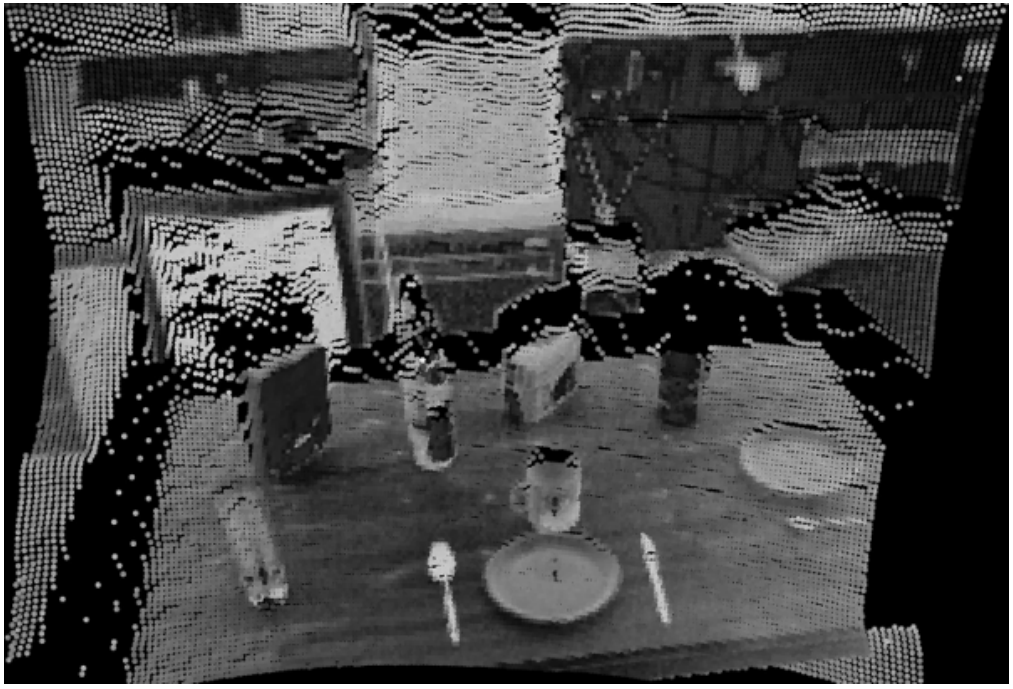
The Point Cloud Library PCL

Bastian Steder

University of Freiburg

Thanks to Radu Rusu from Willow Garage for some of the slides!

What are Point Clouds

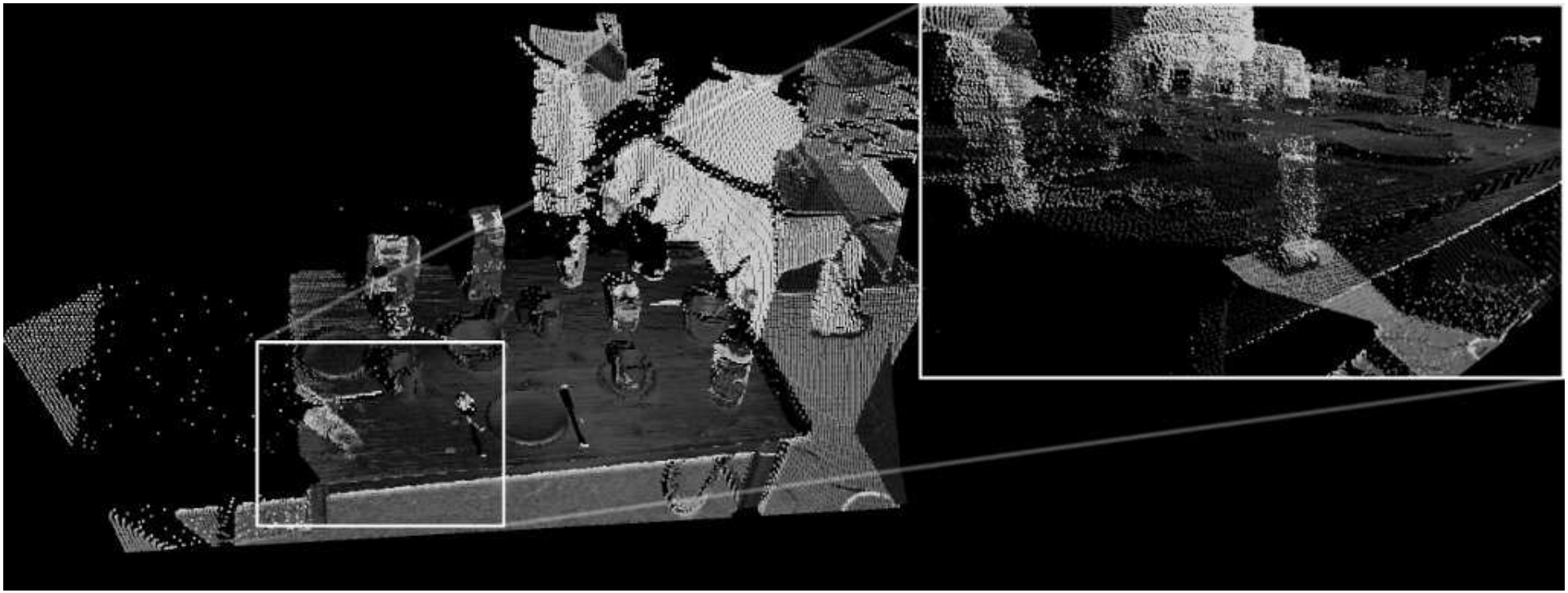


- Point Cloud = a "cloud" (i.e., collection) of nD points (usually n = 3)
- $\mathbf{p}_i = \{x_i, y_i, z_i\} \longrightarrow \mathcal{P} = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_i, \ldots, \mathbf{p}_n\}$
- used to represent 3D information about the world

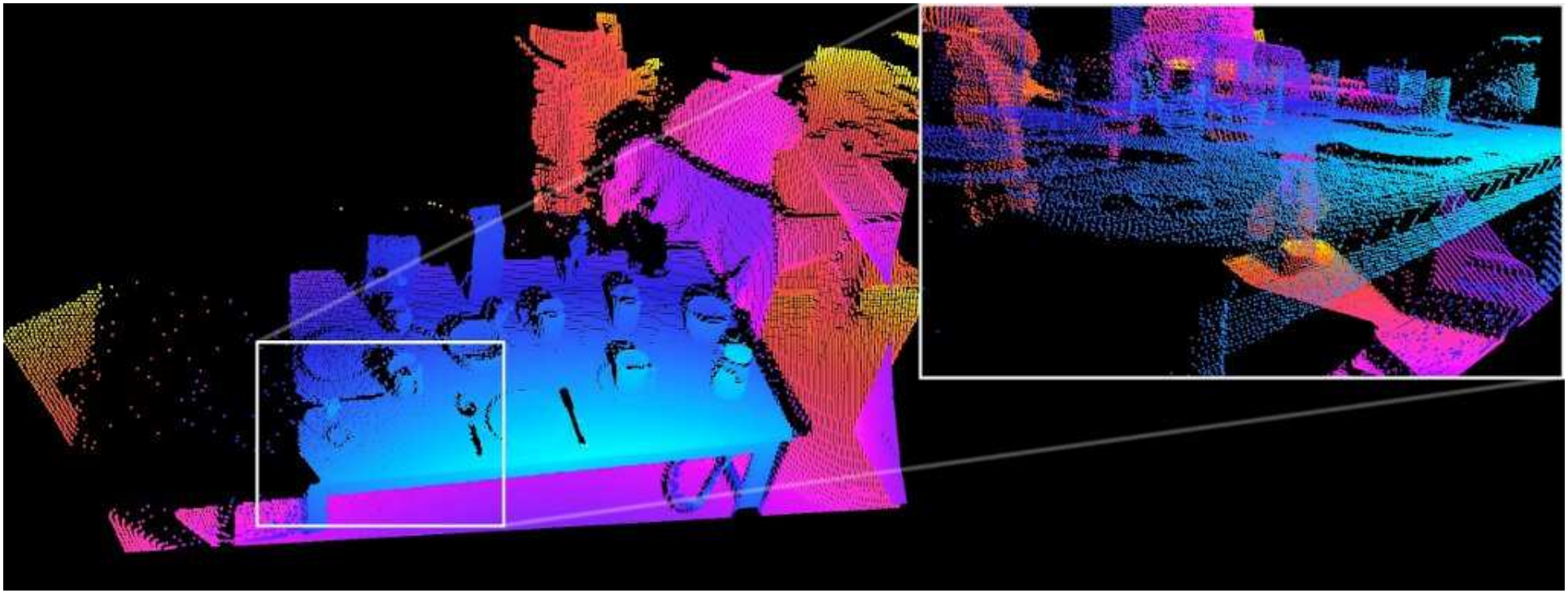
What are Point Clouds

- besides XYZ data, each point $\mathbf{p}$ can hold additional information
- examples include: RGB colors, intensity values, distances, segmentation results, etc…

