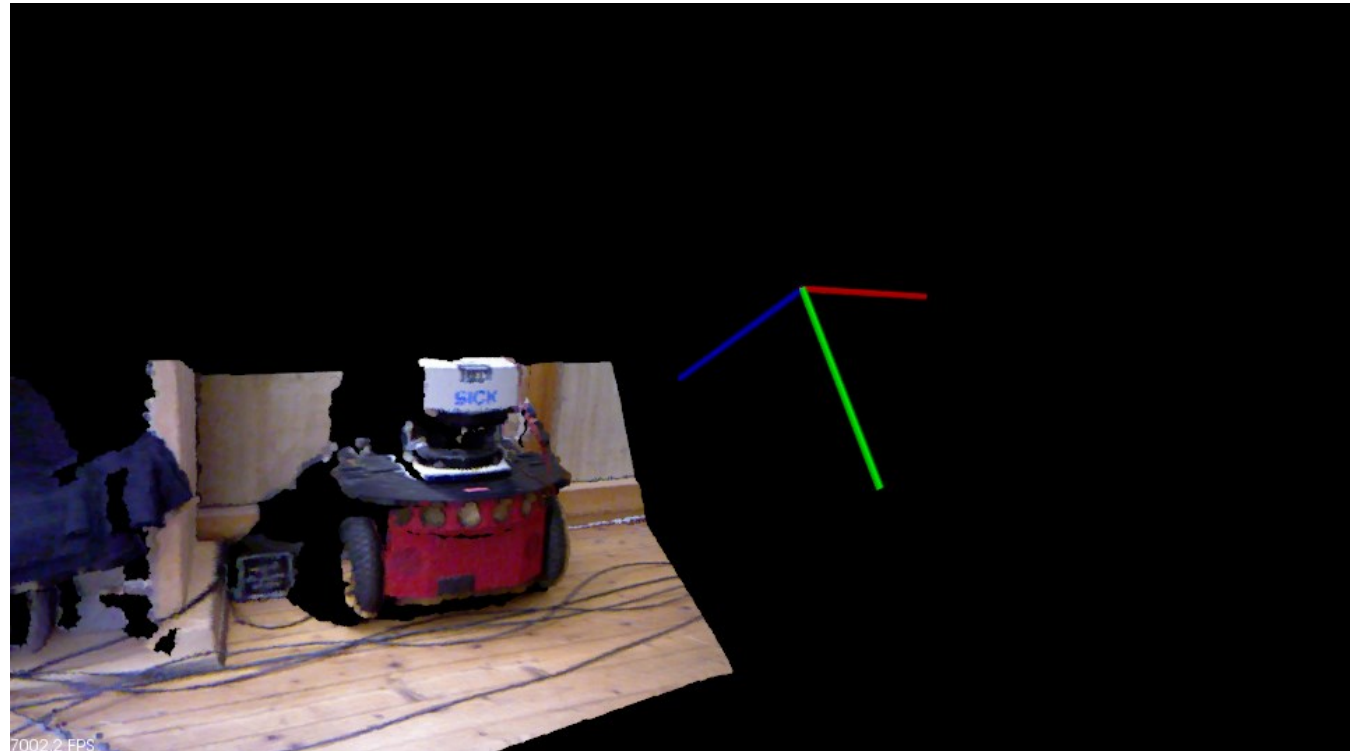
```
cloud_filters  cloud_normals  clustering  depth2cloud  icp  viewer
```

```
bloisi@bloisi-U36SG:~/workspace/pcl_examples/bin$
```

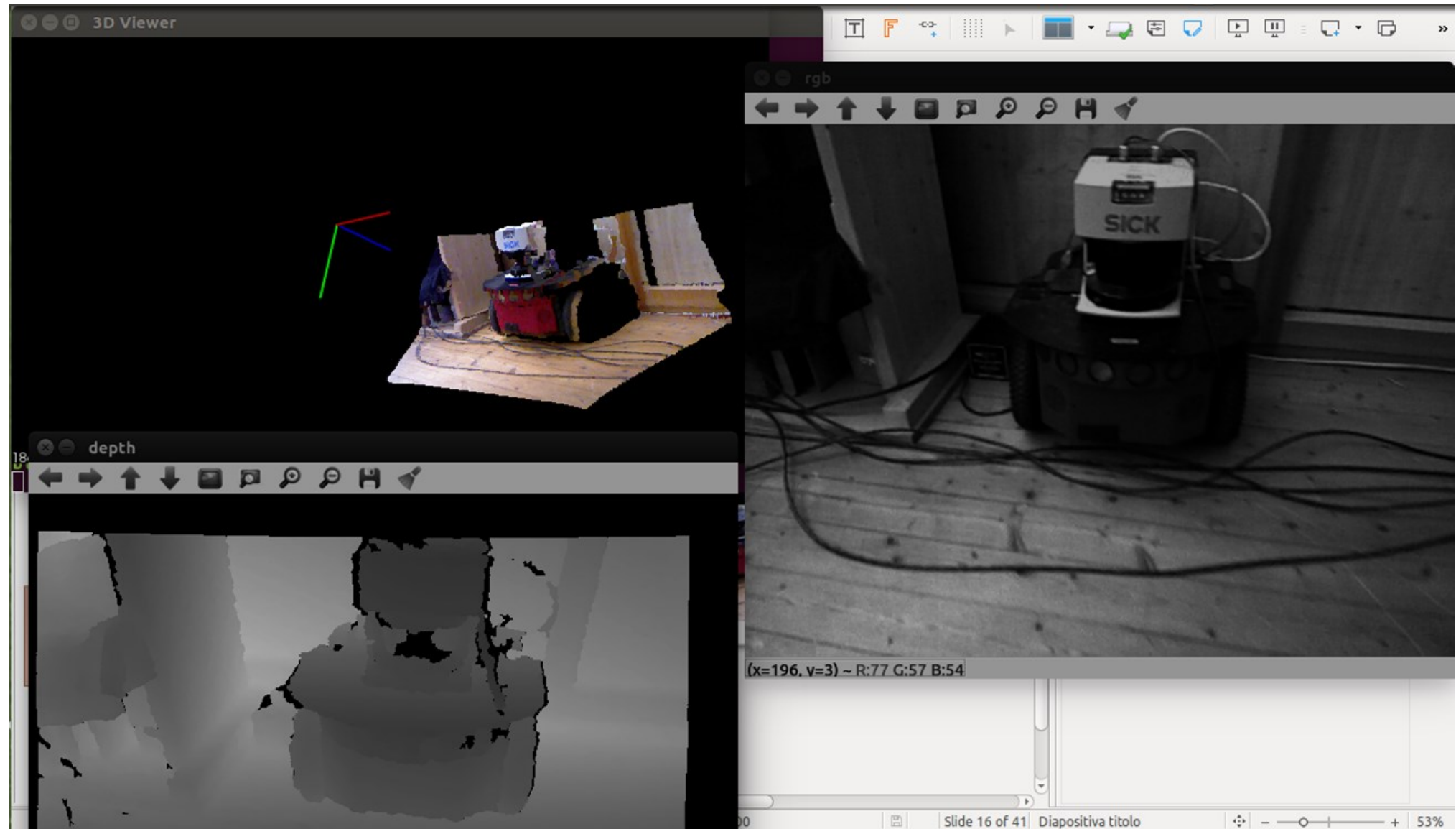
# Visualize a cloud

---

```
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:

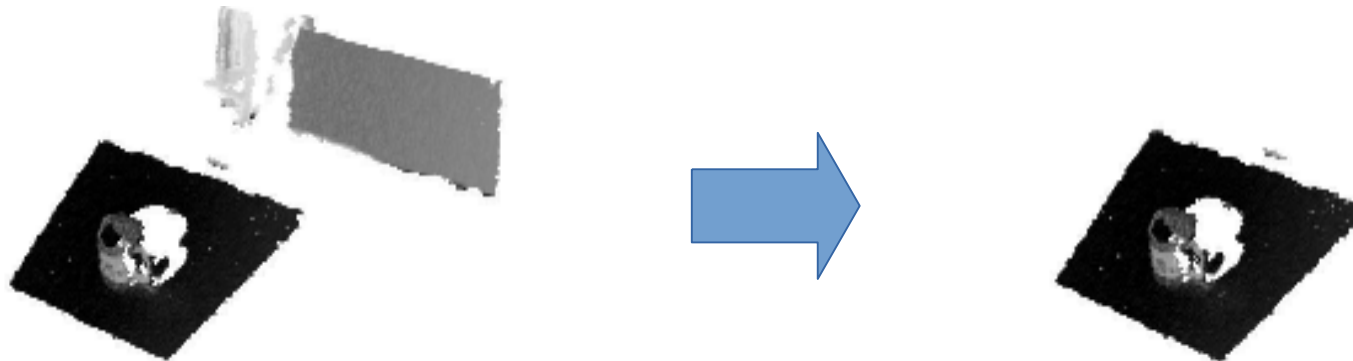
- use **setInputCloud()** to give the input
