
# PointCloud(2) procesing in ROS.
# Point Cloud Library.

**Radu Bogdan RUSU**

May 2, 2010

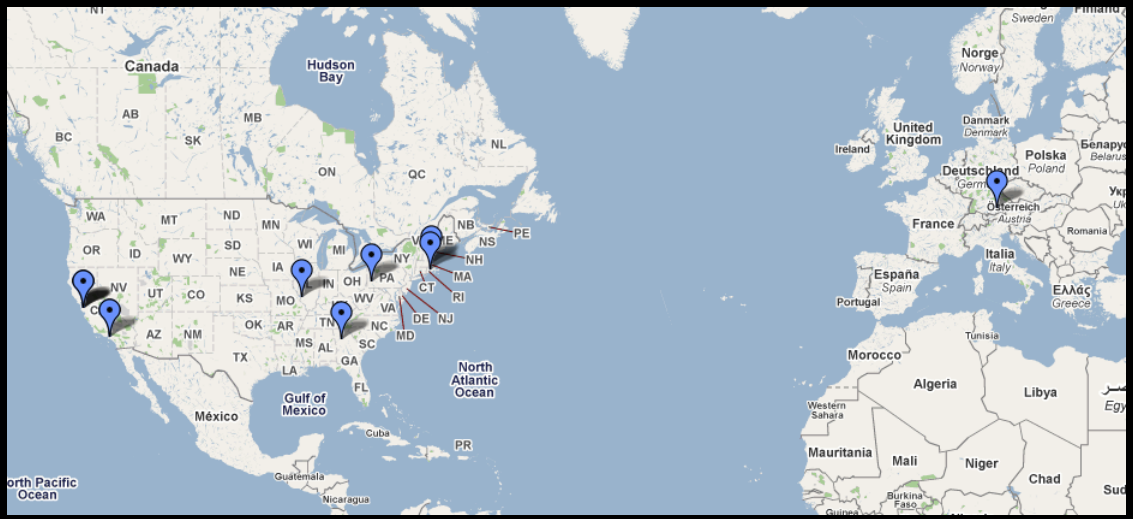
# Outline

# Introduction (1/6)

## What is ROS?

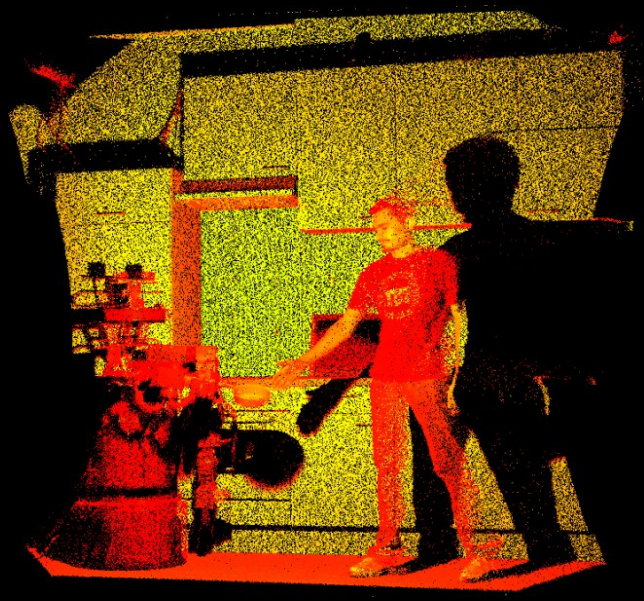
ROS is a comprehensive meta operating system for robotics

- organized via nodes, packages, stacks, repositories
- collection of building blocks
- community driven effort - all free/open source (!)
- stop reinventing the wheel, bring the best ouf of robotics
- replicating research results
- Friday tutorial (!!!)

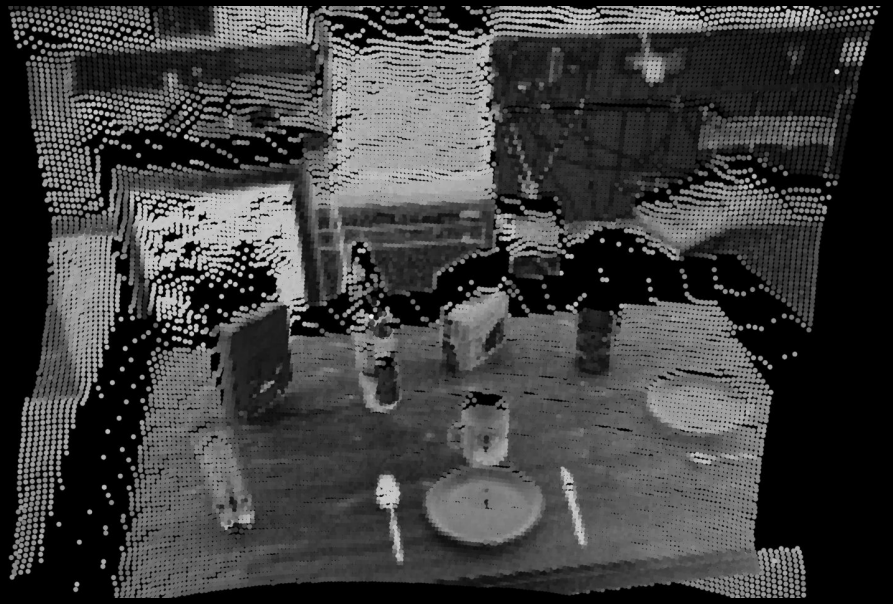


# Introduction (2/6)

What are Point Clouds?



*



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

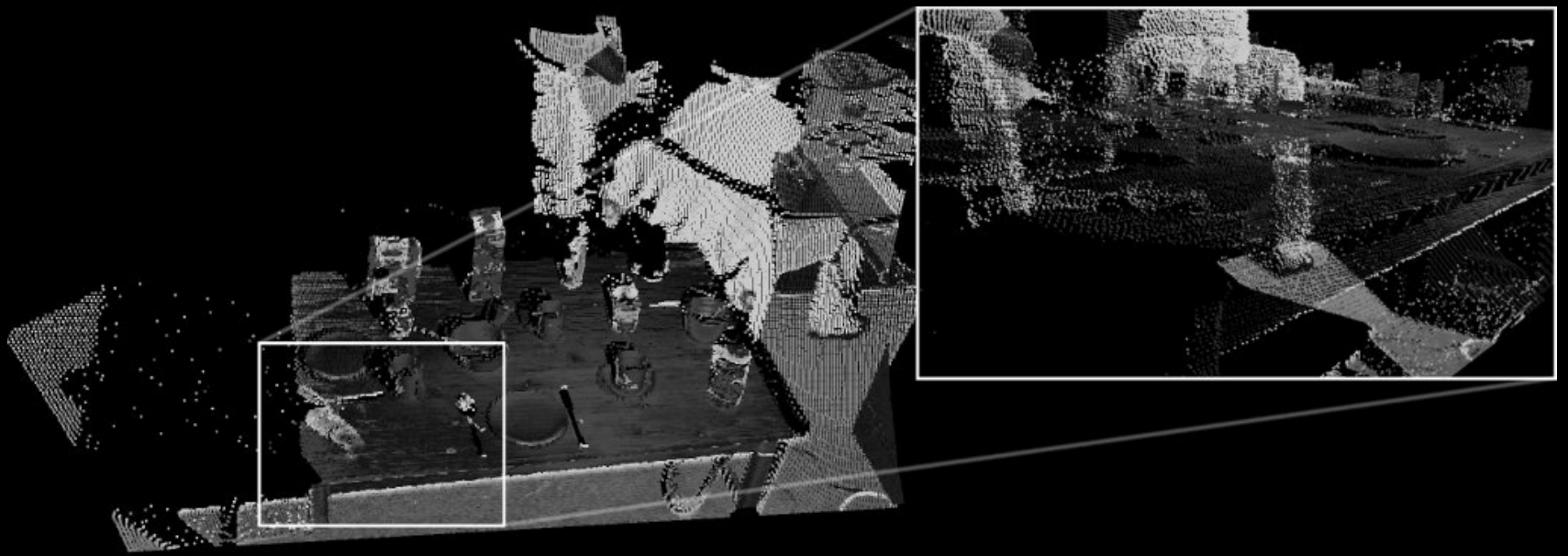
# Introduction (3/6)

What are Point Clouds?