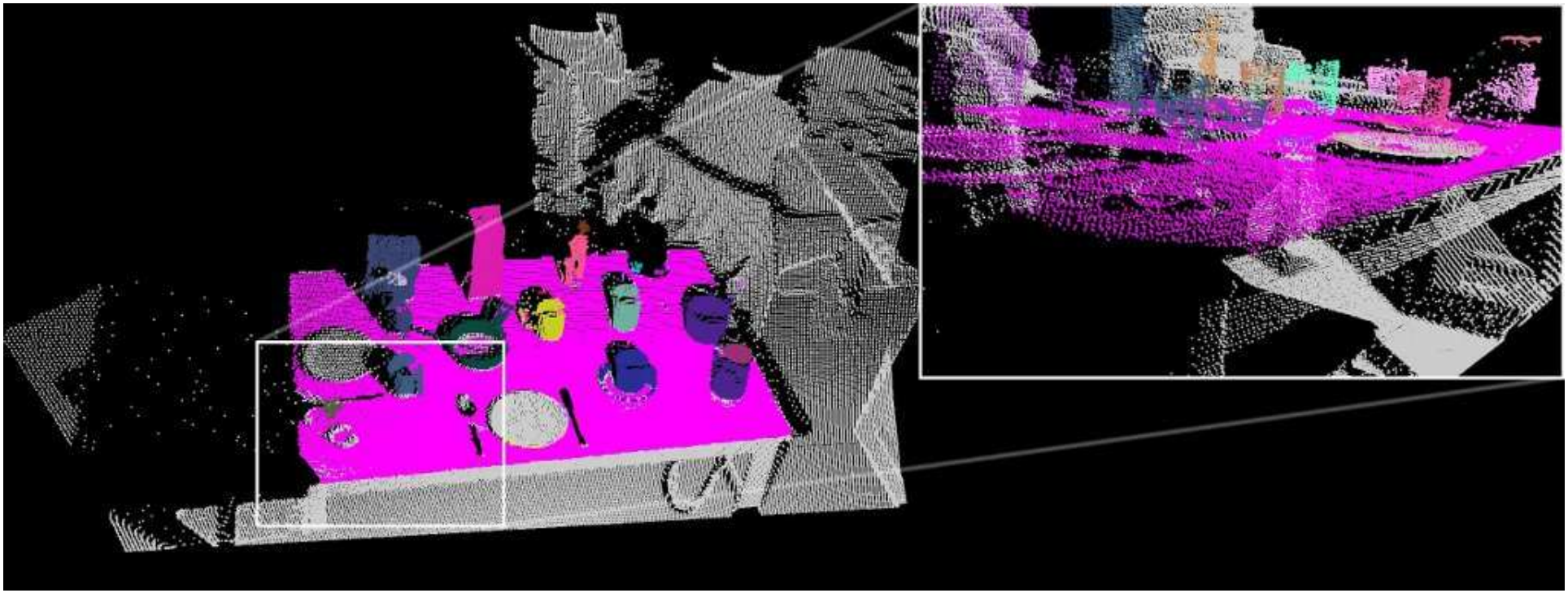
What are Point Clouds



What are Point Clouds



What are Point Clouds

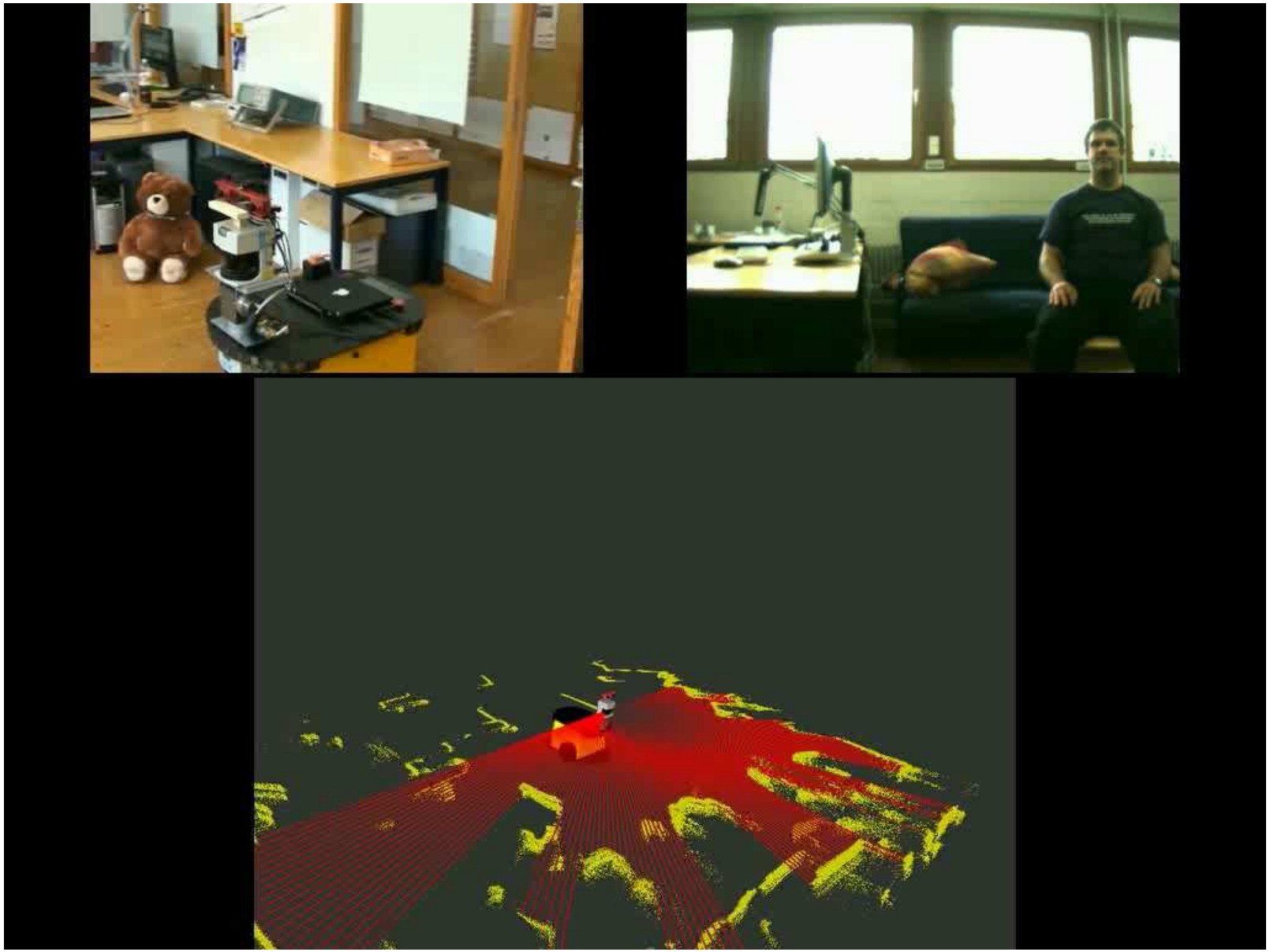


Where do they come from?

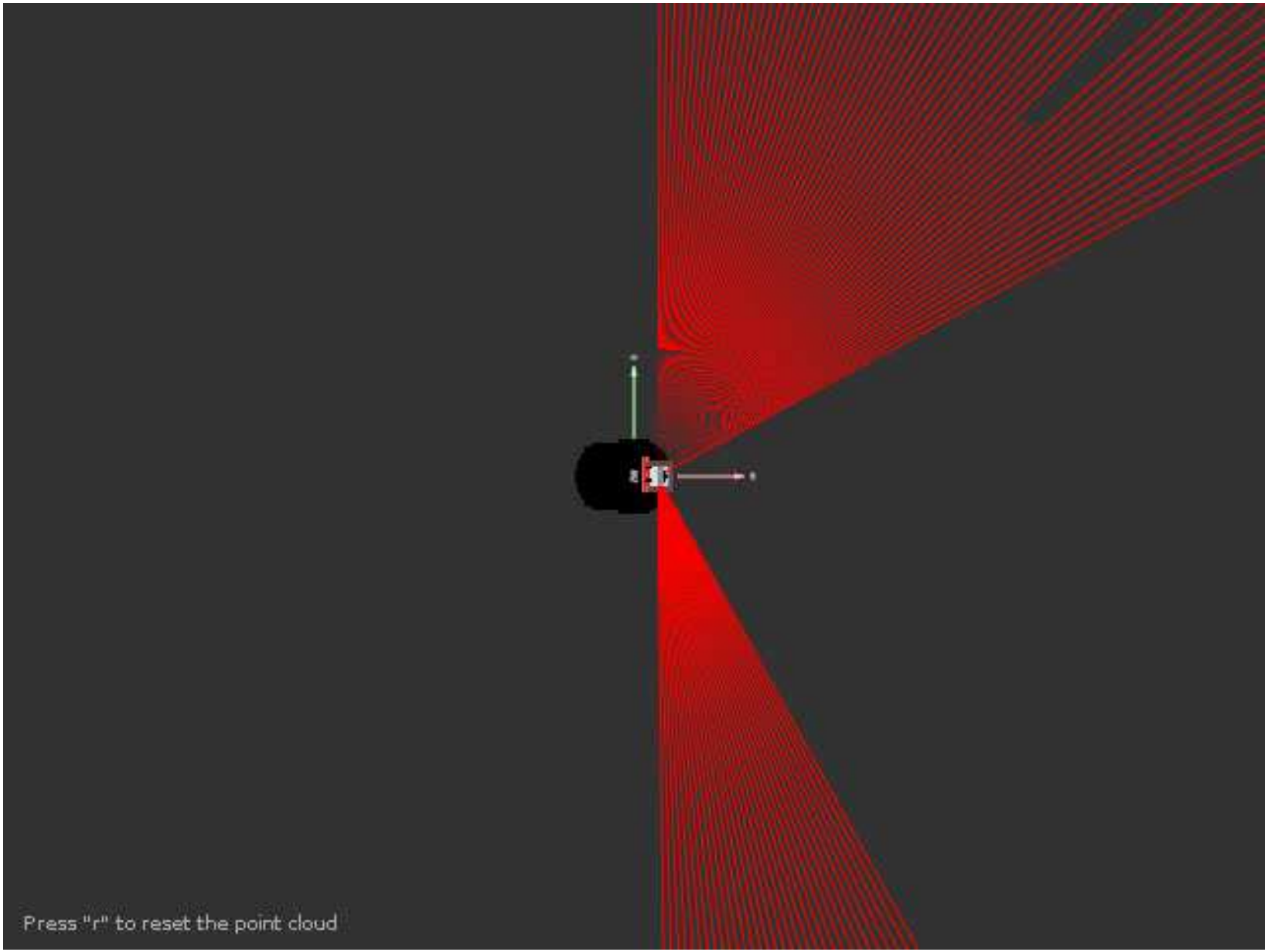
- Laser scans (high quality)
- Stereo cameras (passive & fast but dependent on texture)
- Time of flight cameras (fast but not as accurate/robust)
- Simulation
- …