- set some parameters
- call **compute()** or **filter()** or **align()** or ... to get the output

# PassThrough Filter

---

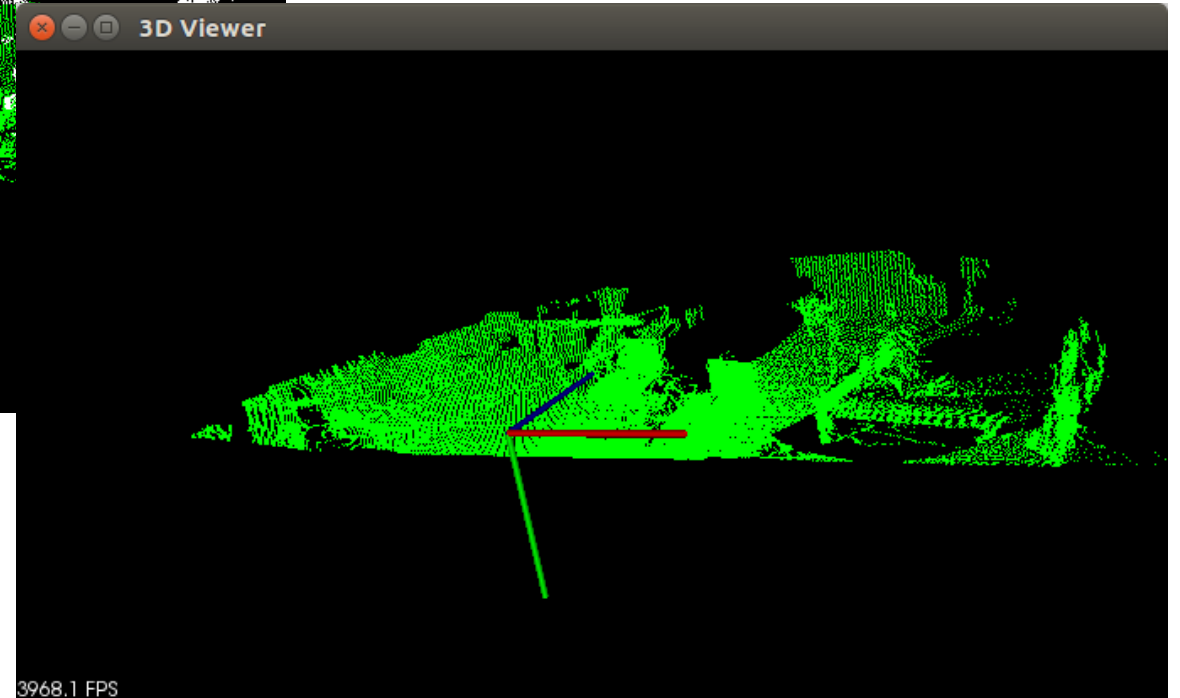
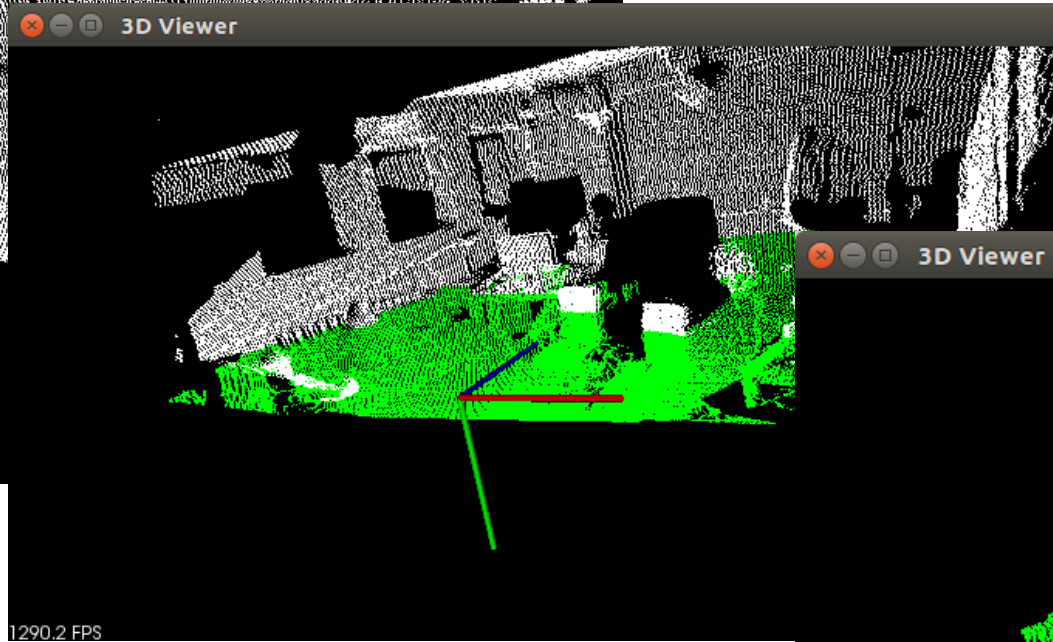
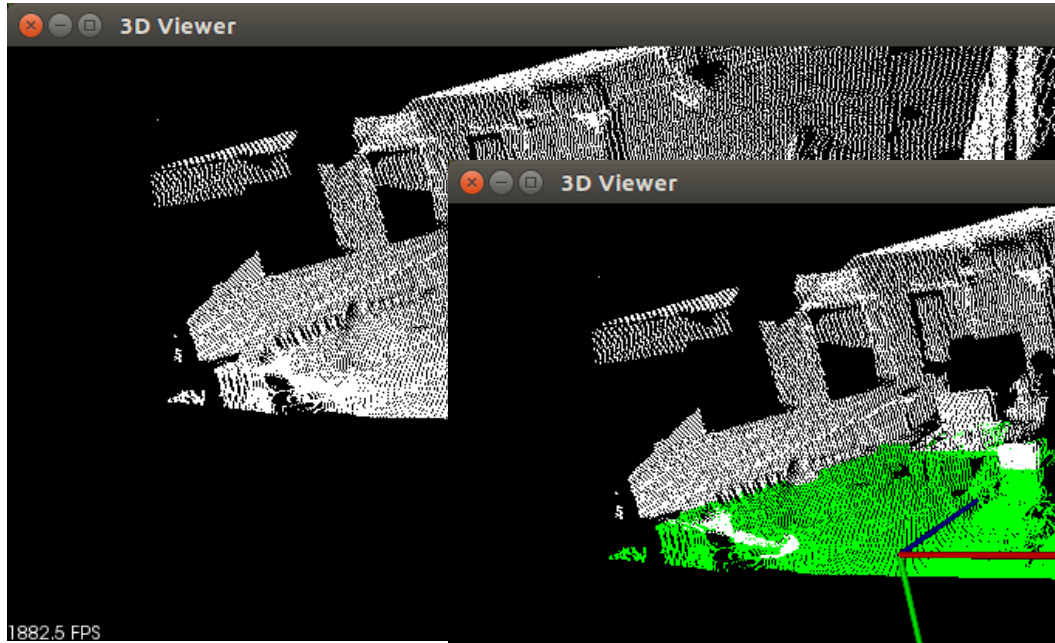
Filter out points outside a specified range in one dimension.

```
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

---