- besides XYZ data, each point ***p*** can hold additional information
- examples include: RGB colors, intensity values, distances, segmentation results, etc

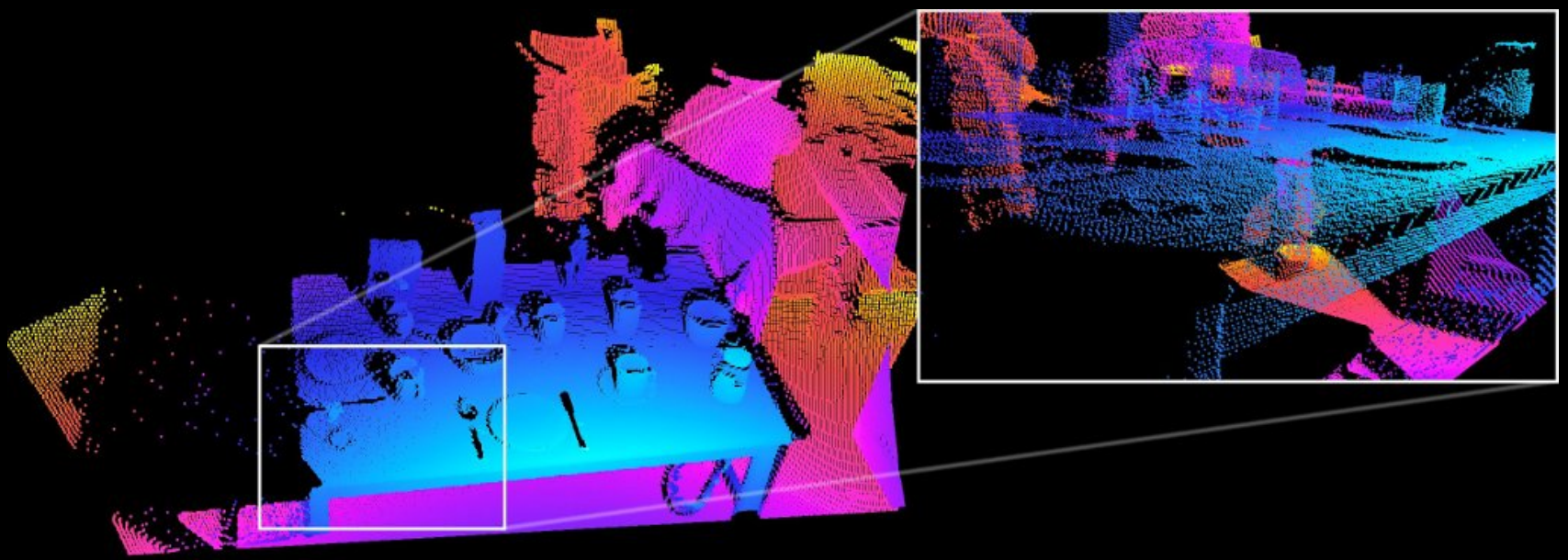
# Introduction (4-6/6)

What are Point Clouds?



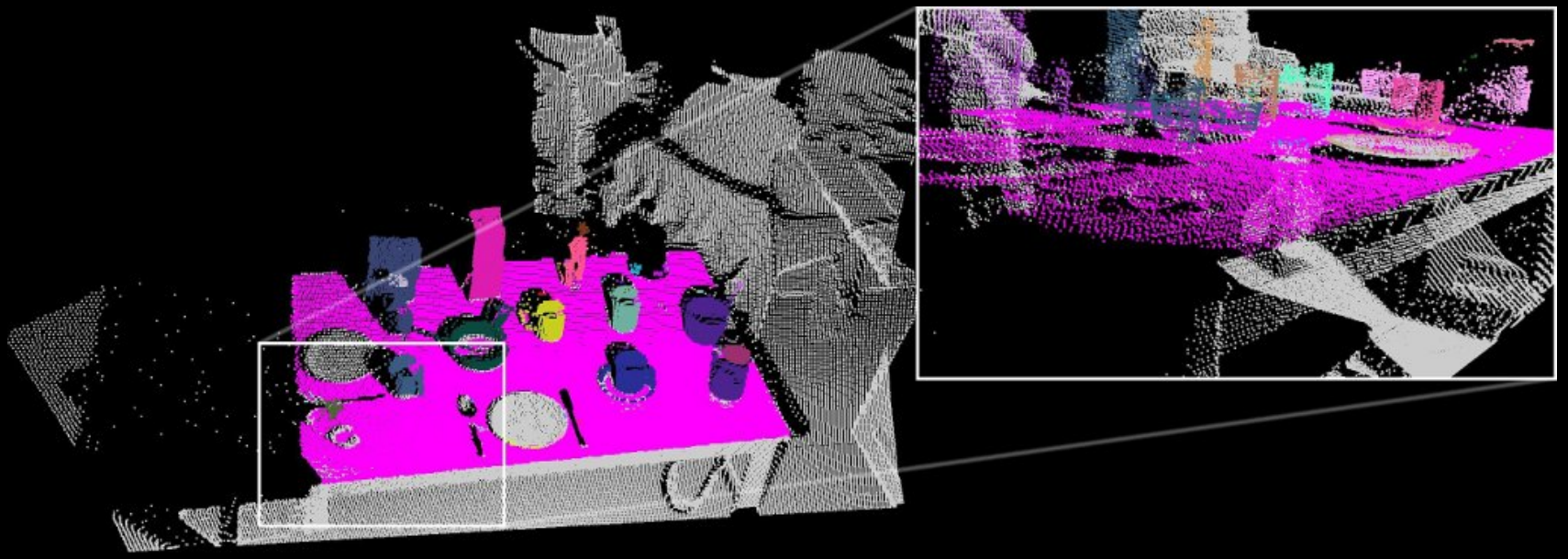
# Introduction (4-6/6)

What are Point Clouds?



# Introduction (4-6/6)

What are Point Clouds?

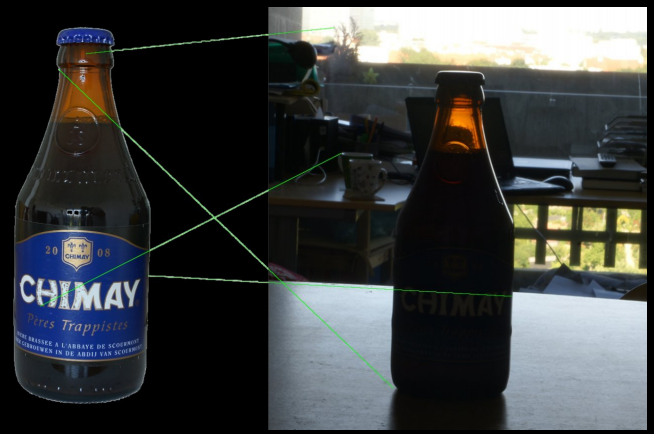


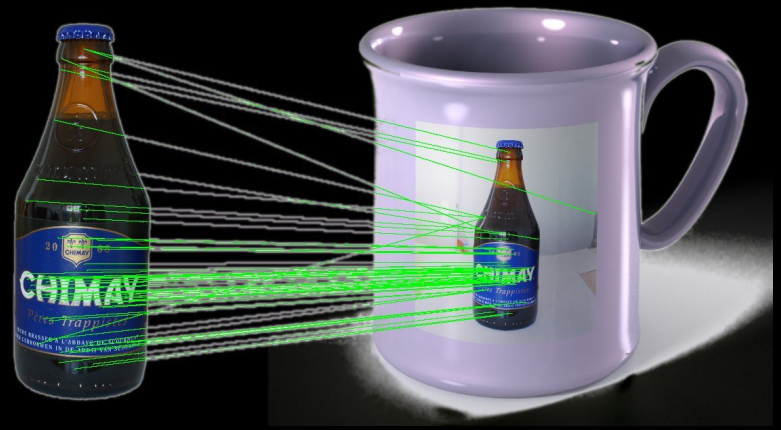
# Outline

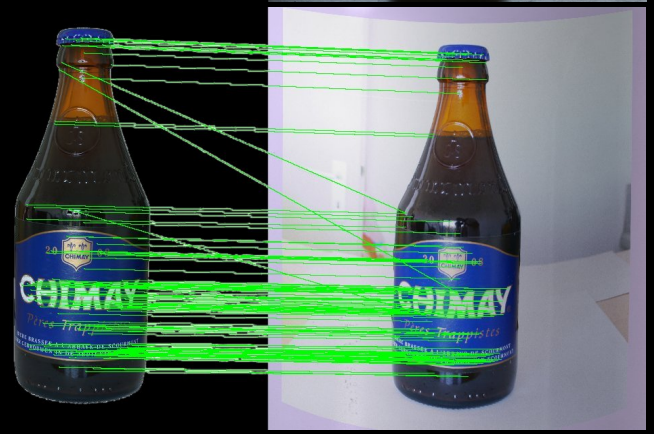
# Motivation (1/4)

Why are Point Clouds important?

Point Clouds are important for a lot of reasons (!). Besides representing geometry, they can complement and supersede images when data has a high dimensionality.