Tilting Laser Scanner

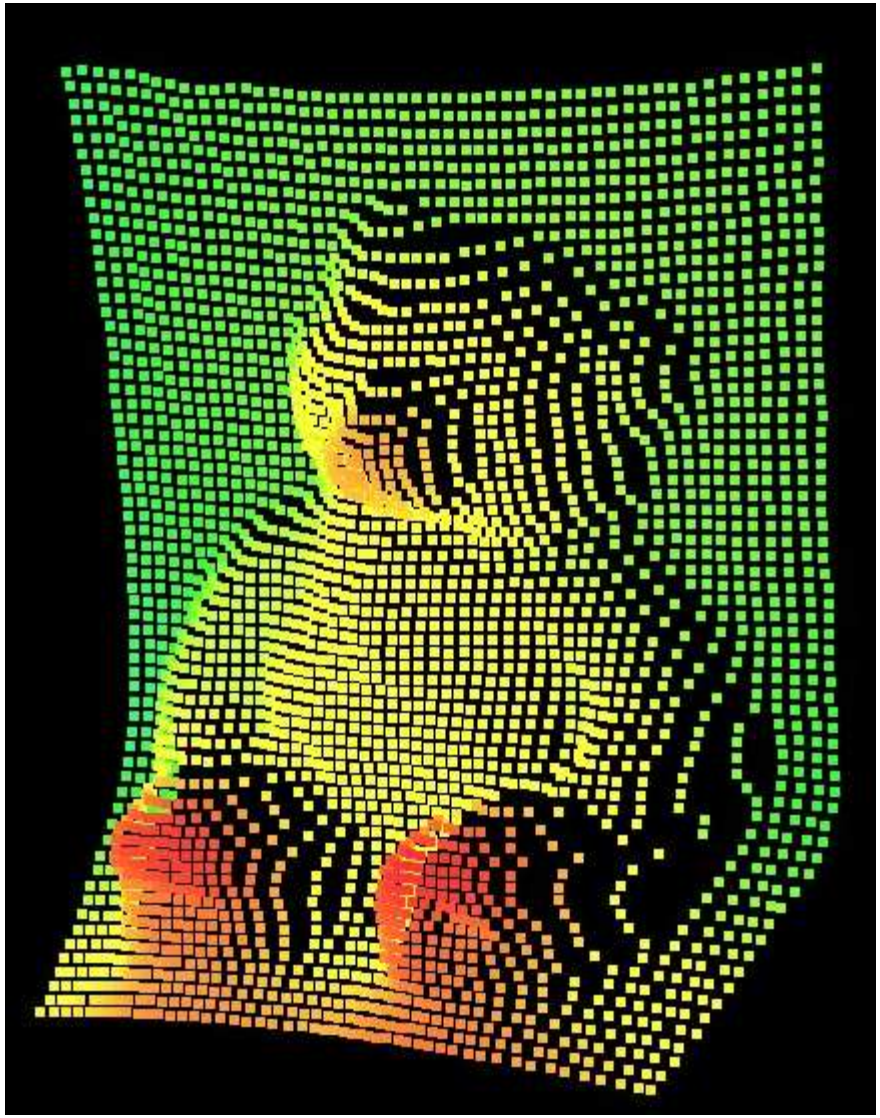
Another example



Tilting scanner on moving robot

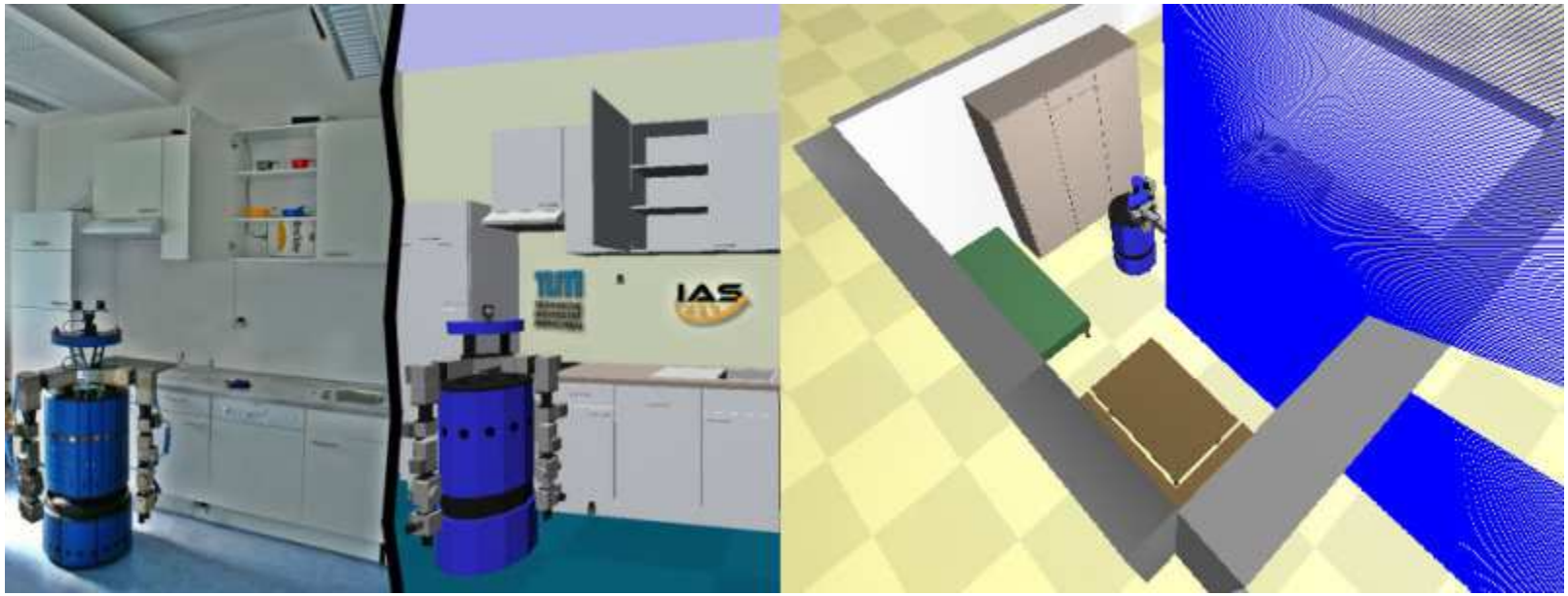


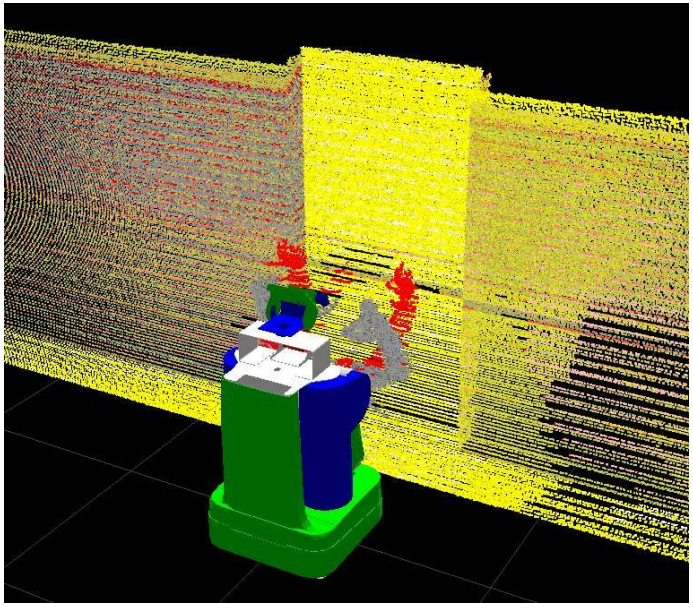
Time of Flight Camera



Stereo camera

Simulation

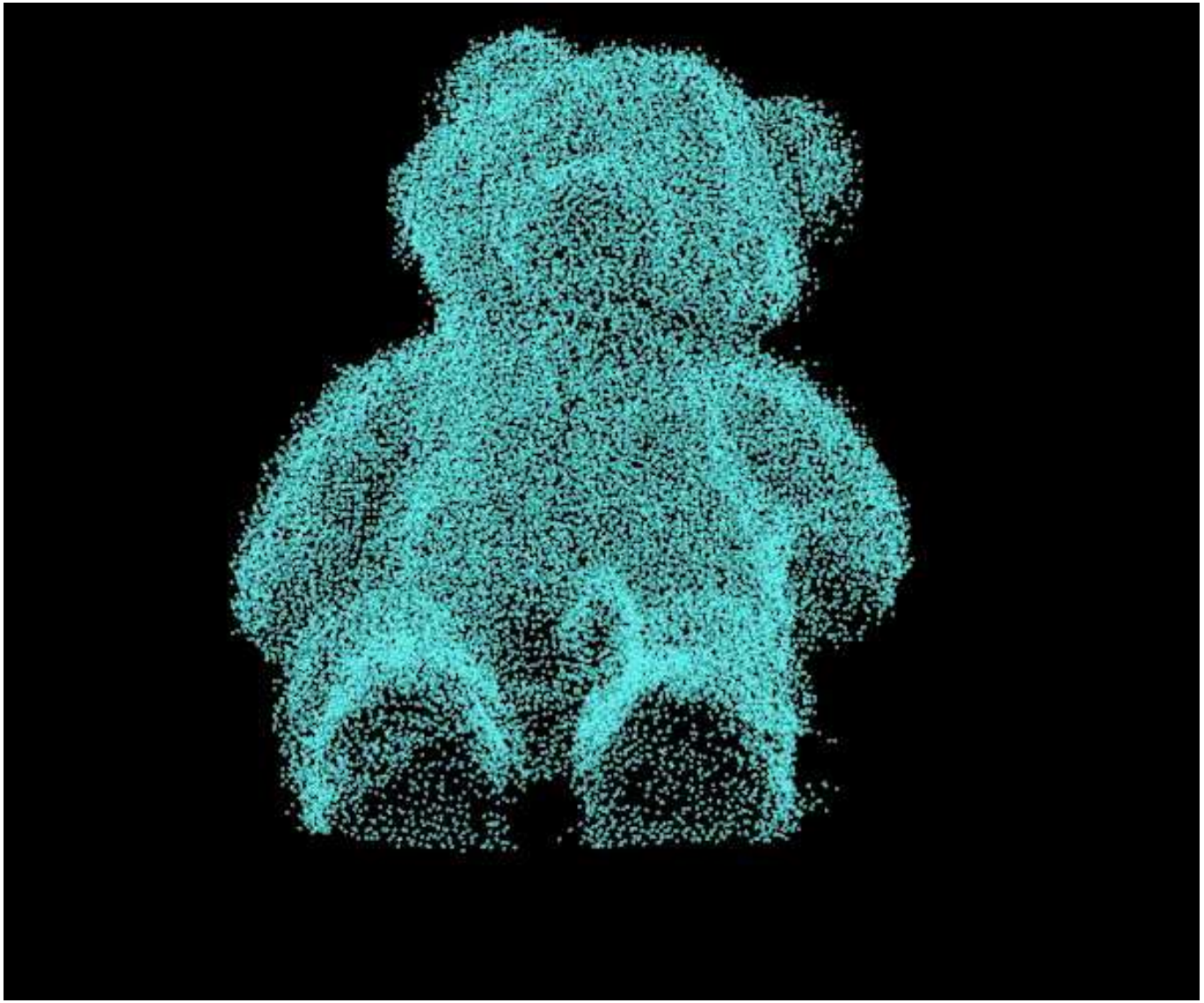




What can you represent

- Single scenes
- Maps
- Object Models
- …

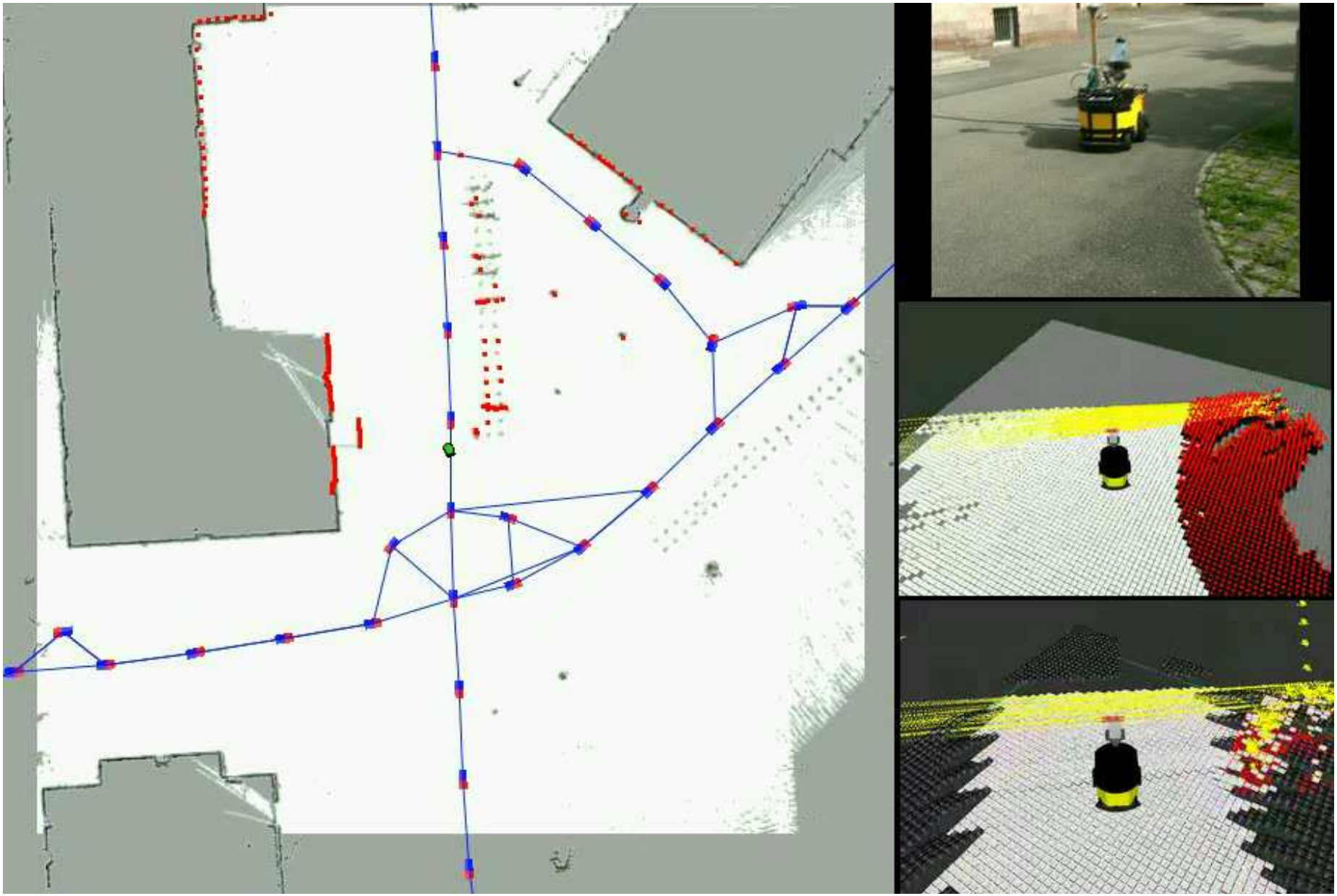
Object models



For what!?