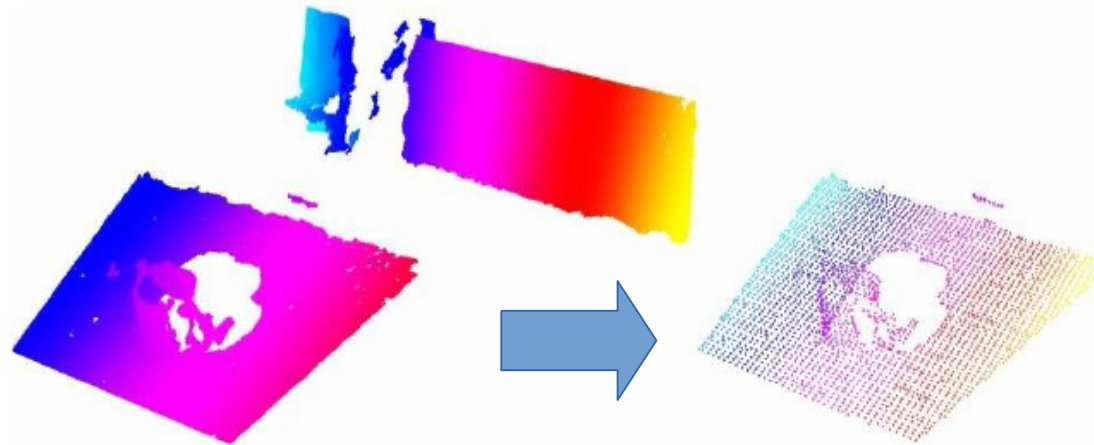


# Downsampling

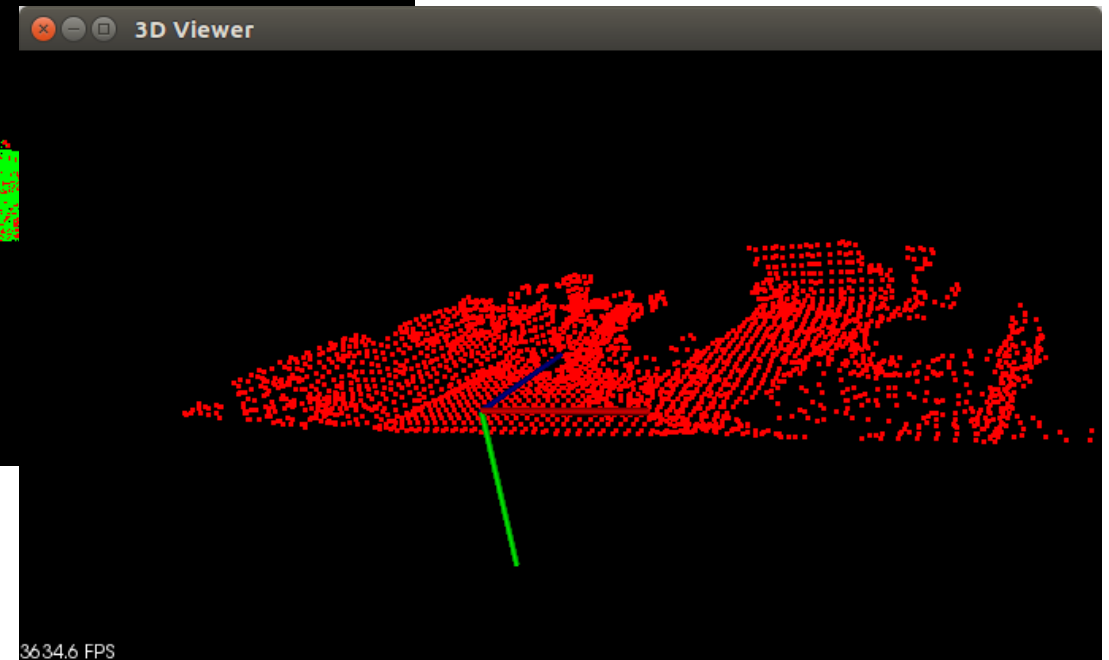
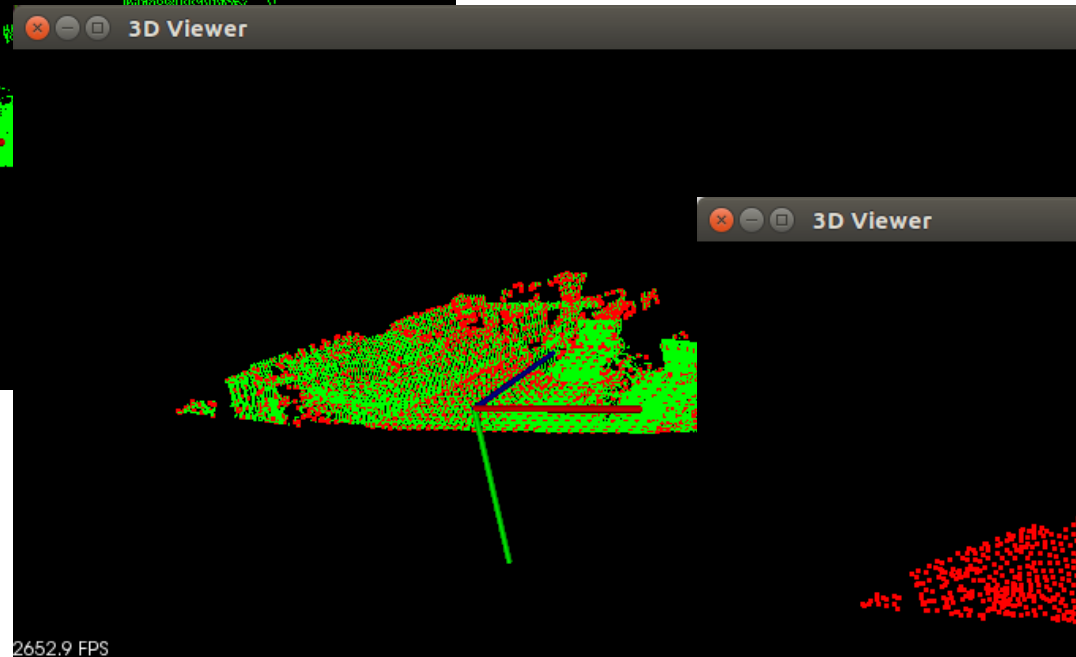
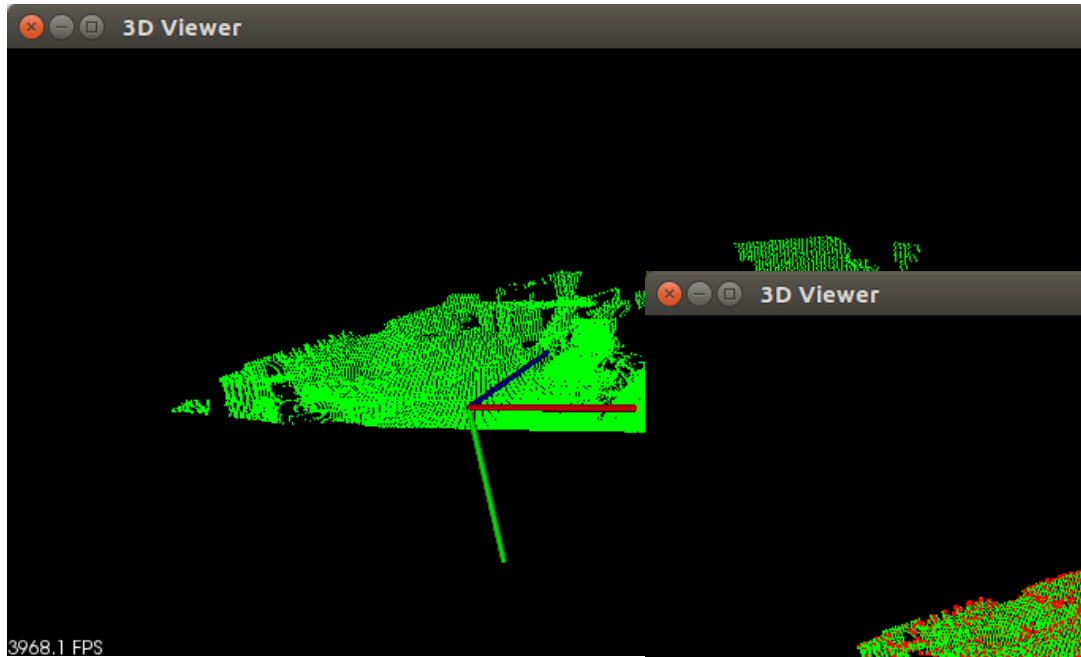
---

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

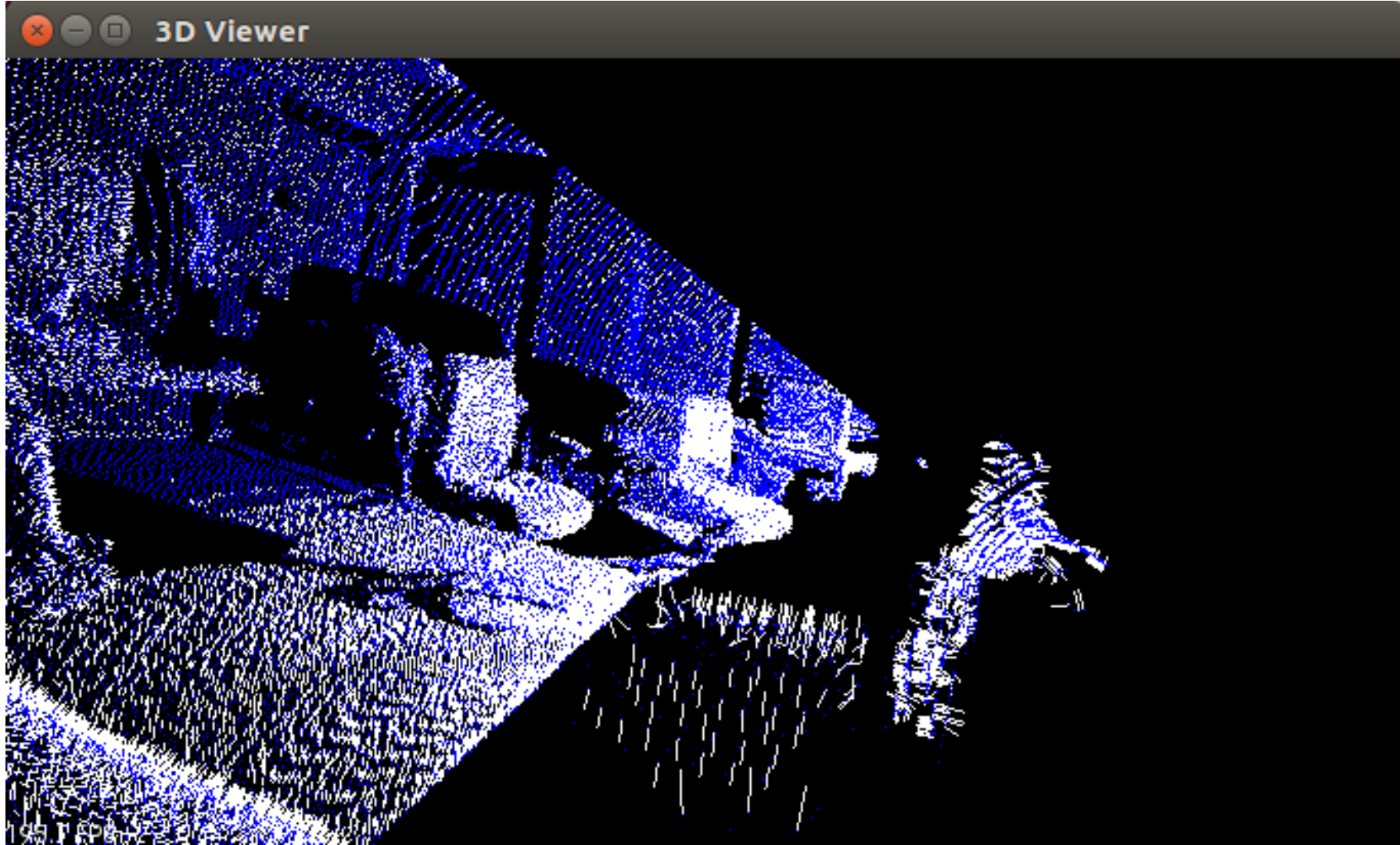
---

```
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

---

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