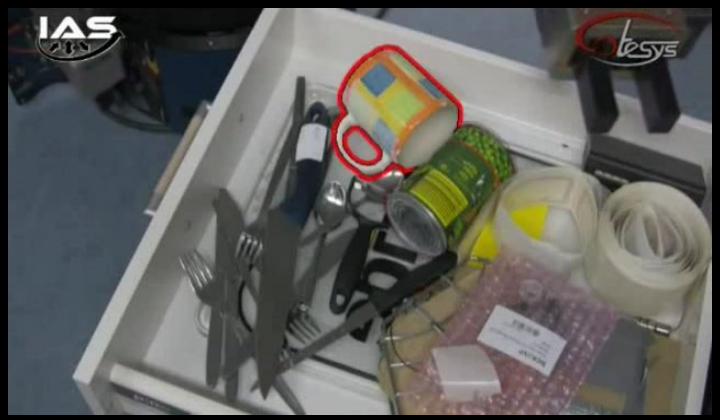



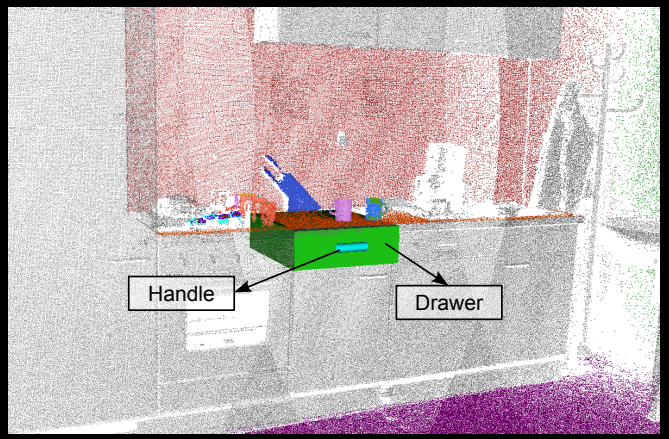




# Motivation (2/4)

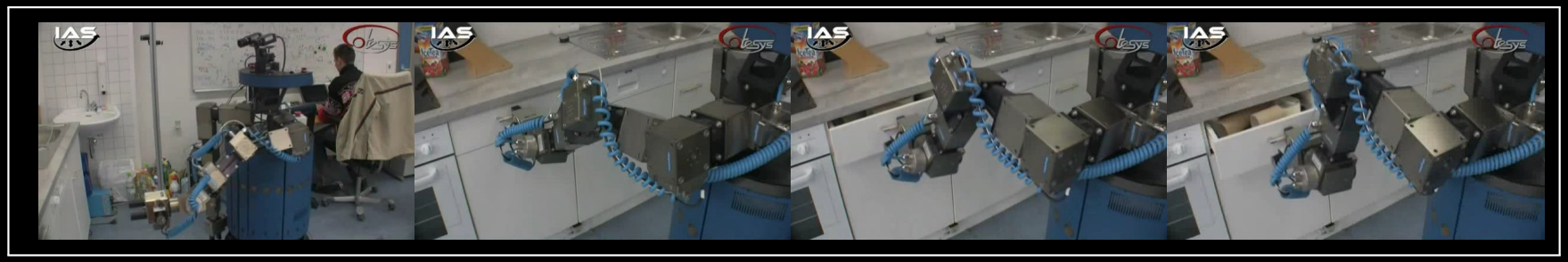
Why are Point Clouds important?

Concrete example 1: get the cup from the drawer.



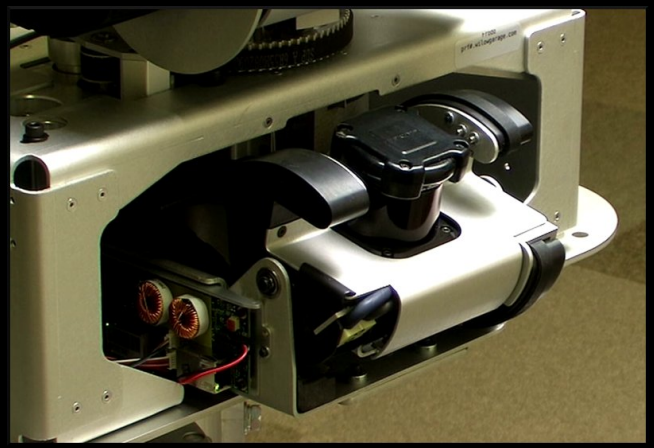

*

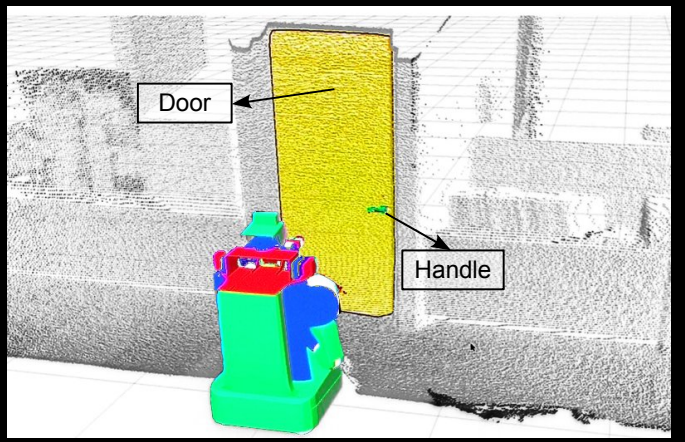


# Motivation (3/4)

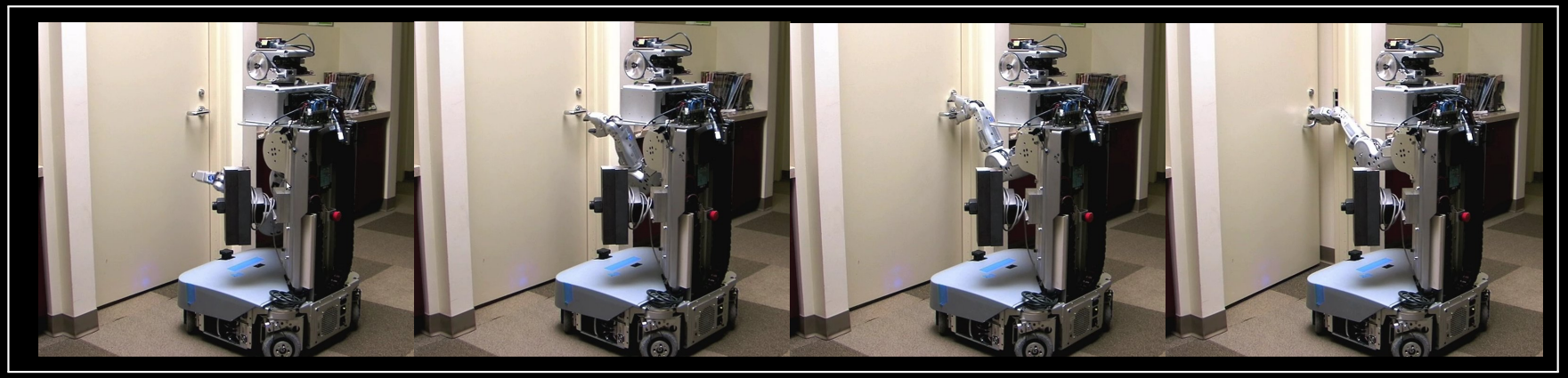
Why are Point Clouds important?

Concrete example 2: find the door and its handle, and open it.





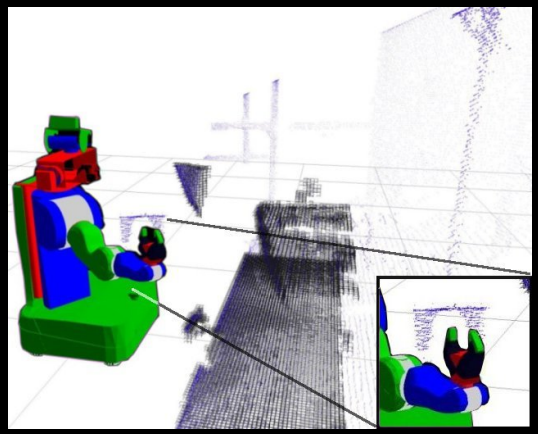

*

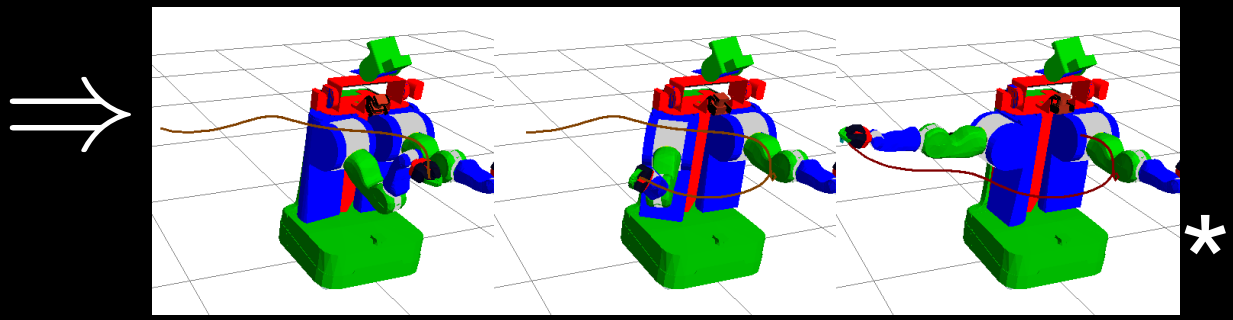


# Motivation (4/4)

Why are Point Clouds important?

Concrete example 3: safe motion planning/manipulation.