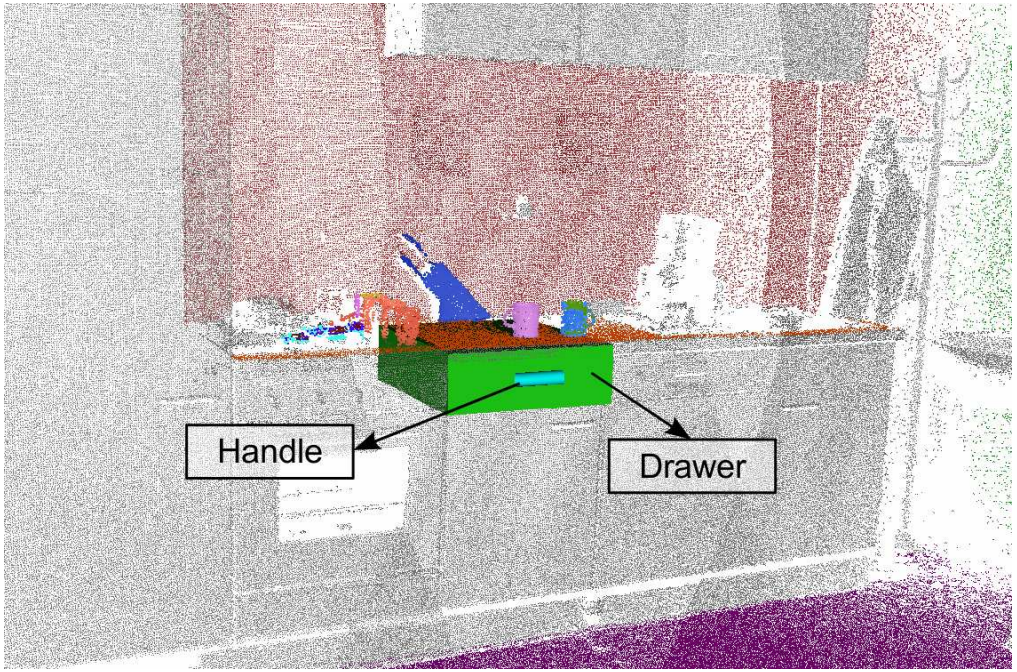
- Spatial information of the environment has many important applications
  - Navigation / Obstacle avoidance
  - Grasping
  - Object recognition
  - …

Obstacle & Terrain detection



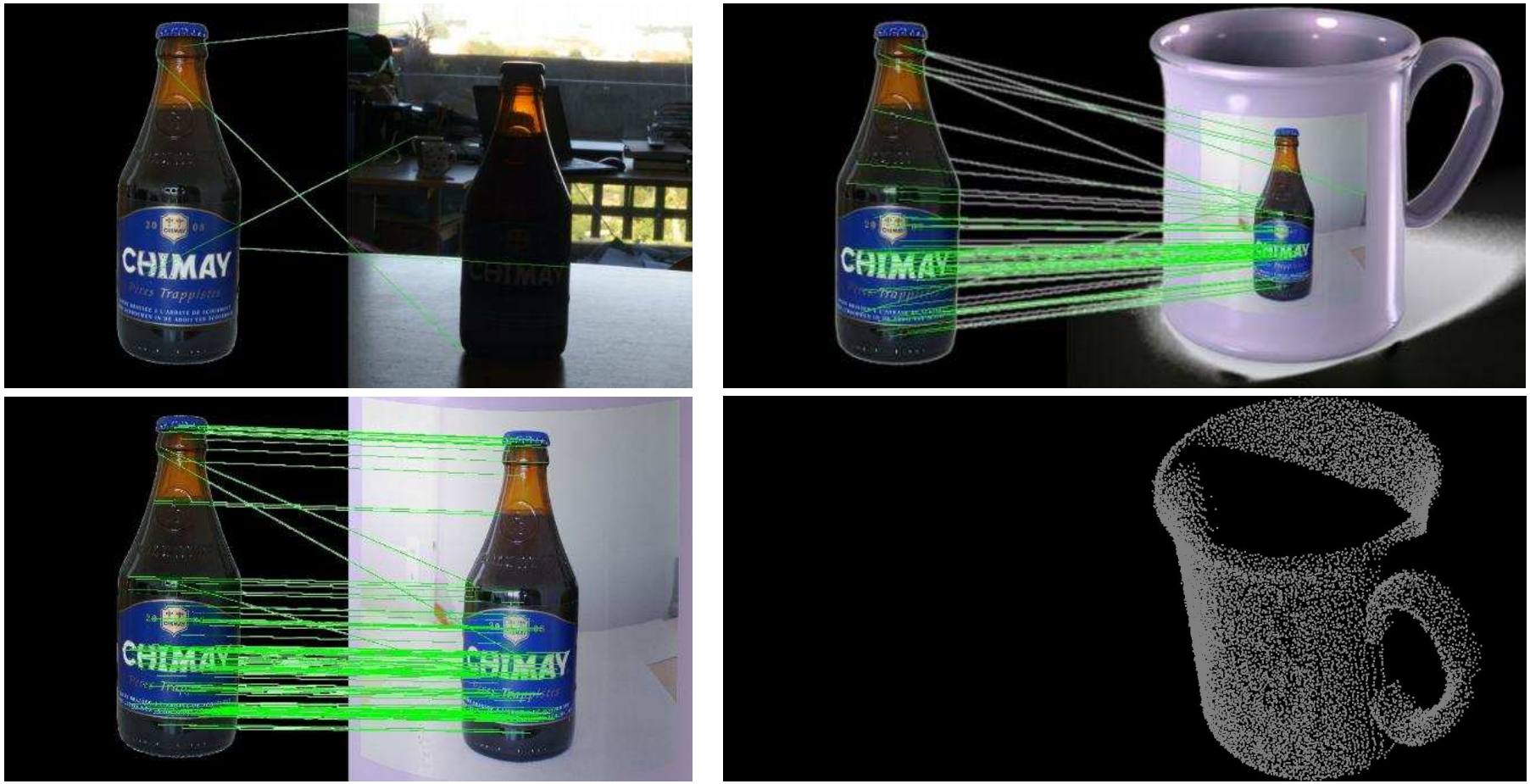
Grasping

Grasping

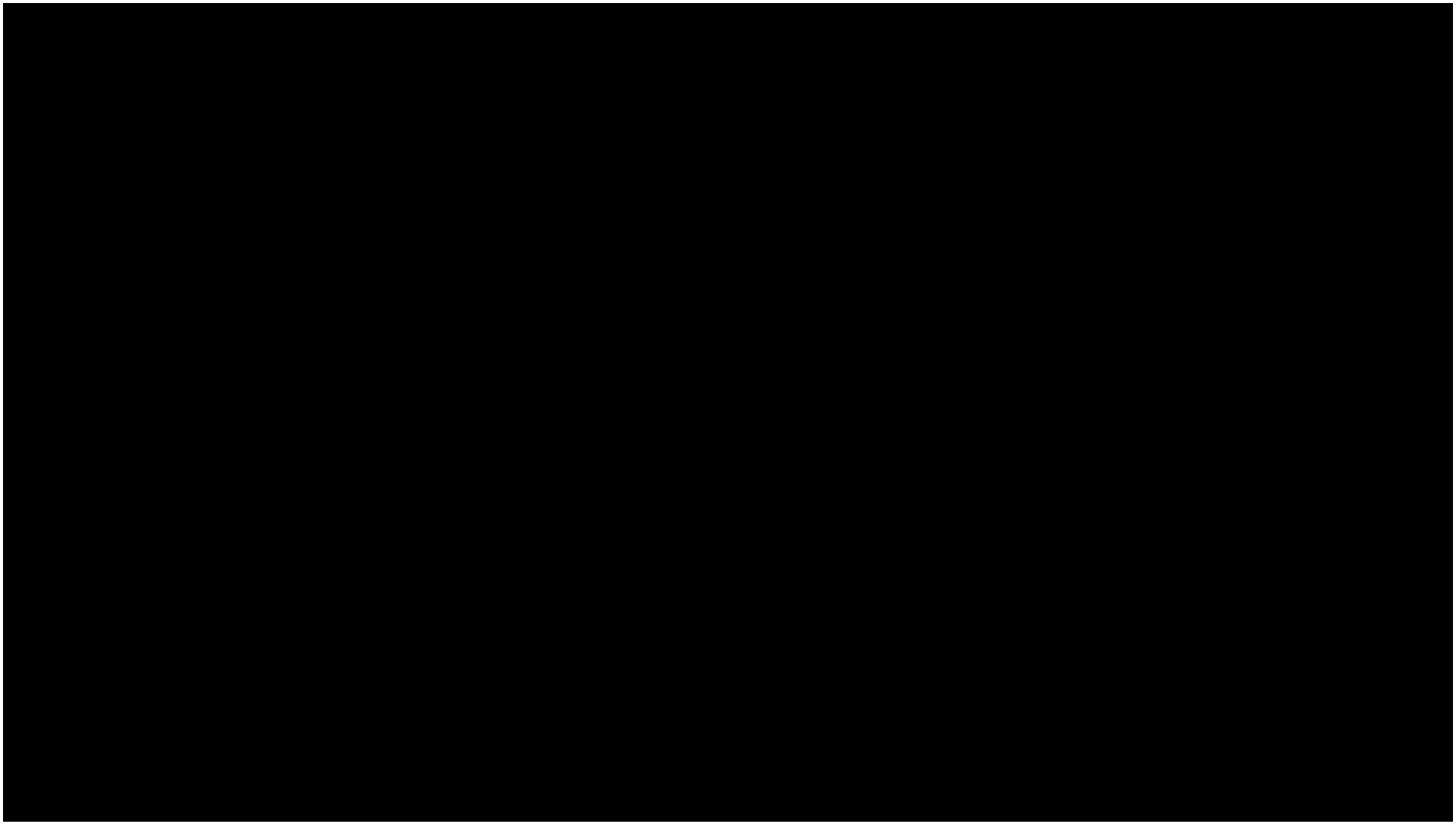


Object Recognition

Point Clouds can complement and supersede images when they are ambiguous.



Object recognition



Data representation

- As previously presented:
  - a point $\mathbf{p}$ is represented as a tuple, e.g., $\mathbf{p}_i = \{x_i, y_i, z_i, r_i, g_i, b_i, dist_i, ...\}$
  - a Point Cloud $\mathcal{P}$ is represented as a collection of points $\mathbf{p}_i$: $\mathcal{P} = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_i, \ldots, \mathbf{p}_n\}$

Data representation

- In terms of data structures:
  - an XYZ point can be represented as:

    ```
    float x
    float y
    float z
    ```

  - a n-dimensional point can be represented as `float[] point`,

    which is nothing else but a

    `std::vector<float> point`

    in C++

Data representation

- In terms of data structures:
  - a point cloud $\mathcal{P}$ is:

    `Point[] points`

    or:

    `std::vector<Point> points`

    in C++, where **`Point`** is the structure/data type representing a single point p

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 (for loading, saving or sending as a ROS message)

Conversion between the two is easy:

`pcl::fromROSMsg` and **`pcl::toROSMsg`**

PointCloud structure

- PointCloud class (templated over the point type):

```
template <typename PointT>
class PointCloud;
```

- Important members:

```
std::vector<PointT> points;  // the data
uint32_t width, height;      // scan structure?
```

- Handles data alignment for proper usage of SSE (for libeigen).