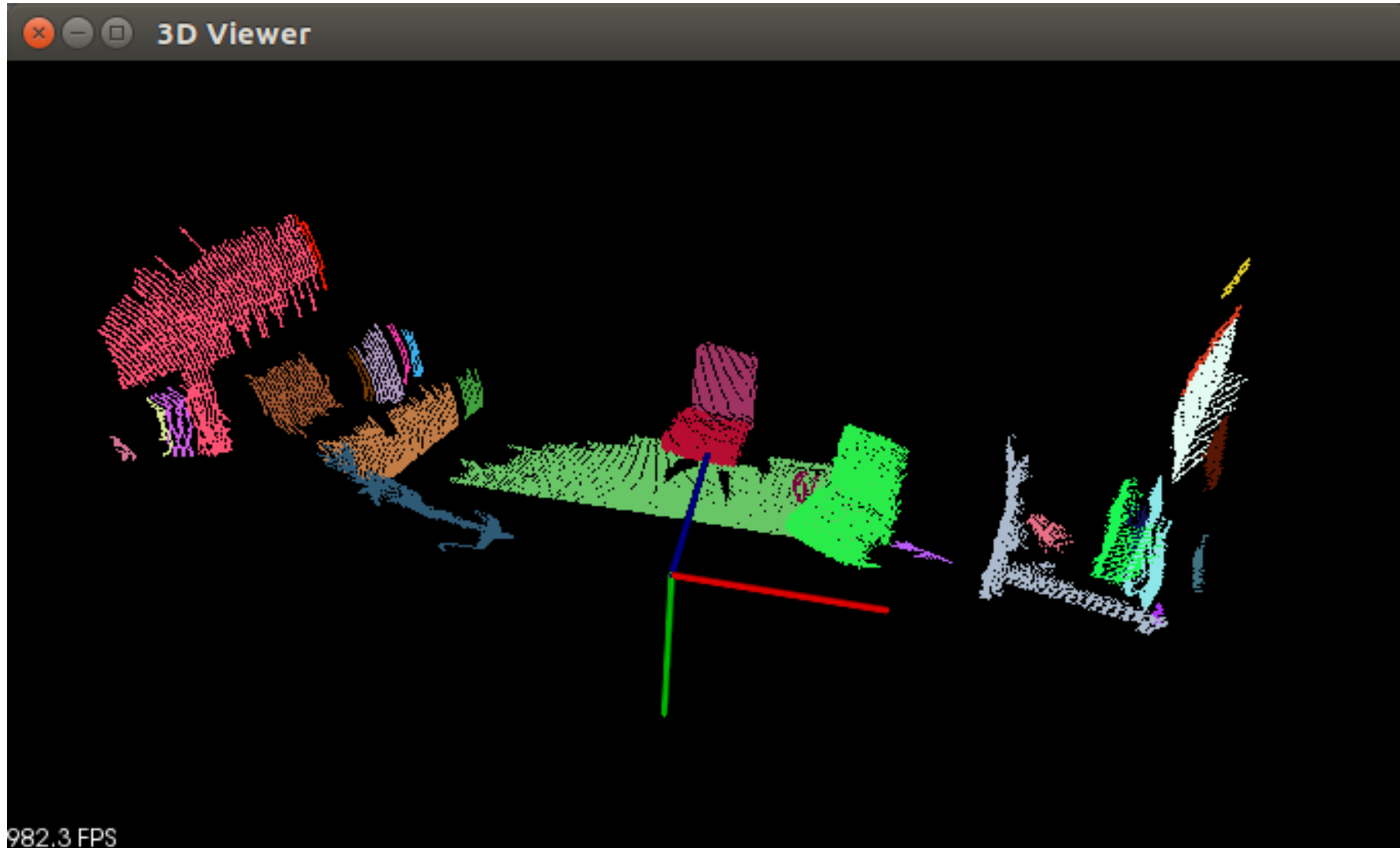
---

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



# clustering.cpp

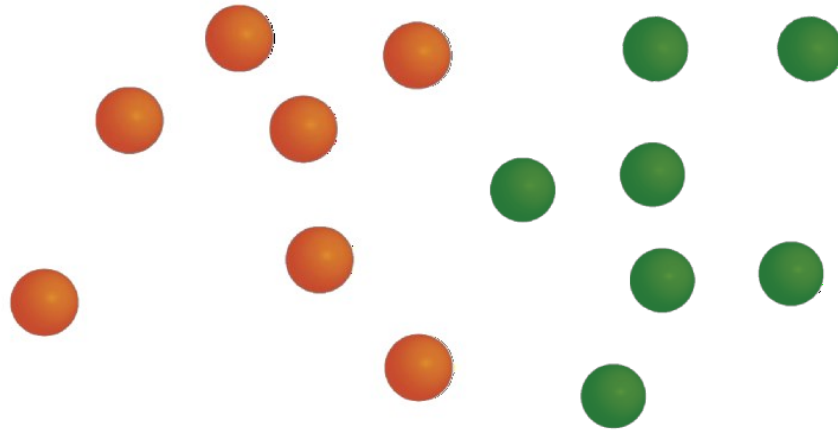
---



# Point Cloud Registration

---

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

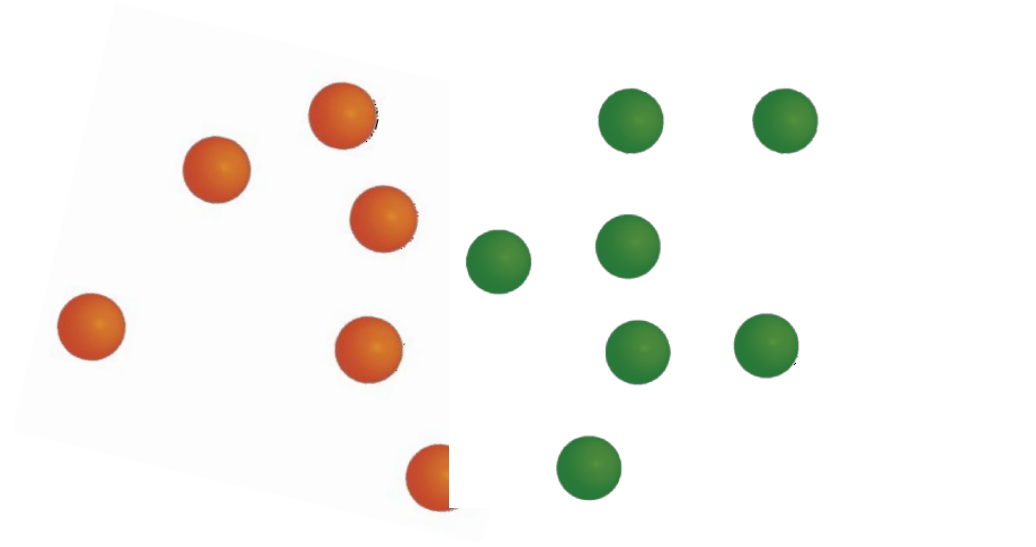




# Point Cloud Registration

---

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

---