*

# Outline

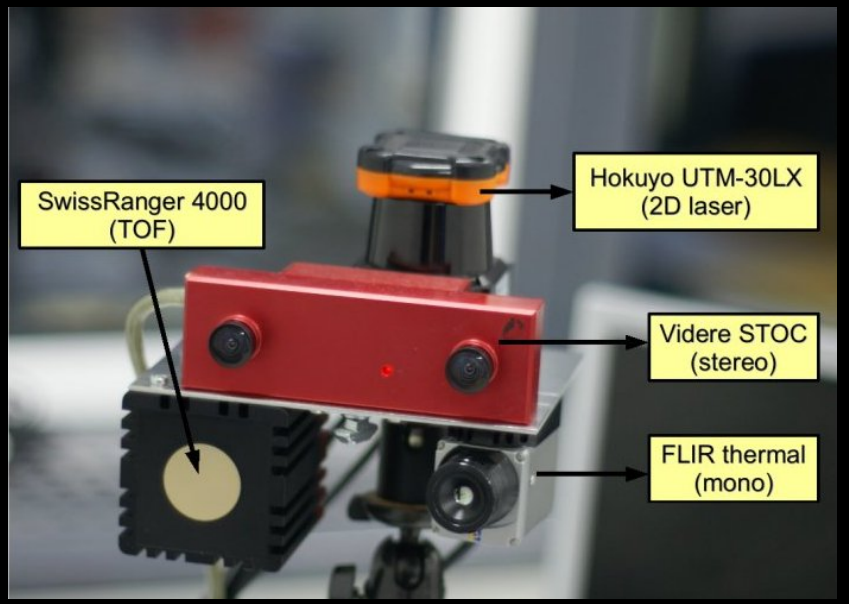
# Acquisition (1/3)

How are Point Clouds acquired? Where do they come from?

There are many different sensors that can generate 3D information. Examples:

- laser/lidar sensors (2D/3D)
- stereo cameras
- time-of-flight (TOF) cameras
- etc...

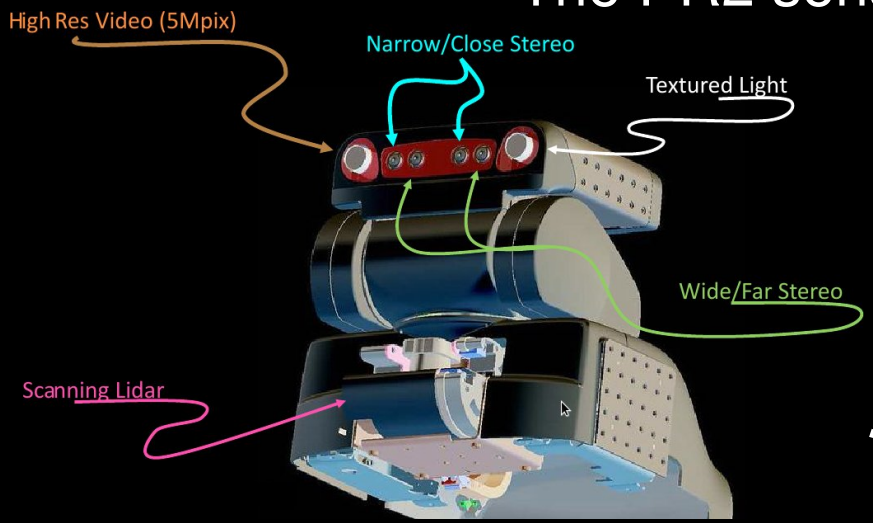

*



# Acquisition (2/3)

How are Point Clouds acquired? Where do they come from?

The PR2 sensor head:



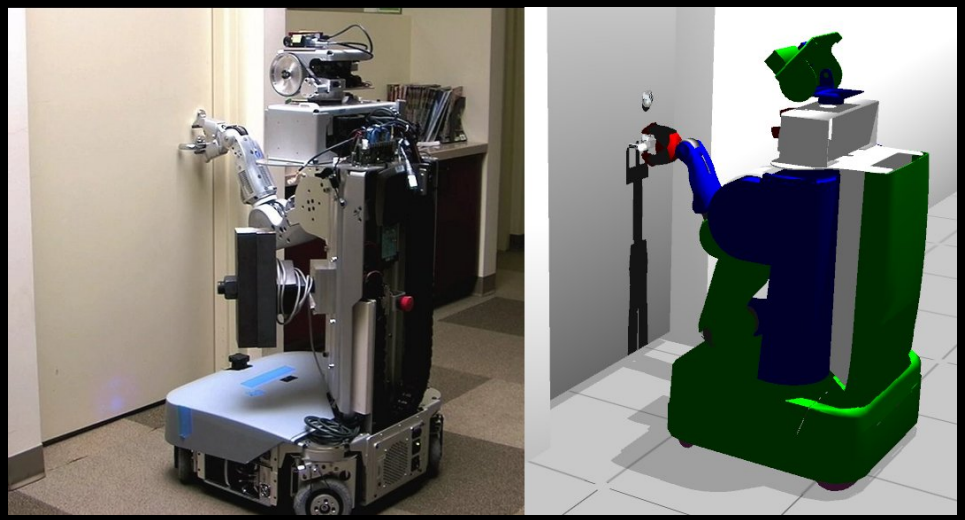

*

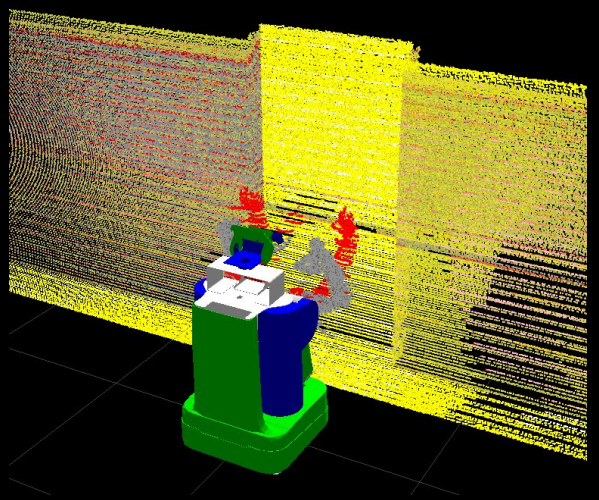
- two pairs of stereo cameras (narrow + wide)
- tilting laser sensor

# Acquisition (3/3)

How are Point Clouds acquired? Where do they come from?

Simulation (!):





- raytracing + stereo imagery fed into the same algorithmic modules that are used to process real data

# Outline

# Data representation (1/7)

## Representing Point Clouds

As previously presented:

- a point $\boldsymbol{p}$ is represented as an n-tuple, e.g., $\boldsymbol{p}_i = \{x_i, y_i, z_i, r_i, g_i, b_i, dist_i, \cdots\}$
- a Point Cloud $\mathcal{P}$ is represented as a collection of points $\boldsymbol{p}_i$, e.g., $\mathcal{P} = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \cdots, \boldsymbol{p}_i, \cdots, \boldsymbol{p}_n\}$

# Data representation (2/7)

## Point Cloud Data structures

In terms of data structures:

- an XYZ point can be represented as:
  ```
  float32 x
  float32 y
  float32 z
  ```
- a n-dimensional point can be represented as:
  ```
  float32[] point
  ```
  which is nothing else but a:
  ```
  std::vector<float32> point
  ```
  in C++
- potential problem: everything is represented as floats (!)

# Data representation (3/7)

Point Cloud Data structures

In terms of data structures:

- therefore 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***

# Data representation (4/7)

Point Cloud Data structures

Because Point Clouds are big:

- operations on them are typically slower (more data, more computations)
- they are expensive to store, especially if all data is represented as floats/doubles

Solutions:

# Data representation (4/7)

Point Cloud Data structures

Because Point Clouds are big:

- operations on them are typically slower (more data, more computations)
- they are expensive to store, especially if all data is represented as floats/doubles

Solutions:

- store each dimension data in different (the most appropriate) formats, e.g., rgb - 24bits, instead of $3 \times 4$ (sizeof float)
- group data together, and try to keep it aligned (e.g., 16bit for SSE) to speed up computations

# Data representation (5/7)

ROS representations for Point Cloud Data

## The ROS PointCloud(2) data format (sensor_msgs/PointCloud2.msg):

```
#This message holds a collection of nD points, as a binary blob.
Header header

#2D structure of the point cloud. If the cloud is unordered,
#height is 1 and width is the length of the point cloud.
uint32 height
uint32 width

#Describes the channels and their layout in the binary data blob
PointField[] fields

bool     is_bigendian #Is this data bigendian?
uint32   point_step   #Length of a point in bytes
uint32   row_step     #Length of a row in bytes
uint8[]  data         #Actual point data, size is (row_step*height)
bool is_dense         #True if there are no invalid points
```