Point types

- Examples of PointT:

```
struct PointXYZ
{
  float x;
  float y;
  float z;
}
```

or

```
struct Normal
{
  float normal[3];
  float curvature;
}
```

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

Point Cloud file format

Point clouds can be stored to disk as files, into the PCD 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 can be either ascii or binary. If ascii, then

```
...
DATA ascii
0.0054216 0.11349 0.040749 …
-0.0017447 0.11425 0.041273 …
-0.010661 0.11338 0.040916 …
0.026422 0.11499 0.032623 …
...
```

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

What is PCL?

PCL

- is a fully templated modern C++ library for 3D point cloud processing
- uses SSE optimizations (Eigen backend) for fast computations on modern CPUs
- uses OpenMP and Intel TBB for parallelization
- passes data between modules (e.g., algorithms) using Boost shared pointers
- will be made independent from ROS in one of the next releases

PCL structure

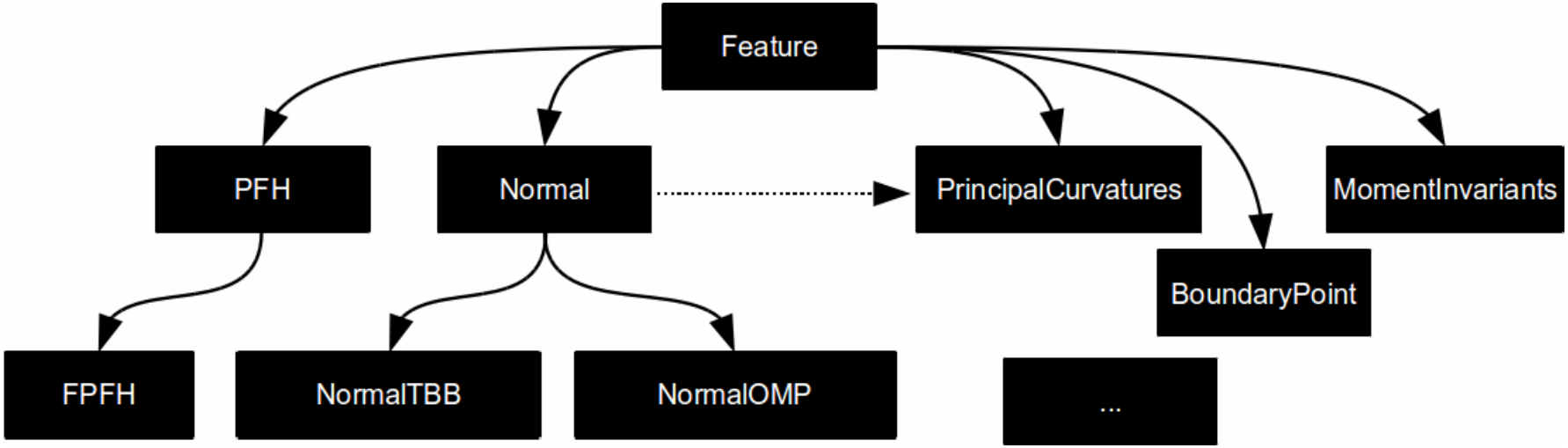
PCL is a collection of smaller, modular C++ libraries:

- 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, …

Architecture

Inheritance simplifies development and testing:




```
pcl::Feature<PointT> feat;
feat = pcl::Normal<PointT> (input);
feat = pcl::FPFH<PointT> (input);
feat = pcl::BoundaryPoint<PointT> (input);
...
feat.compute (&output);
...
```

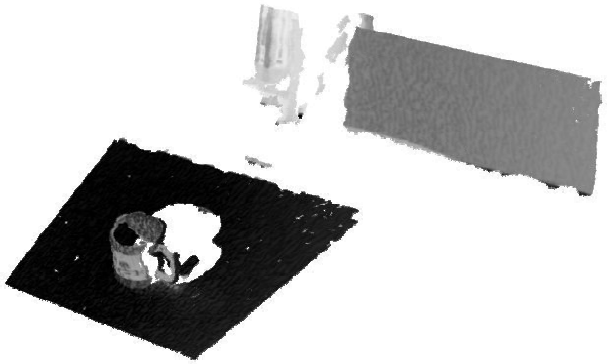
Basic Interface

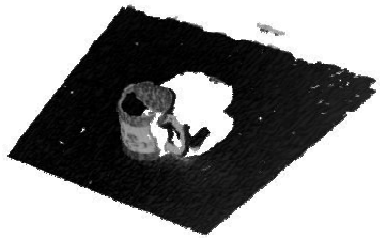
Filters, Features, Segmentation all use the same basic usage interface:

- use `setInputCloud` to give the input
- set some parameters
- call `compute` to get the output

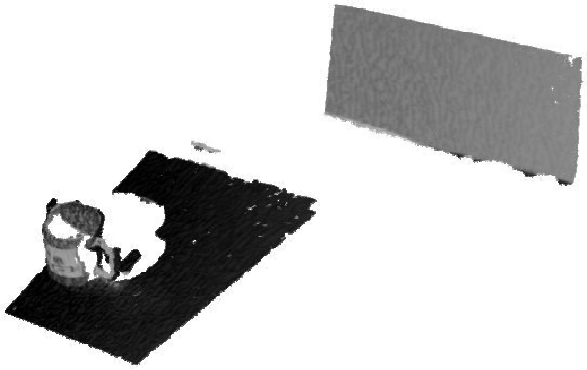
Filter example 1

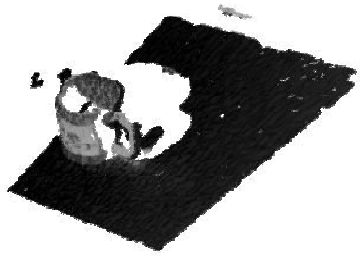
```
pcl::PassThrough<T> p;
p.setInputCloud (data);
p.FilterLimits (0.0, 0.5);
p.SetFilterFieldName ("z");
```





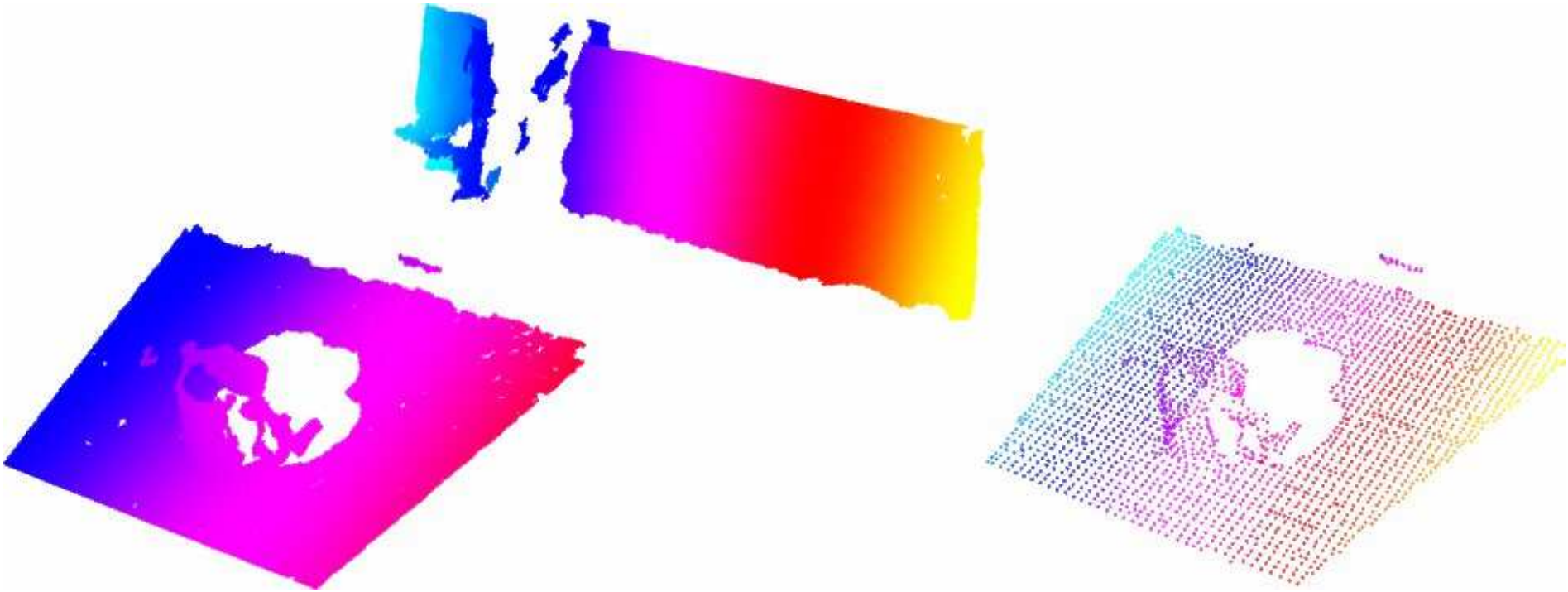
filter_field_name = "x"; | filter_field_name = "xz";





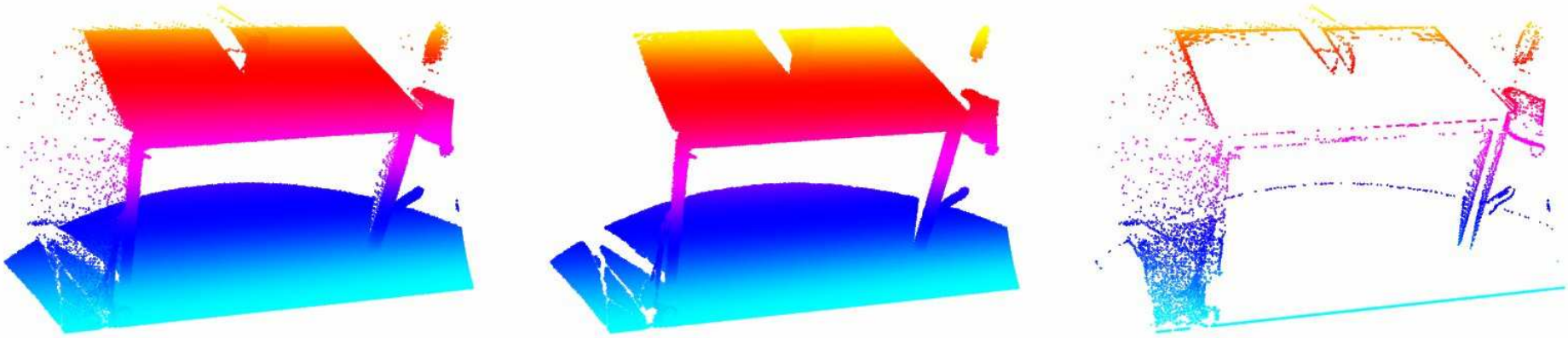
Filter example 2

```
pcl::VoxelGrid<T> p;
p.setInputCloud (data);
p.FilterLimits (0.0, 0.5);
p.SetFilterFieldName ("z");
p.setLeafSize (0.01, 0.01, 0.01);
```