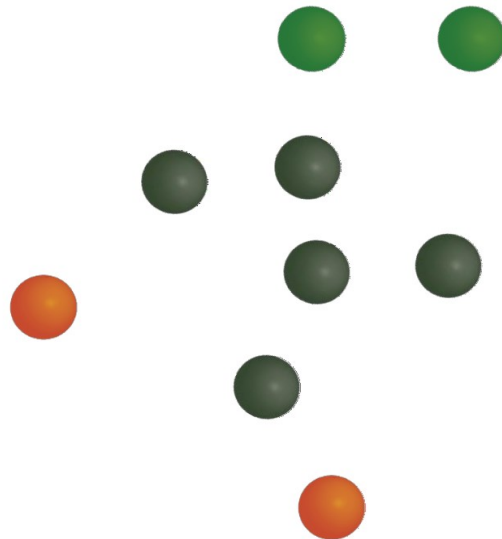
We want to find the translation and the rotation that maximize the overlap between two point clouds



# Point Cloud Registration

---

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



# Iterative Closest Point

---

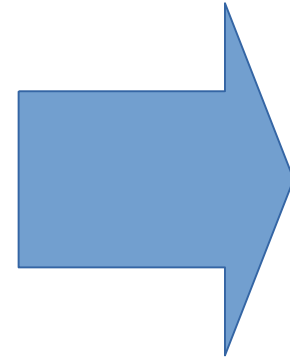
ICP iteratively revises the transformation (translation, rotation) needed to minimize the distance between the points of two raw scans