# Data representation (6/7)

ROS representations for Point Cloud Data

## where PointField (sensor_msgs/PointField.msg) is:

```
#This message holds the description of one point entry in the #PointCloud2 message format.
uint8 INT8    = 1
uint8 UINT8   = 2
uint8 INT16   = 3
uint8 UINT16  = 4
uint8 INT32   = 5
uint8 UINT32  = 6
uint8 FLOAT32 = 7
uint8 FLOAT64 = 8
string name      # Name of field
uint32 offset    # Offset from start of point struct
uint8  datatype  # Datatype enumeration see above
uint32 count     # How many elements in field
```

## PointField examples:

```
"x",         0, 7,  1
"y",         4, 7,  1
"z",         8, 7,  1
"rgba",     12, 6,  1
"normal_x", 16, 8,  1
"normal_y", 20, 8,  1
"normal_z", 24, 8,  1
"fpfh",     32, 7, 33
```

# Data representation (7/7)

ROS representations for Point Cloud Data

- binary blobs are hard to work with
- we provide a custom converter, Publisher/Subscriber, transport tools, filters, etc, similar to images
- templated types: PointCloud2 $\longrightarrow$ PointCloud<**PointT**>
- examples of **PointT**:

```
struct PointXYZ
{
  float x;
  float y;
  float z;
}
struct Normal
{
  float normal[3];
  float curvature;
}
```

# Outline

# Point Cloud Data storage (1/2)

ROS input/output

- PointCloud2.msg and PointField.msg are ROS messages
- they can be published on the network, saved/loaded to/from BAG files (ROS message logs)
- usage example:

```
$ rostopic find sensor_msgs/PointCloud2 | xargs rosrecord -F foo
[ INFO] [1271297447.656414502]: Recording to foo.bag.
^C
[ INFO] [1271297450.723504983]: Closing foo.bag.
$ rosplay -c foo.bag
bag: foo.bag
version: 1.2
start_time: 1271297447974280542
end_time: 1271297449983577462
length: 2009296920
topics:
  - name: /narrow_stereo_textured/points2
    count: 3
    datatype: sensor_msgs/PointCloud2
    md5sum: 1158d486dd51d683ce2f1be655c3c181
```

# Point Cloud Data storage (2/2)

PCD (Point Cloud Data) file format

In addition, point clouds can be stored to disk as files, into the PCD format.

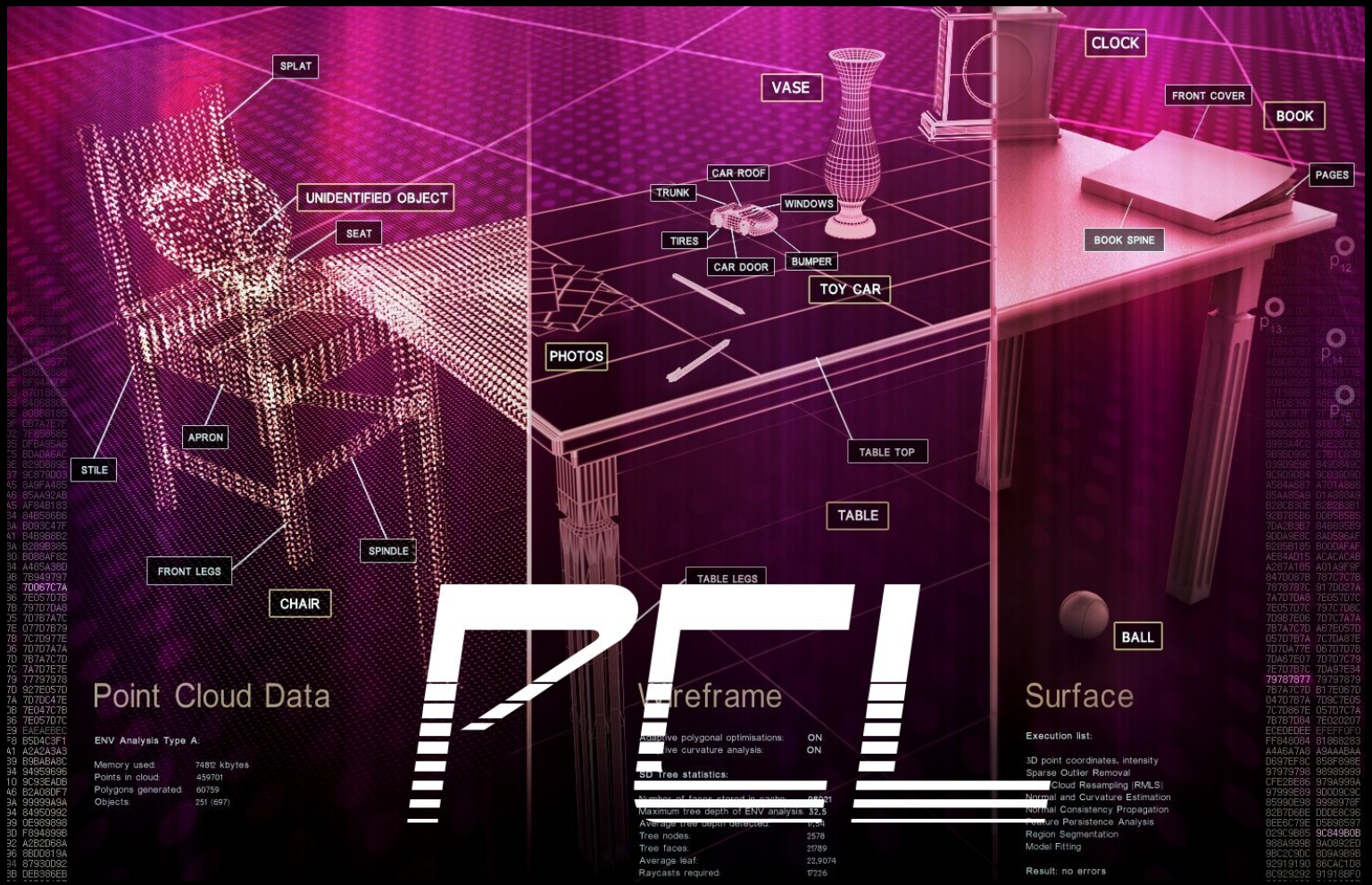
```
# Point Cloud Data (PCD) file format v.5
FIELDS x y z rgba
SIZE 4 4 4 4
TYPE F F F U
WIDTH 307200
HEIGHT 1
POINTS 307200
DATA binary
...
```