Filter example 3

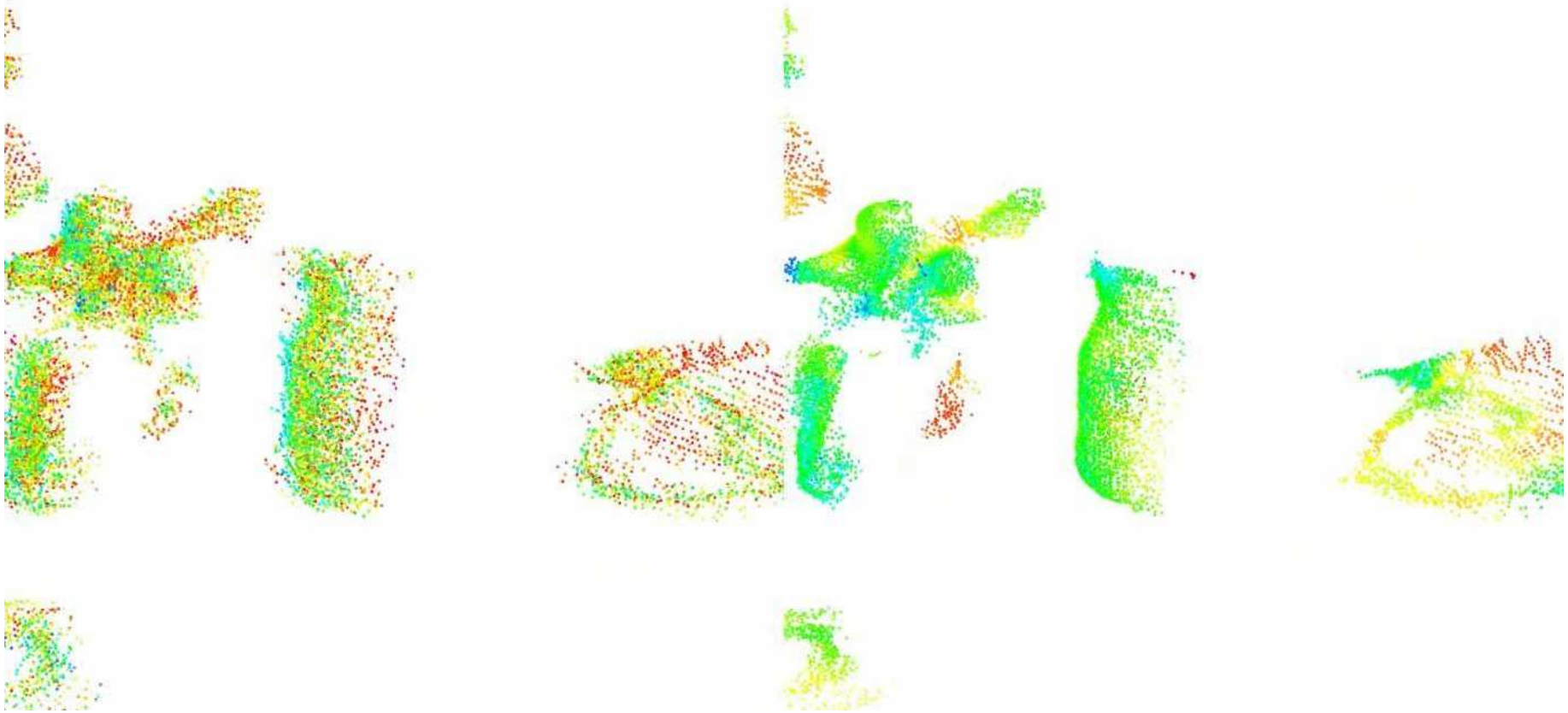
```
pcl::StatisticalOutlierRemoval<T> p;
p.setInputCloud (data);
p.setMeanK (50);
p.setStddevMulThresh (1.0);
```



Filter example 4

```
pcl::MovingLeastSquares<T> p;
p.setInputCloud (data);
p.setPolynomialOrder (3);
p.setSearchRadius (0.02);
```

(note: more of a surface reconstruction)



Features example 1

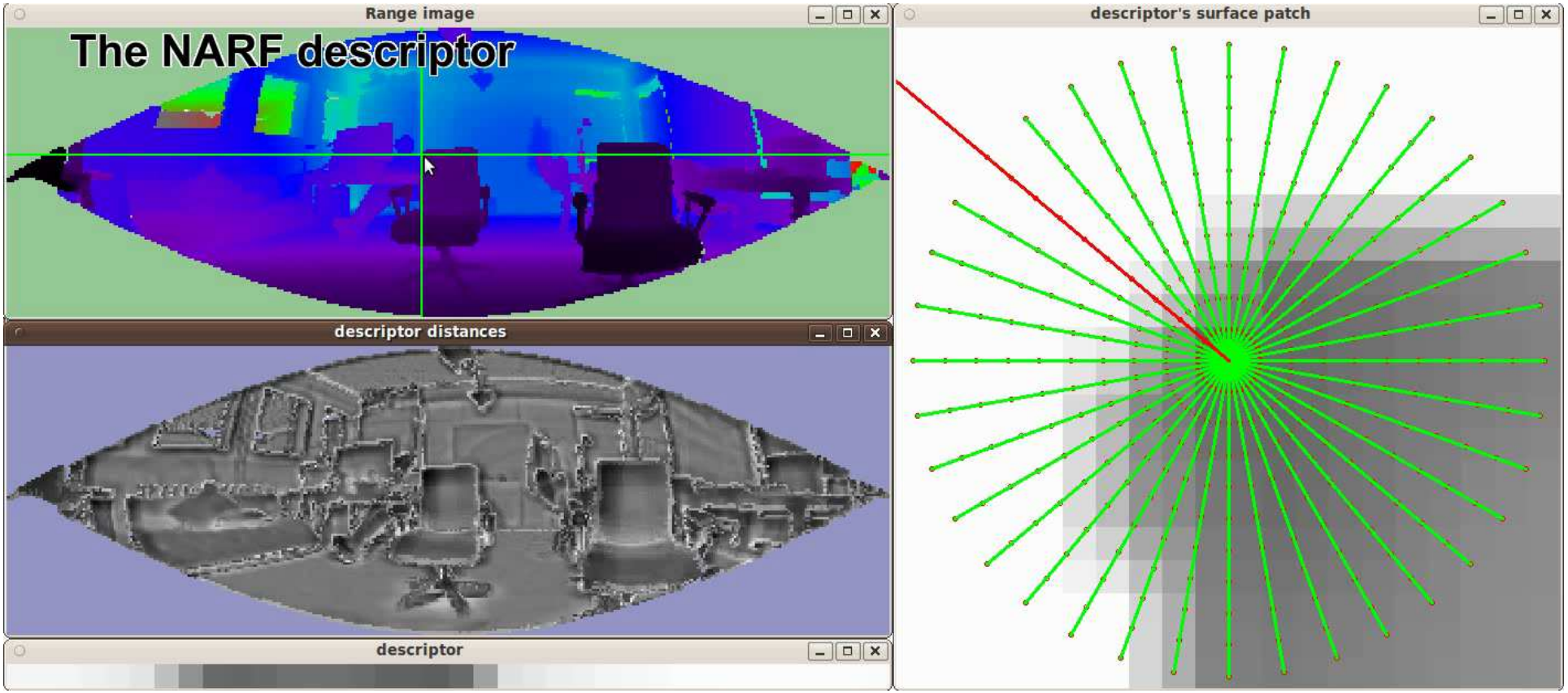
```
pcl::NormalEstimation<T> p;
p.setInputCloud (data);
p.SetRadiusSearch (0.01);
```

Features example 2

```
pcl::BoundaryEstimation<T,N> p;
p.setInputCloud (data);
p.setInputNormals (normals);
p.SetRadiusSearch (0.01);
```

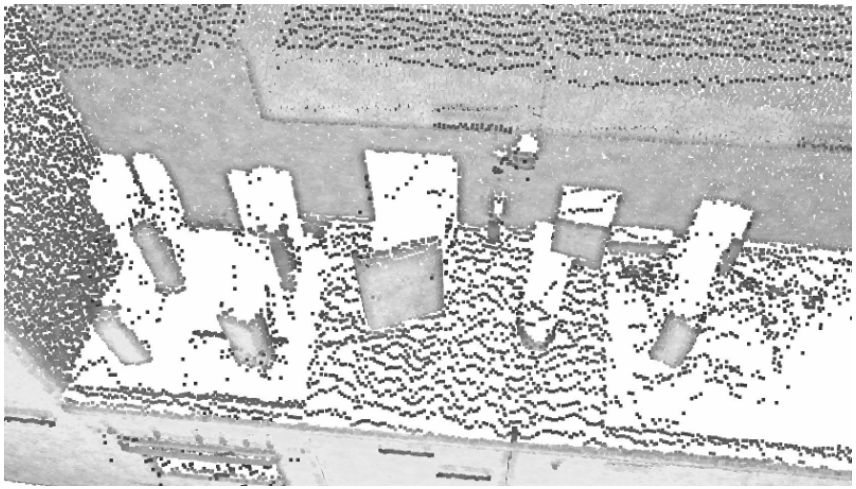
Features example 3

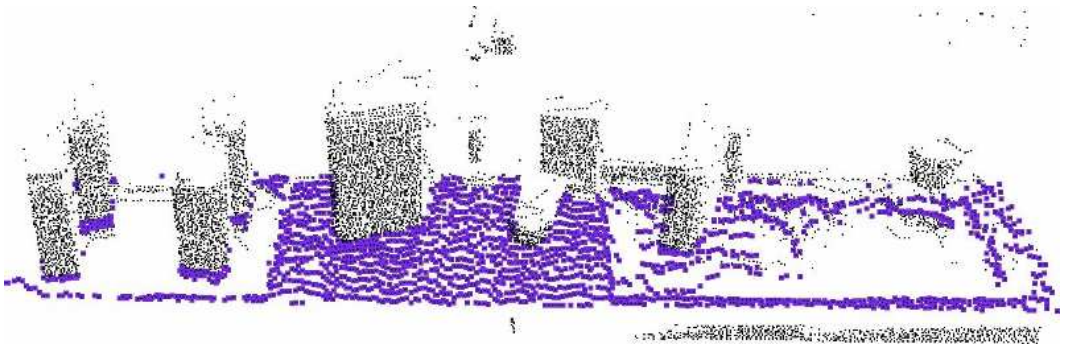
```
NarfDescriptor narf_descriptor(&range_image);
narf_descriptor.getParameters().support_size = 0.3;
narf_descriptor.getParameters().rotation_invariant = false;
PointCloud<Narf36> narf_descriptors;
narf_descriptor.compute(narf_descriptors);
```



Segmentation example 1

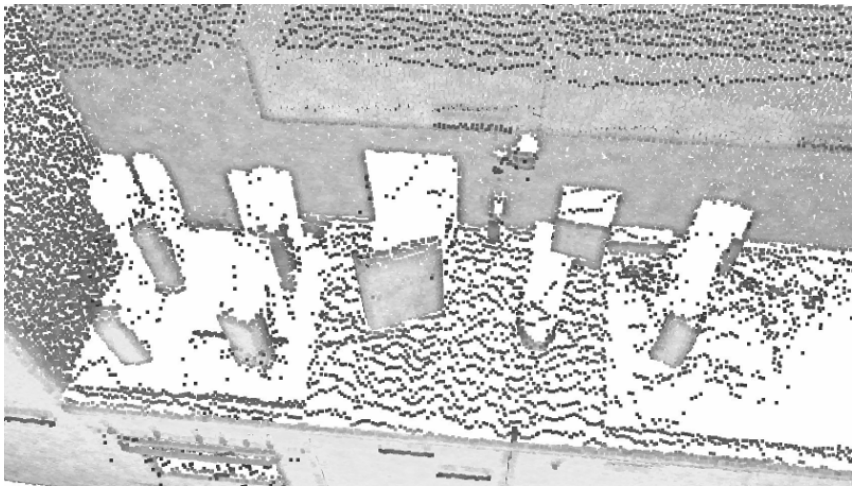
```
pcl::SACSegmentation<T> p;
p.setInputCloud (data);
p.setModelType (pcl::SACMODEL_PLANE);
p.setMethodType (pcl::SAC_RANSAC);
p.setDistanceThreshold (0.01);
```