**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

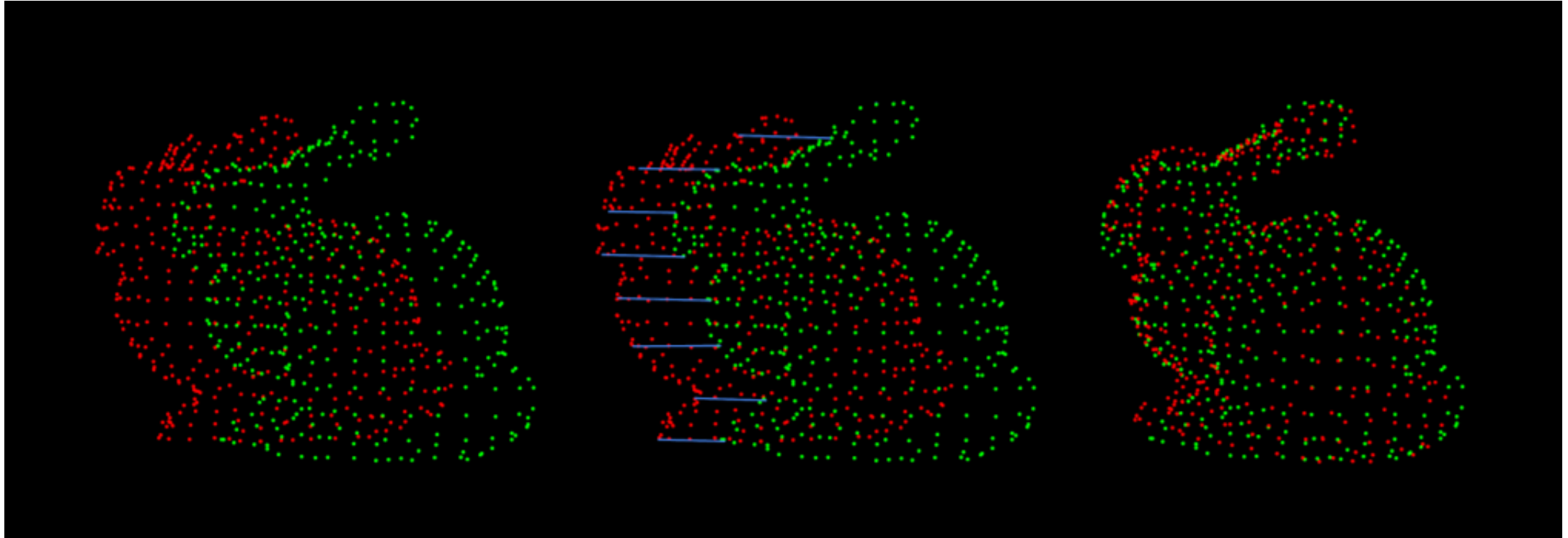
---

```
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);
// Align cloud_source to cloud_source_registered
Eigen::Matrix4f transformation = icp.getFinalTransformation();
```



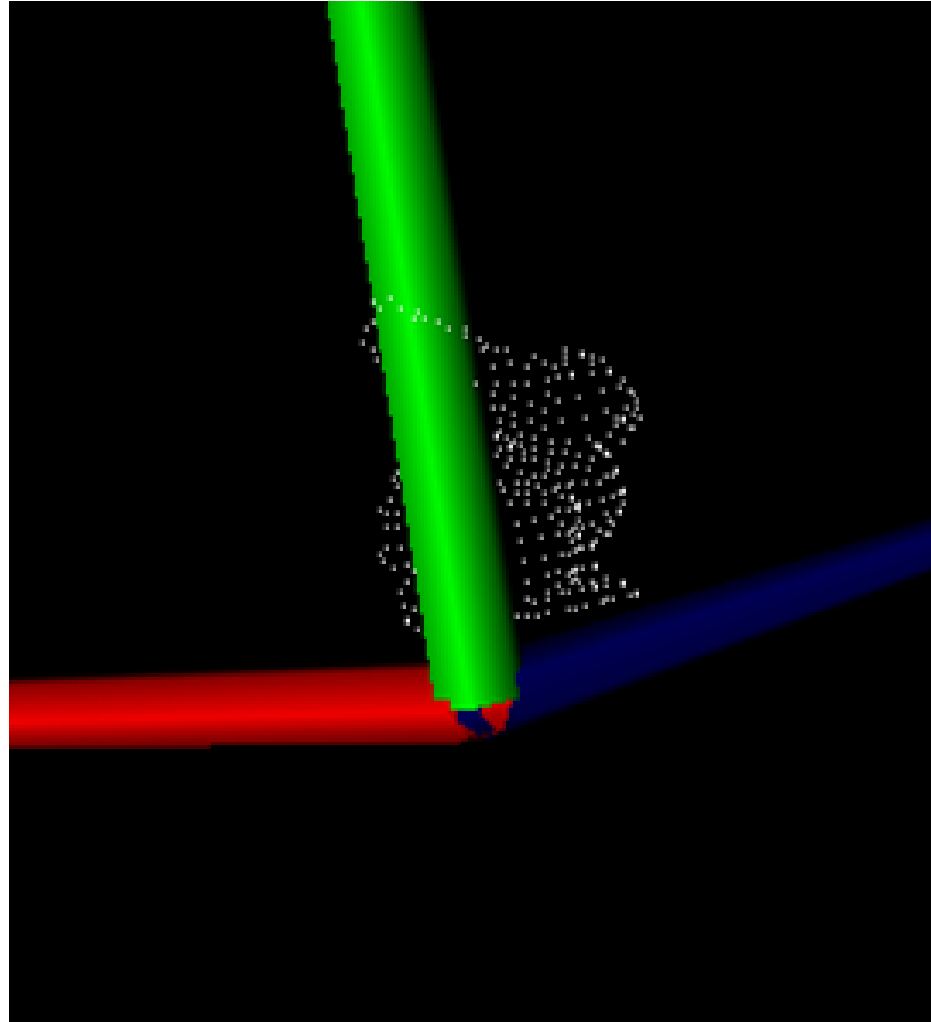
# Iterative Closest Point: Example

---