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
...
```

# Outline

# Point Cloud Library (1/11)




# Point Cloud Library

http://www.ros.org/wiki/pcl

# Point Cloud Library (2/11)

## What is PCL (Point Cloud Library)?

PCL is:

- 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
- OpenCV's 3D little sister

PCL deprecates older ROS packages such as point_cloud_mapping and replaces sensor_msgs/PointCloud.msg with the modern sensor_msgs/PointCloud2.msg format (!)

# Point Cloud Library (3/11)

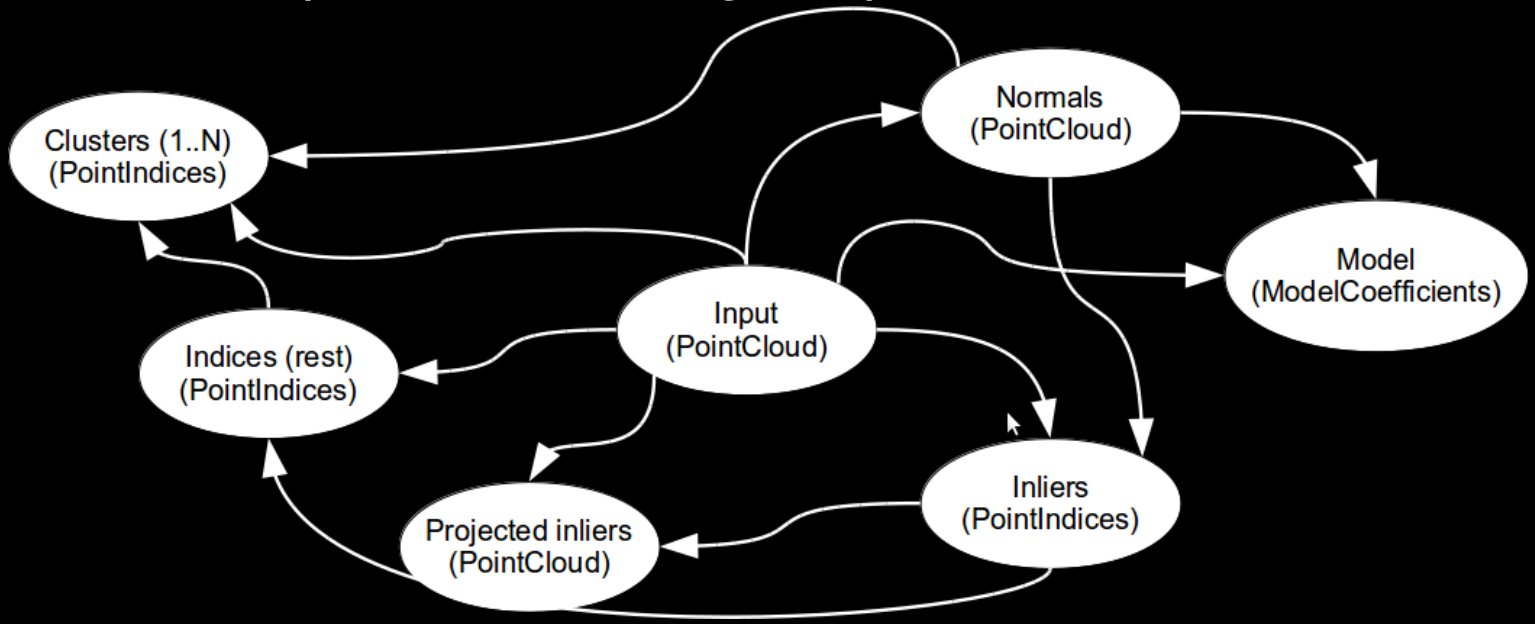
PCL (Point Cloud Library) structure

- 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, ...)
- unit tests, examples, tutorials (some are work in progress)
- C++ classes are templated building blocks (nodelets!)

# Point Cloud Library (4/11)

PPG: Perception Processing Graphs

- Philosophy: *write once, parameterize everywhere*
- PPG: Perception Processing Graphs

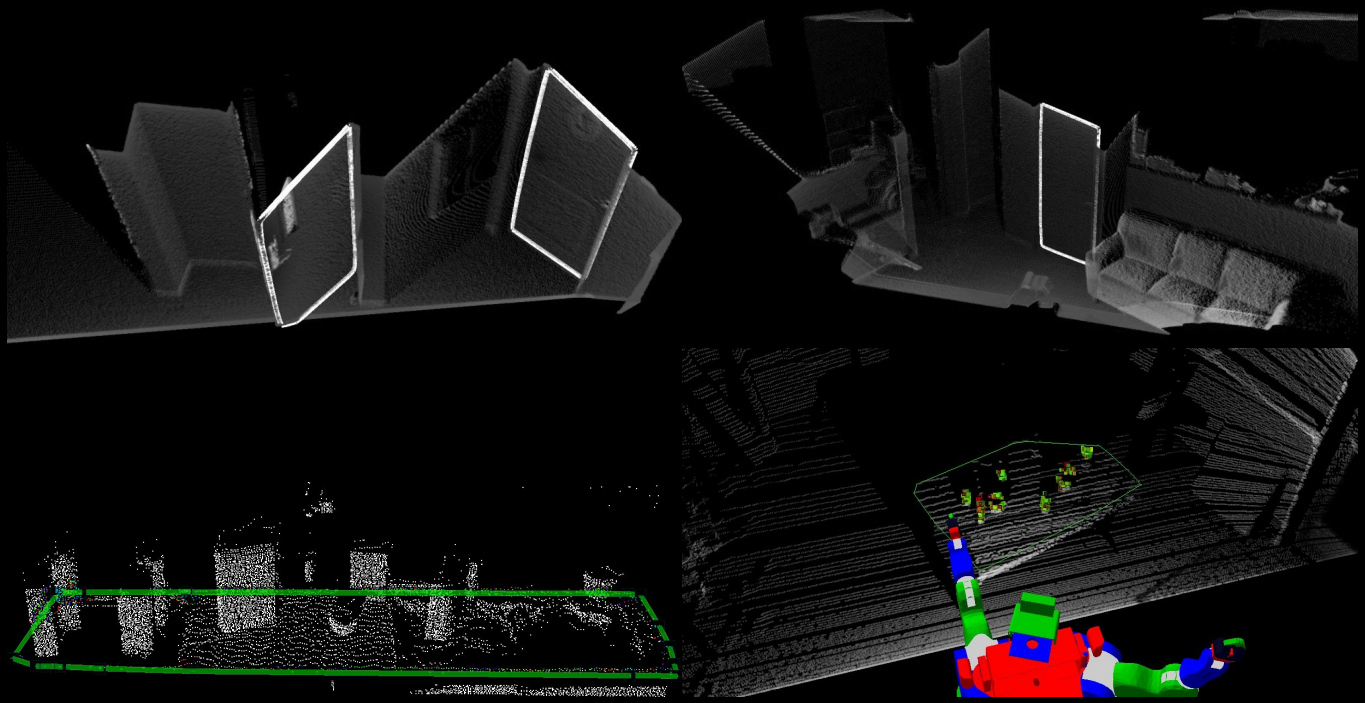


# Point Cloud Library (5/11)

PPG: Perception Processing Graphs

Why PPG?

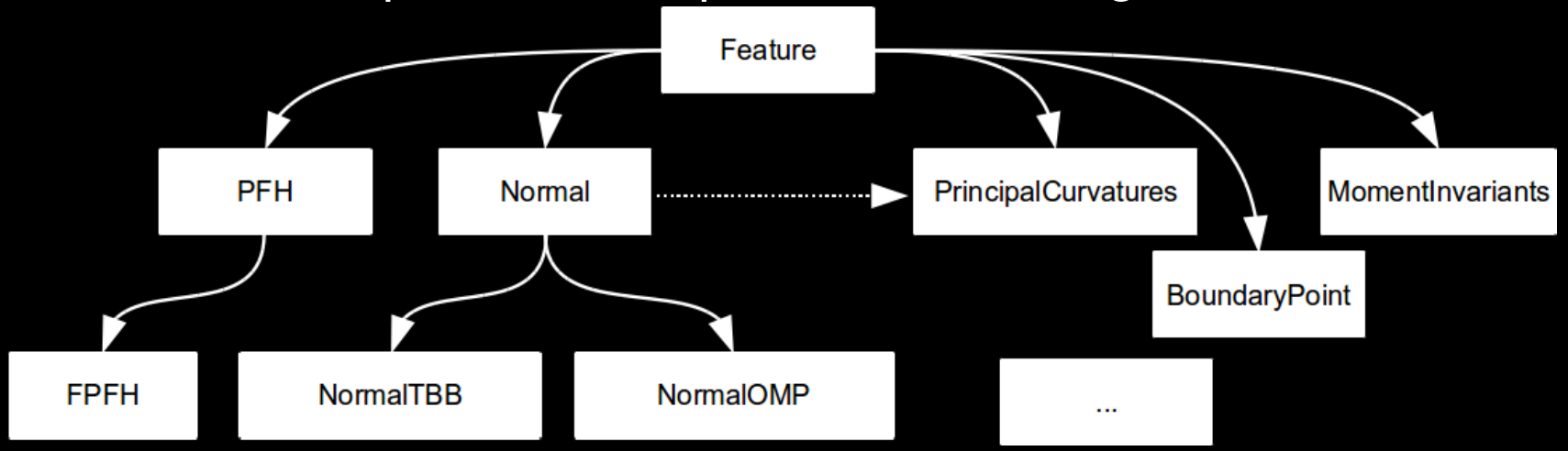
- Algorithmically:
  door detection = table detection = wall detection = ...
- the only thing that changes is: parameters (constraints)!



# Point Cloud Library (6/11)

## More on 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);
...
```

# Point Cloud Library (7/11)

PCL 0.1.x statistics

Misc, stats:

- over 100 classes
- over 25k lines of code
- external dependencies (for now) on eigen, cminpack, ANN, FLANN, TBB
- internal dependencies (excluding the obvious) on dynamic_reconfigure, message_filters
- tf_pcl package for TF integration

# Point Cloud Library (8/11)

## Nodelets

- *write once, parameterize everywhere* $\Longrightarrow$ modular code
- ideally, each algorithm is a “building block” that consumes input(s) and produces some output(s)
- in ROS, this is what we call a node. inter-process data passing however is inefficient. ideally we need shared memory.

**Solution:**

nodelets = “nodes in nodes” = single-process, multi-threading

# Point Cloud Library (8/11)

## Nodelets

- *write once, parameterize everywhere* $\Longrightarrow$ modular code
- ideally, each algorithm is a “building block” that consumes input(s) and produces some output(s)
- in ROS, this is what we call a node. inter-process data passing however is inefficient. ideally we need shared memory.