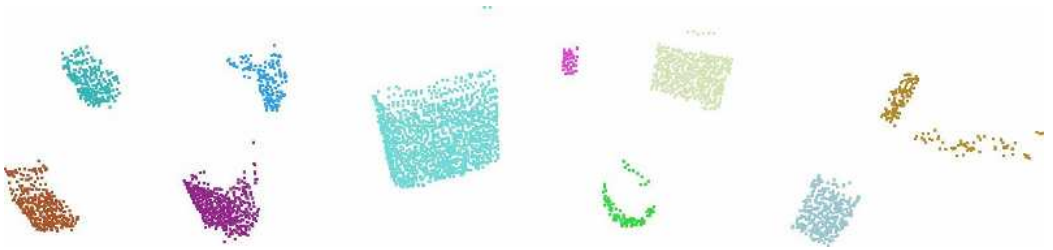




Segmentation example 2

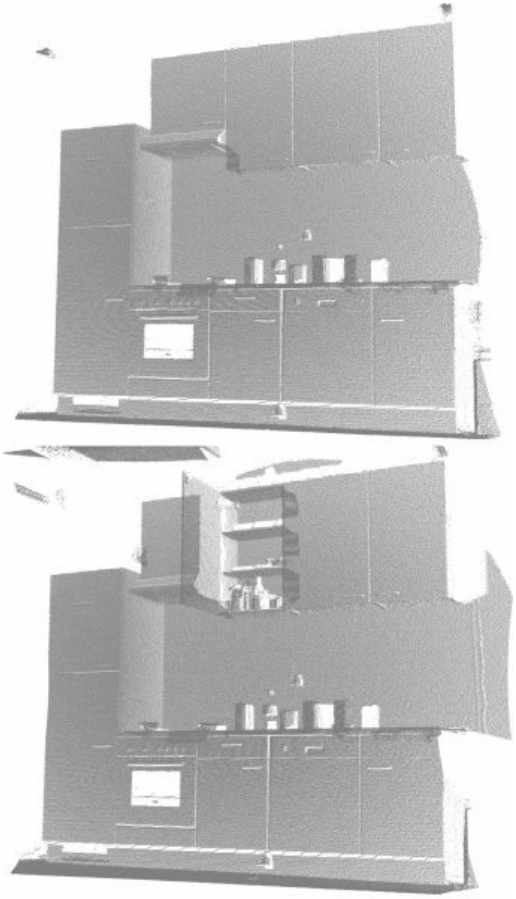
```
pcl::EuclideanClusterExtraction<T> p;
p.setInputCloud (data);
p.setClusterTolerance (0.05);
p.setMinClusterSize (1);
```

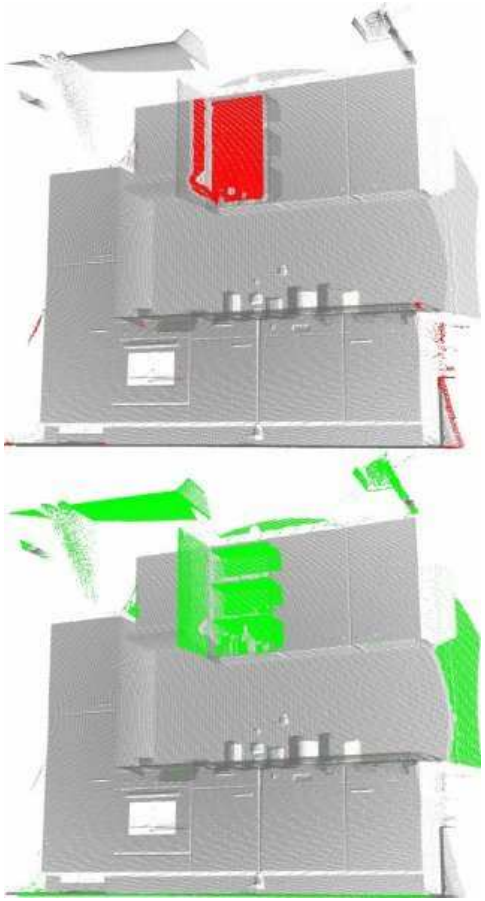




Segmentation example 3

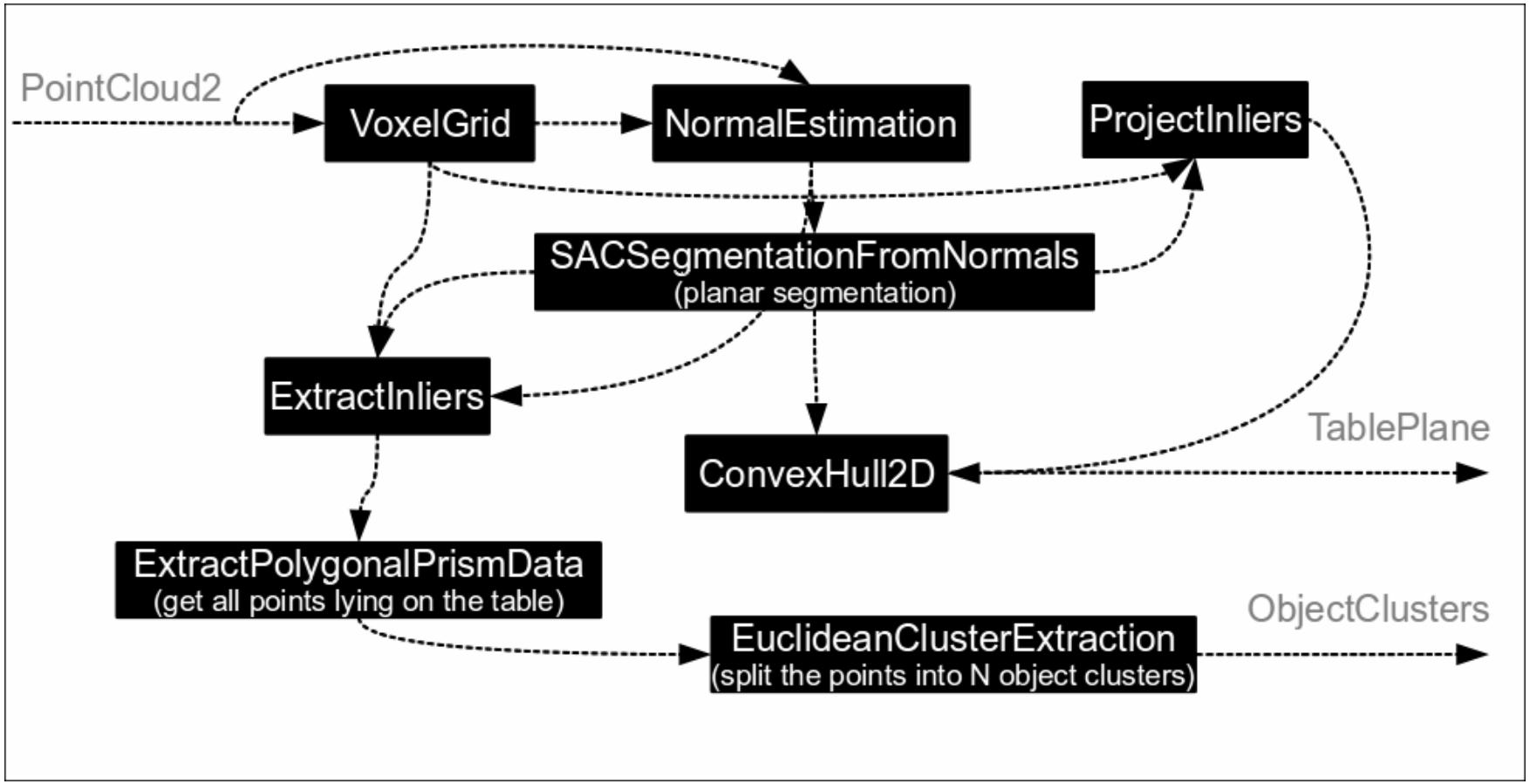
```
pcl::SegmentDifferences<T> p;
p.setInputCloud (source);
p.setTargetCloud (target);
p.setDistanceThreshold (0.001);
```