# icp.cpp

---

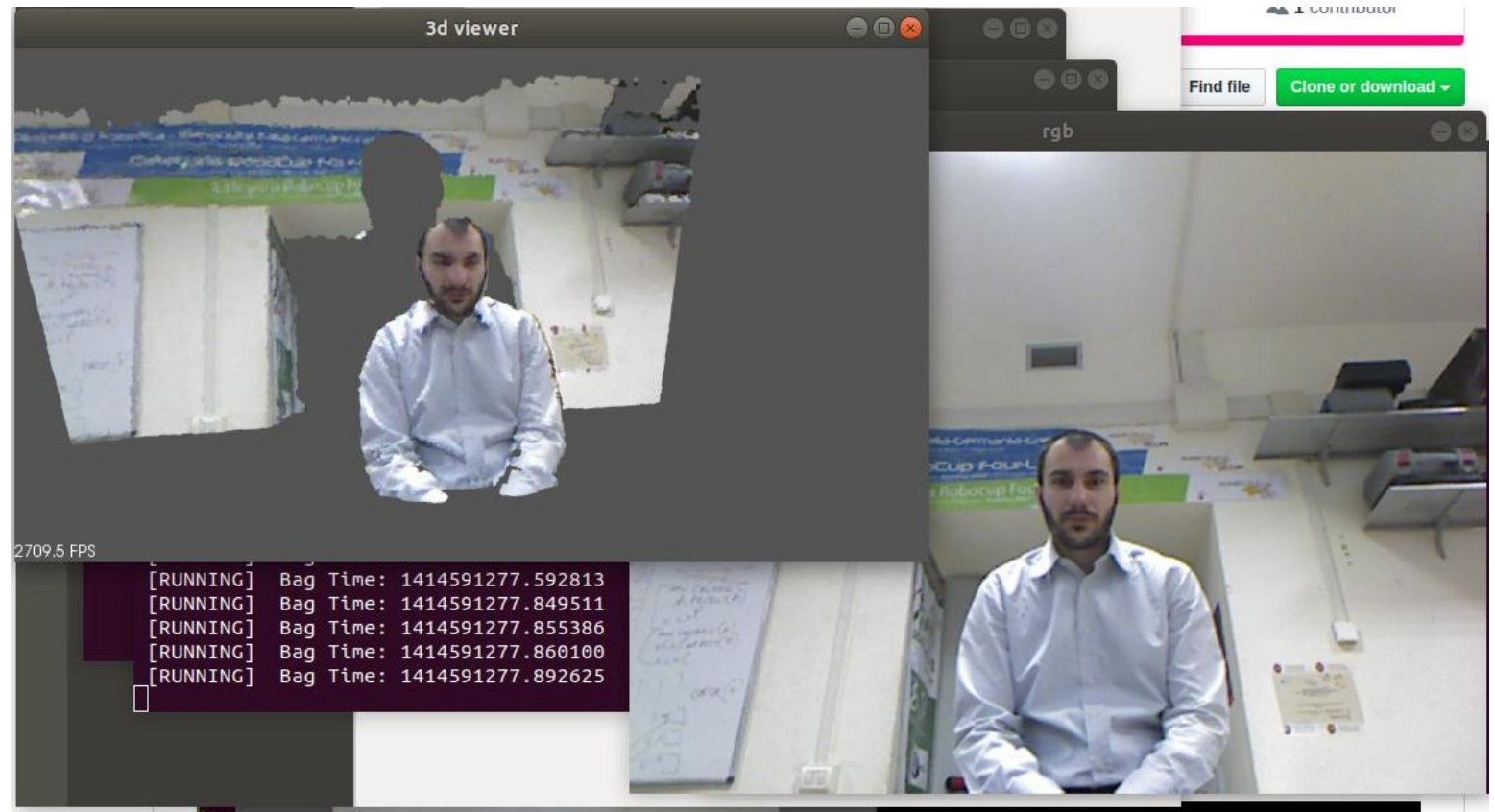


# basic 3D visualizer

A simple 3D visualizer for RGBD data

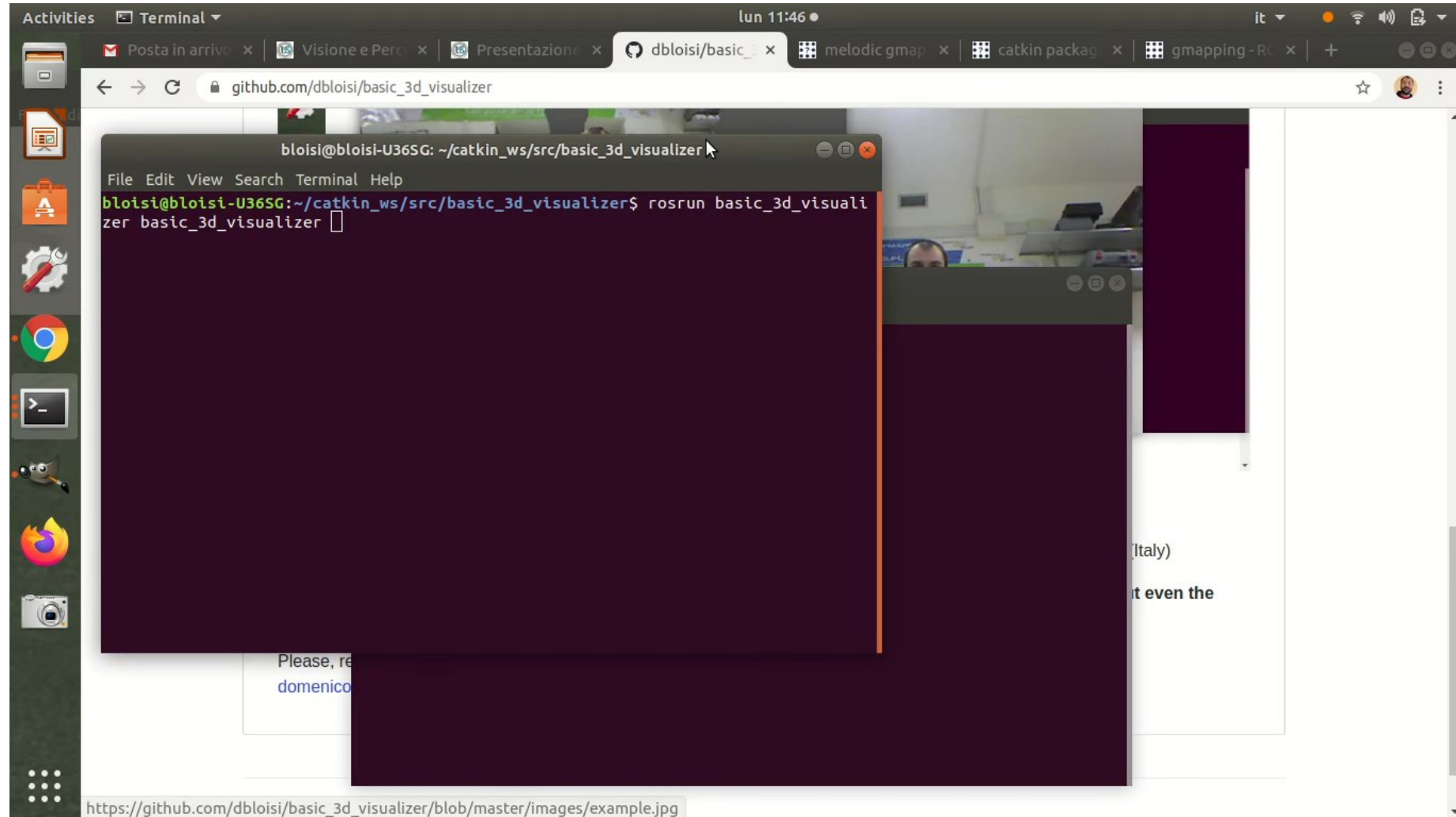
basic\_3d\_visualizer uses

- ROS (tested with melodic)
- OpenCV (tested with version 4.2)
- PCL (tested with version 1.8)

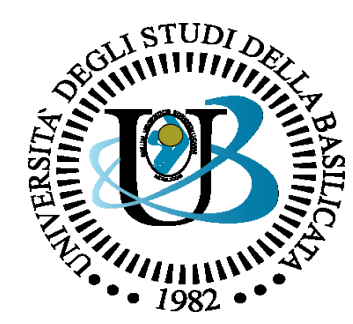


# basic 3D visualizer

---



<https://youtu.be/HTCzk7520Sc>