**Solution:**

nodelets = “nodes in nodes” = single-process, multi-threading

- same ROS API as nodes (subscribe, advertise, publish)
- dynamically (un)loadable
- optimizations for zero-copy Boost shared_ptr passing
- PCL nodelets use dynamic_reconfigure for on-the-fly parameter setting

# Point Cloud Library (9/11)

## Nodelets

Performance hints:

- 29 msg/s (inter process) vs 4664 mgs/s (intra process)
- fast (un)loading for algorithms (one service request)
- natively multithreaded (boost::thread::hardware_concurrency ())
- threading scheduler and granularity controllable in OpenMP/TBB

Goals:

- GUI for managing nodelets => 3D “Simulink” (!)
- get to a point where fast prototyping is easy

# Point Cloud Library (10/11)

## Downsample and filtering example with nodelets

```
<launch>
  <node pkg="nodelet" type="standalone_nodelet"
        name="pcl_manager" output="screen" />

  <node pkg="nodelet" type="nodelet" name="foo"
        args="voxel_grid␣VoxelGrid␣pcl_manager">
    <remap from"/voxel_grid/input"
           to="/narrow_stereo_textured/points" />
    <rosparam>
     # -[ Mandatory parameters
     leaf_size: [0.015, 0.015, 0.015]
     # -[ Optional parameters
     # field containing distance values (for filtering)
     filter_field_name: "z"
     # filtering points outside of <0.8,5.0>
     filter_limit_min: 0.8
     filter_limit_max: 5.0
     use_indices: false        # false by default
    </rosparam>
  </node>
...
</launch>
```

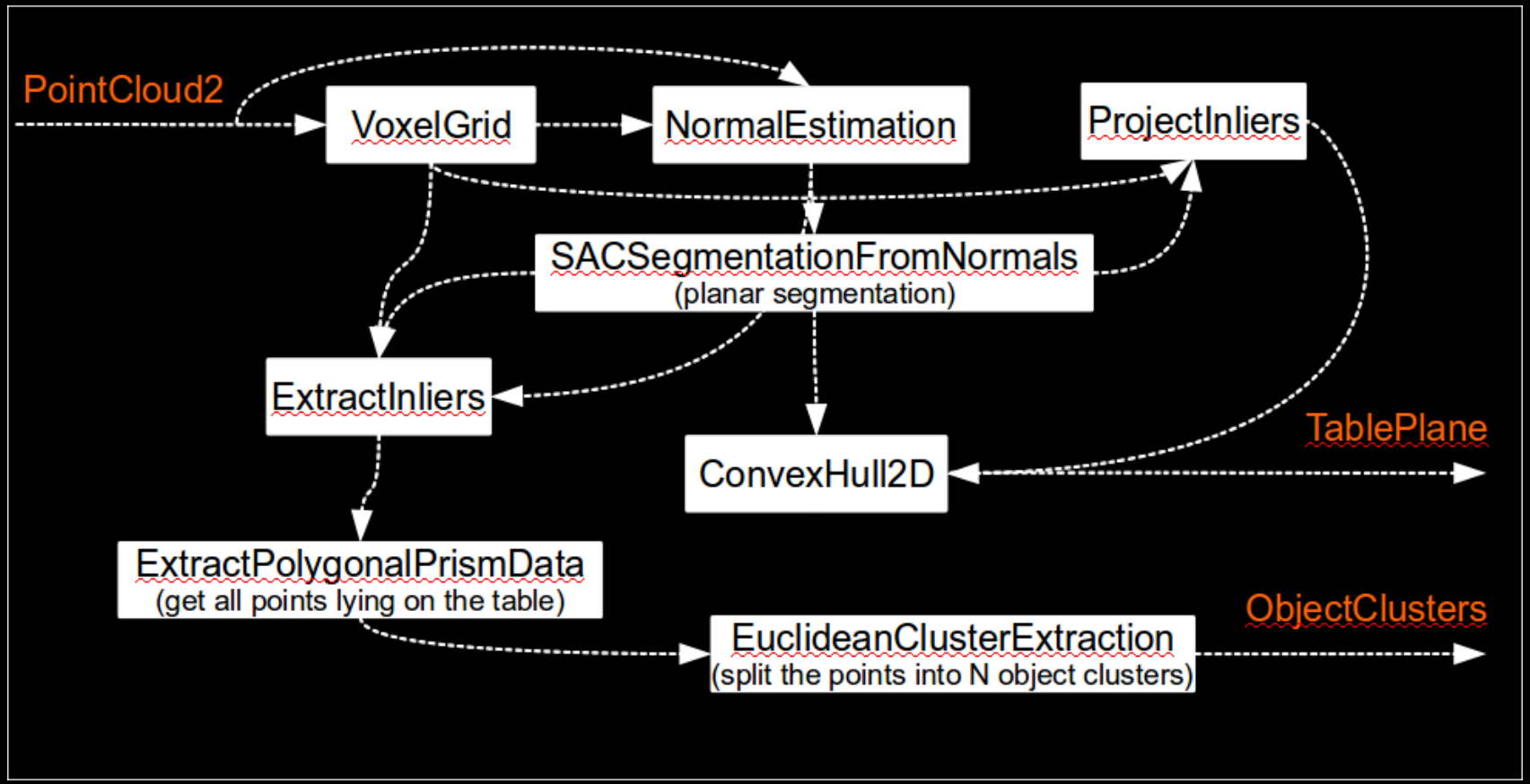
# Point Cloud Library (11/11)

## Normal estimation example with nodelets

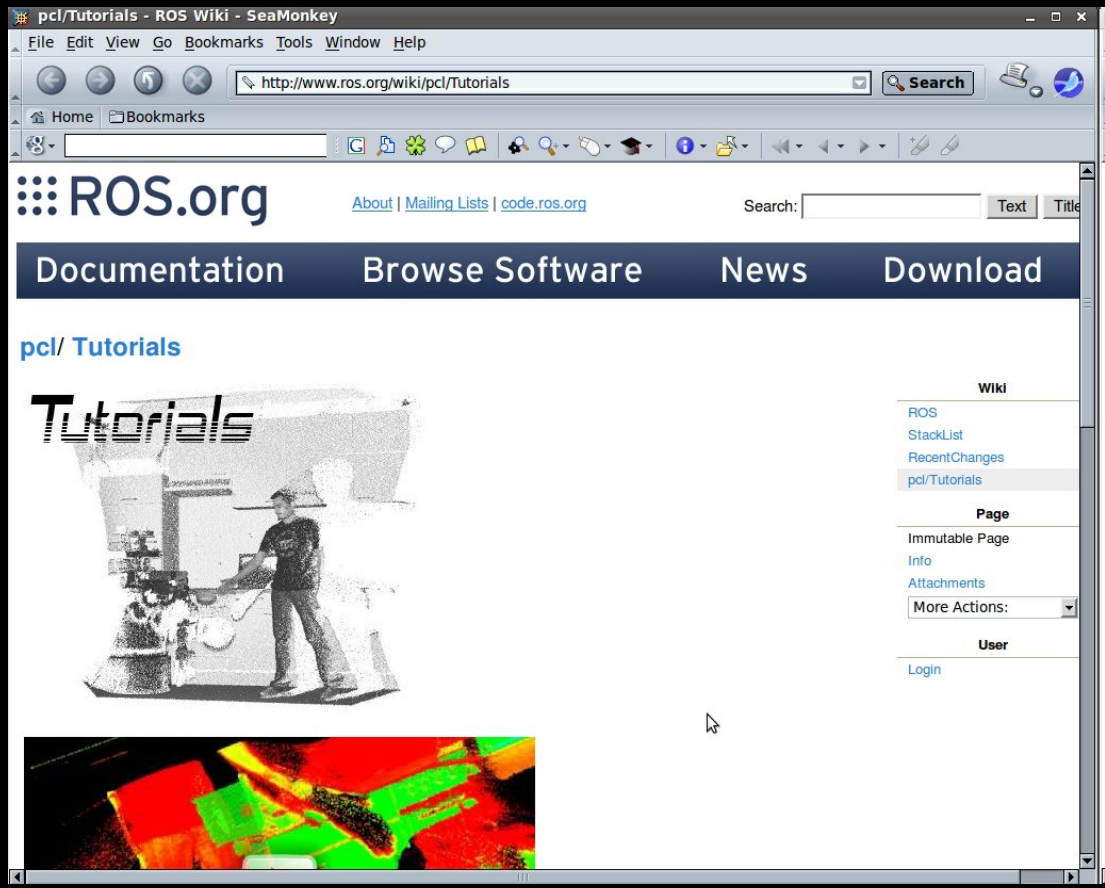
```
<launch>

  <node pkg="nodelet" type="nodelet" name="foo"
        args="normal_estimation␣NormalEstimation␣pcl_manager">
    <remap from="/normal_estimation/input"
           to="/voxel_grid/output" />
    <remap from="/normal_estimation/surface"
           to="/narrow_stereo_textured/points" />
    <rosparam>
      # -[ Mandatory parameters
      # Set either 'k_search' or 'radius_search'
      k_search: 0
      radius_search: 0.1
      # Set the spatial locator. Possible values are:
      # 0 (ANN), 1 (FLANN), 2 (organized)
      spatial_locator: 0
    </rosparam>
  </node>
...
</launch>
```