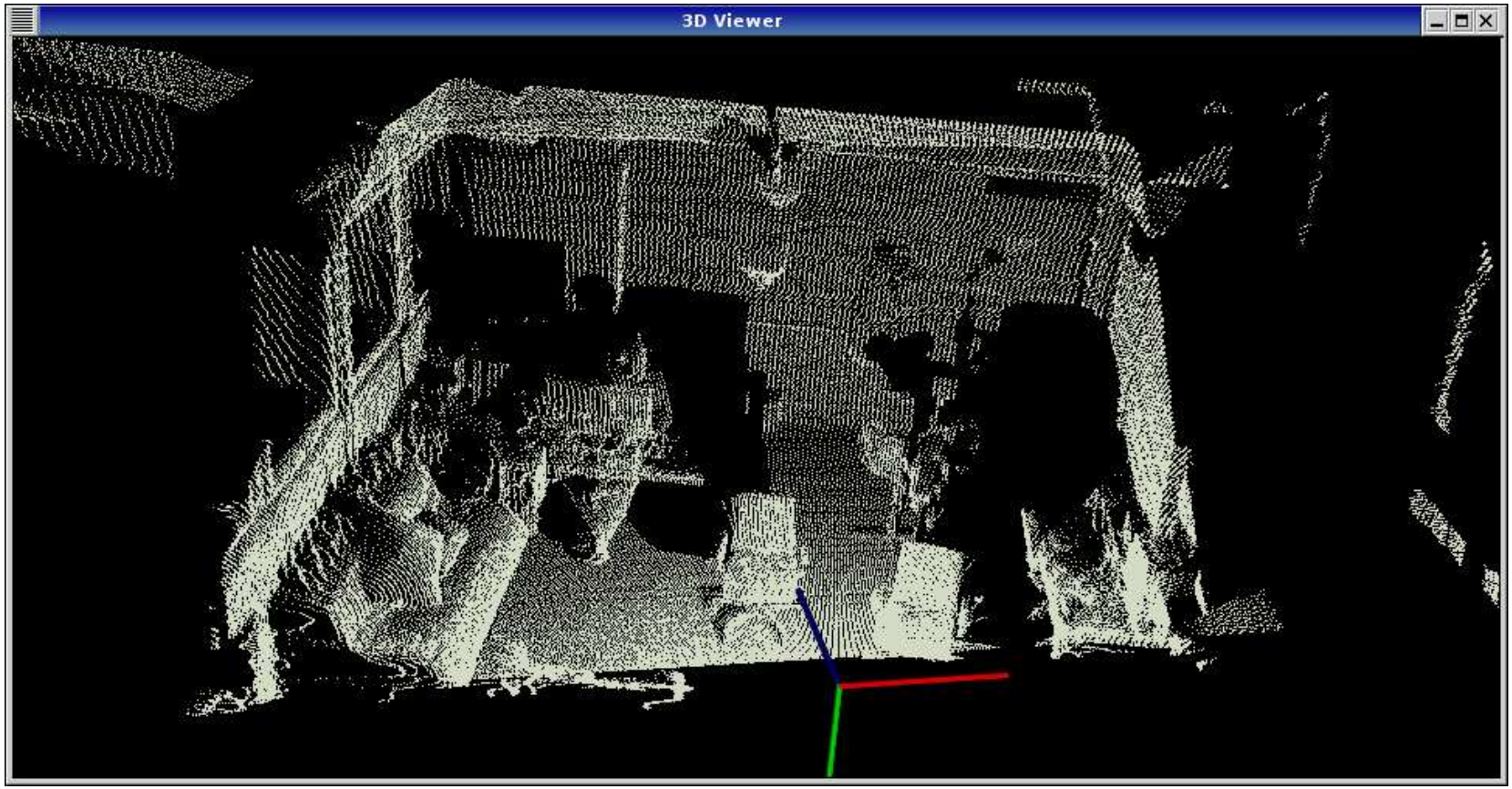
Higher level example

How to extract a table plane and the objects lying on it

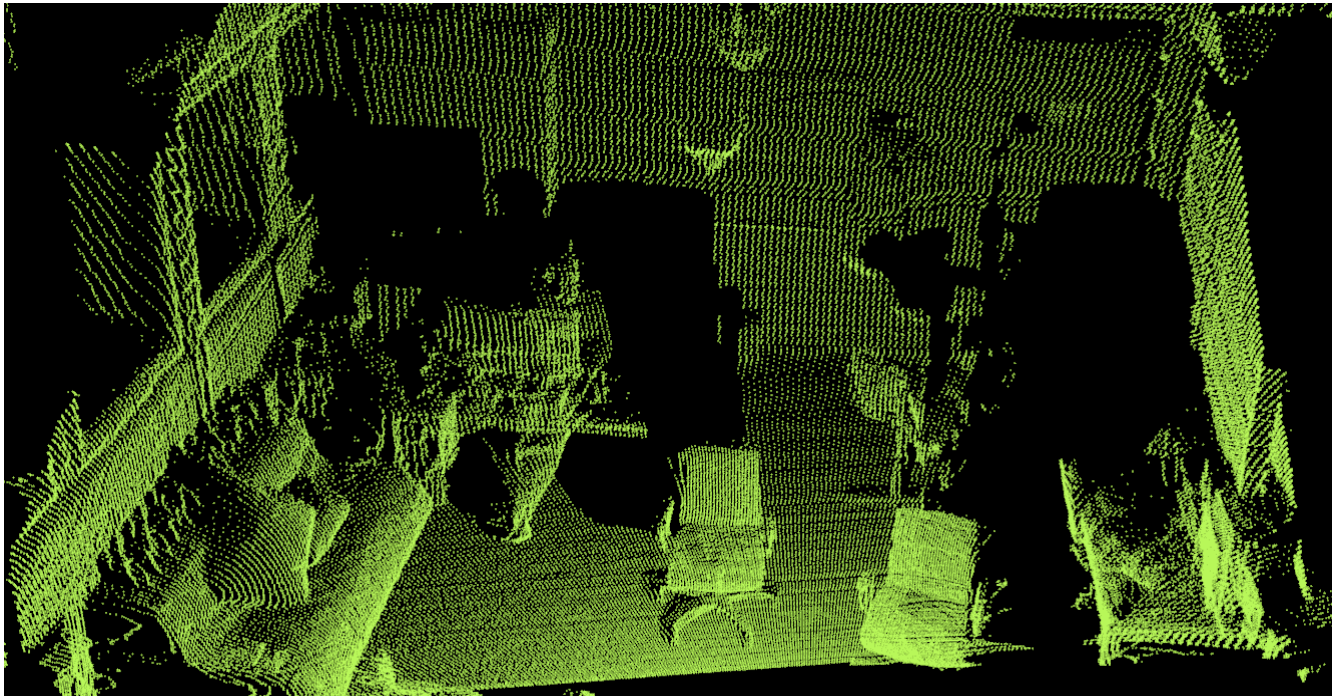


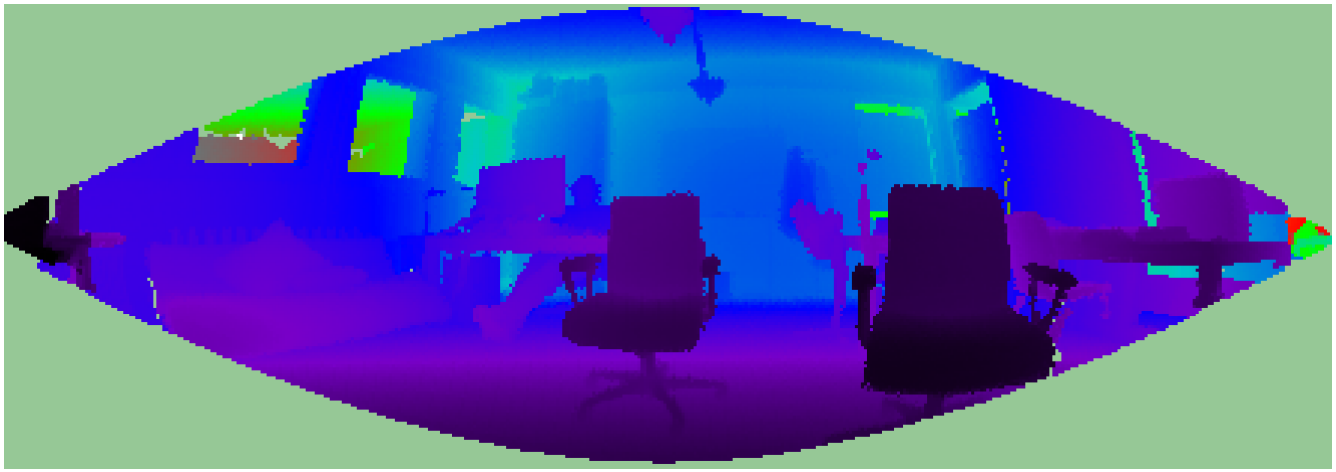
Visualization tools

```
PCLVisualizer viewer("3D Viewer");
viewer.addCoordinateSystem(1.0f);
viewer.addPointCloud(point_cloud, “our point cloud");
viewer.spin();
```



Range images





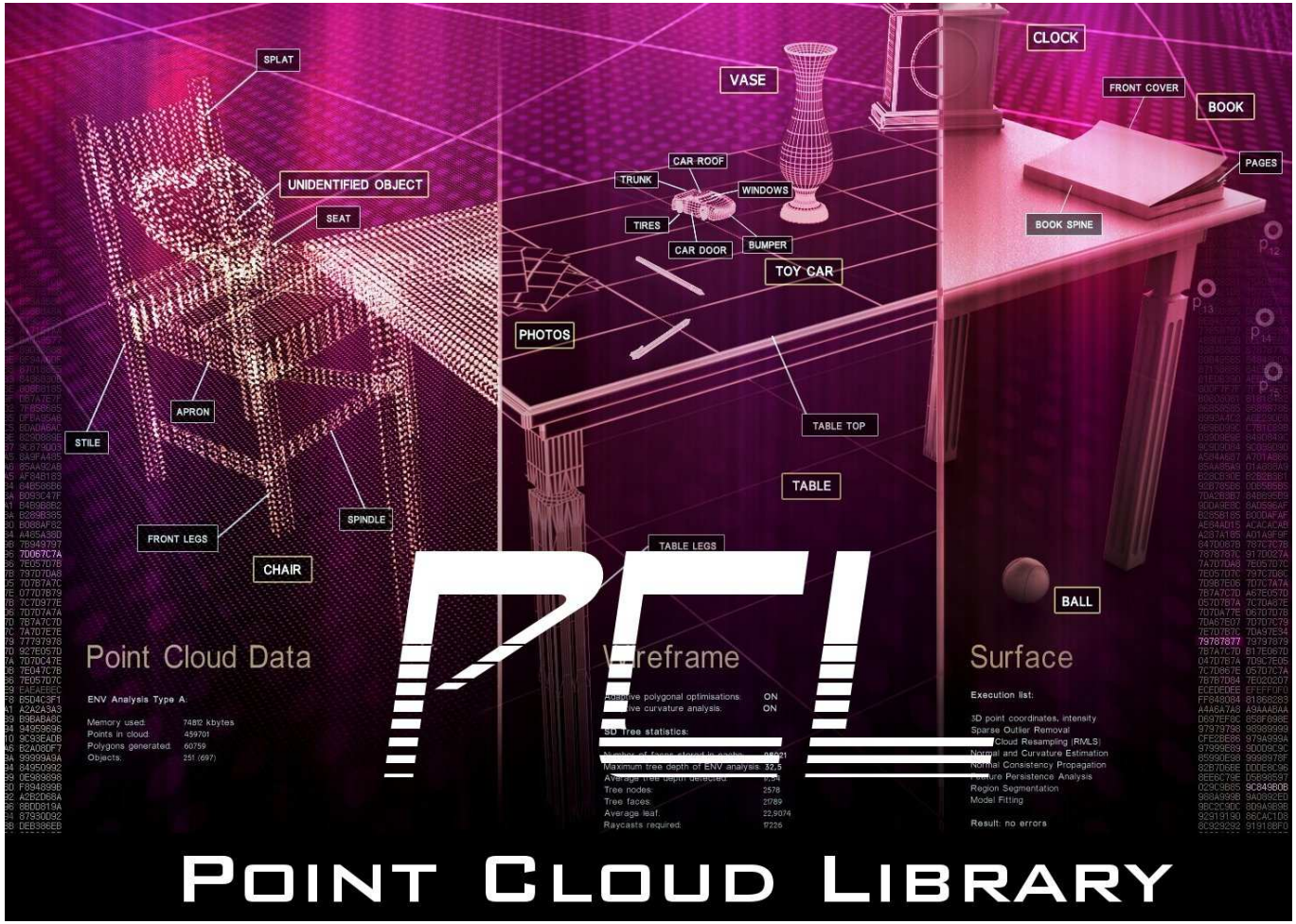
Range image class

A range image

- is derived from `PointCloud<PointWithRange>`
- can be created from a normal point cloud given a viewpoint
- is sometimes more convenient to work with:
  - neighborhood accessible
  - methods from vision usable
  - encodes negative information
- is easy to visualize
  (have a look at pcl_visualization -> range_image_visualizer)

More details

- See http://www.ros.org/wiki/pcl
  and http://www.ros.org/wiki/pcl/Tutorials



Passthrough filter example

```
#include "pcl/io/pcd_io.h"
#include "pcl/point_types.h"
#include "pcl/filters/passthrough.h"

int main (int argc, char** argv)
{
  pcl::PointCloud<pcl::PointXYZ> cloud, cloud_filtered;

  // Fill in the cloud data
  cloud.width  = 5;
  cloud.height = 1;
  cloud.points.resize (cloud.width * cloud.height);

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

  std::cerr << "Cloud before filtering: " << std::endl;
  for (size_t i = 0; i < cloud.points.size (); ++i)
    std::cerr << "    " << cloud.points[i].x << " " << cloud.points[i].y << " " << cloud.points[i].z << std::endl;

  pcl::PassThrough<pcl::PointXYZ> pass; // Create the filtering object
  pass.setInputCloud (boost::make_shared<pcl::PointCloud<pcl::PointXYZ> >(cloud));
  pass.setFilterFieldName ("z");
  pass.setFilterLimits (0.0, 1.0);
  pass.filter (cloud_filtered);

  std::cerr << "Cloud after filtering: " << std::endl;
  for (size_t i = 0; i < cloud_filtered.points.size (); ++i)
    std::cerr << "    " << cloud_filtered.points[i].x << " " << cloud_filtered.points[i].y << " " <<
     cloud_filtered.points[i].z << std::endl;

  return (0);
}
```

Passthrough filter example

- Create a new folder, e.g.: `mkdir $HOME/filter_test`
- Enter the folder: `cd $HOME/filter_test`
- Download the code for the current session:
  http://ros.informatik.uni-freiburg.de/wiki/doku.php?id=workshop
- Put it in the folder and unpack it:
  `tar xfz filter_test.tgz`
- `export ROS_PACKAGE_PATH=$HOME/filter_test:$ROS_PACKAGE_PATH`
- Edit `filter_test.cpp` and add e.g.:

```
pass.setFilterFieldName ("z");
pass.setFilterLimits (0.0, 1.0);
```

  To only keep elements with $0.0 \leq z \leq 1.0$
- `make` and run `bin/filter_test`

Uncomment the last part to have visualization