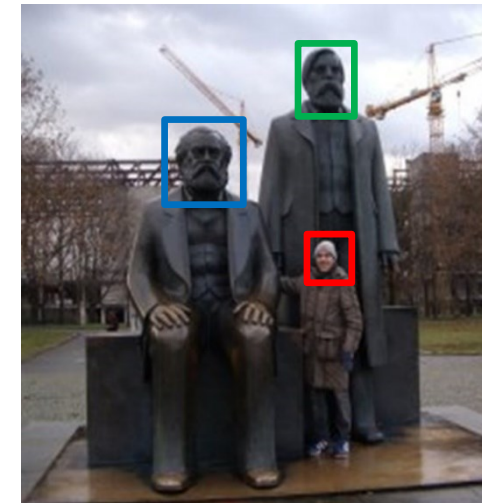
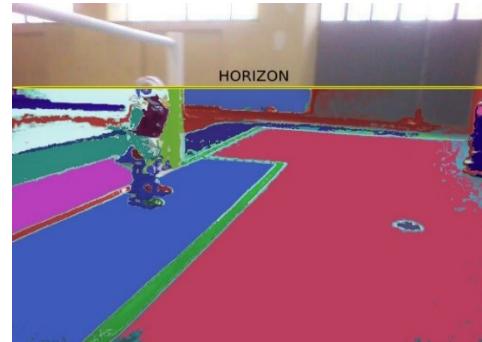
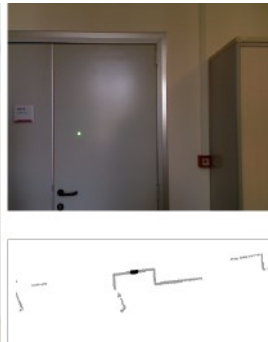
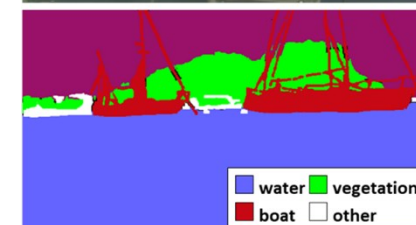
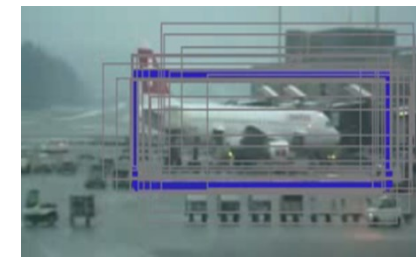


**UNIVERSITÀ DEGLI STUDI  
DELLA BASILICATA**

*Corso di Visione e Percezione*  
*A.A. 2019/2020*

Docente  
**Domenico Daniele Bloisi**

# Visualizzazione dati 3D



Maggio 2020