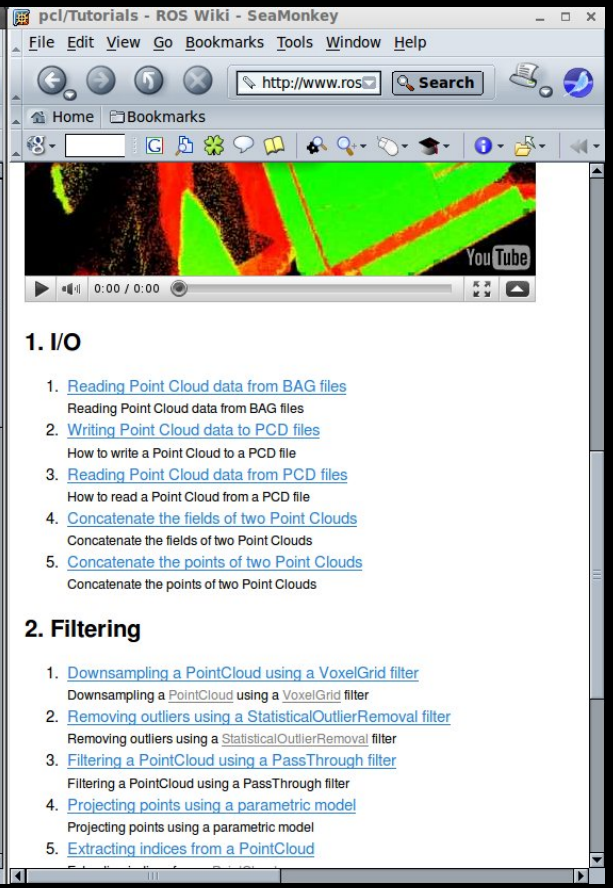
# PCL - Table Object Detector

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

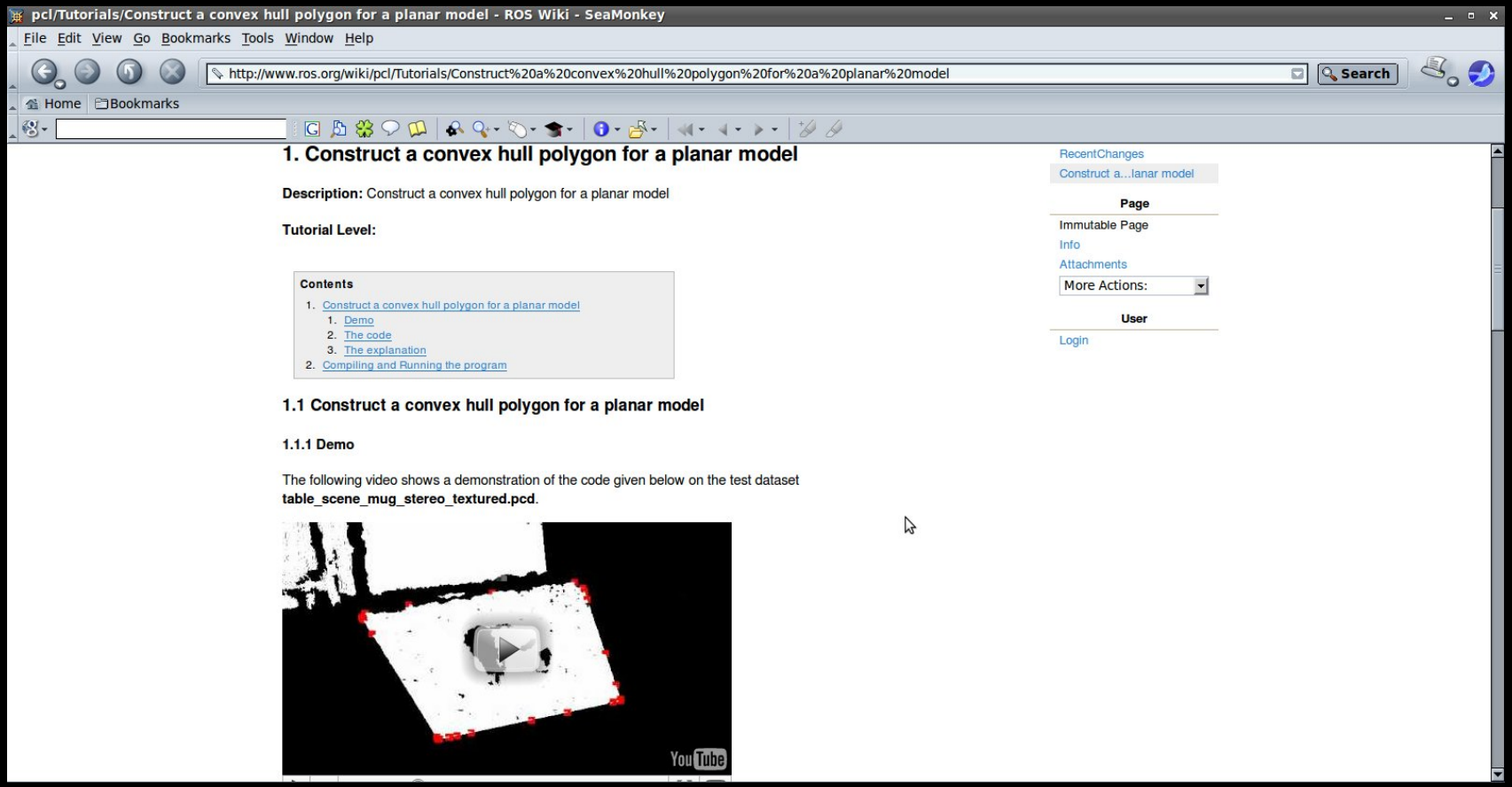


# PCL - Web tutorials (1/3)



# PCL - Web tutorials (2/3)



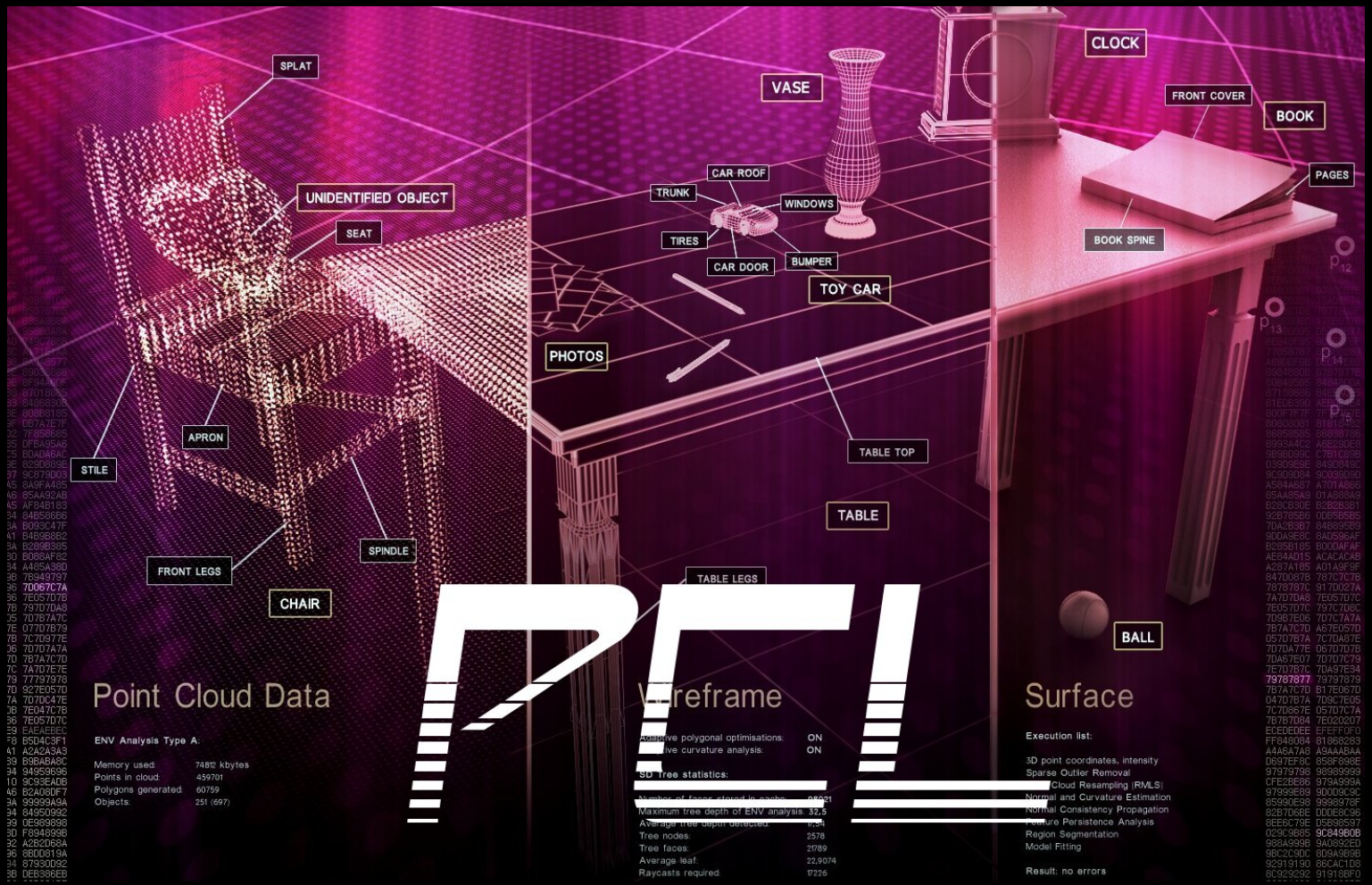
# PCL - Web tutorials (3/3)

Demo

`http://www.ros.org/wiki/pcl/Tutorials`

# Questions?

Thanks!




http://www.ros.org/wiki/pcl