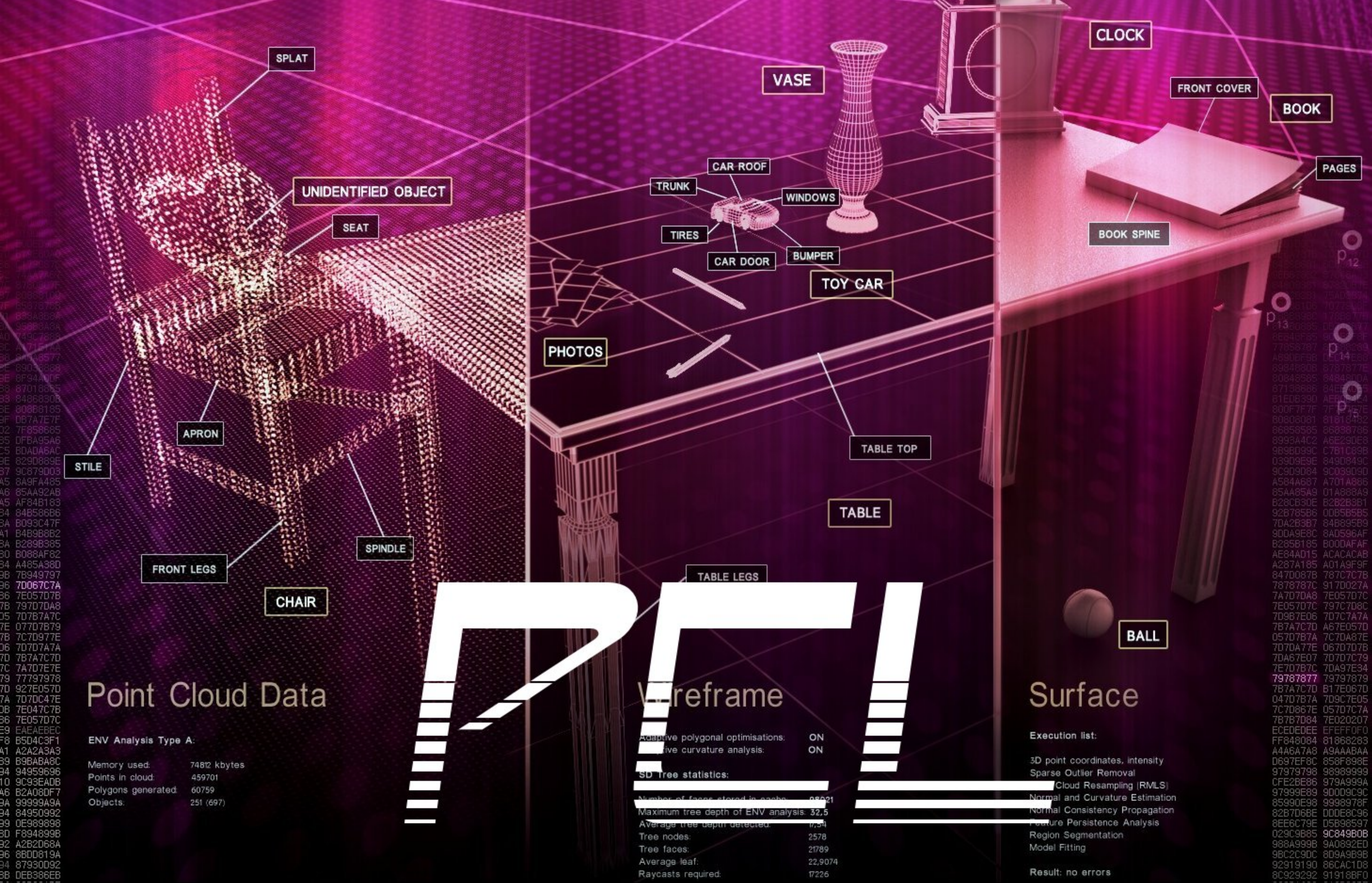


Point Cloud Library

Radu Bogdan Rusu
Research Scientist, Willow Garage

(your friendly 3D processing library. 0.1)

Why PCL? Why PointCloud2?

I. sensor_msgs/PointCloud => inefficient

- not aligned (e.g., SSE)
- no way to represent organized data
- large overhead
- the world is “float” ☺

```
Header header
geometry_msgs/Point32[] points
ChannelFloat32[] channels
```

```
float32 x
float32 y
float32 z
```

```
string name
float32[] values
```

Why PCL? Why PointCloud2?

II. point_cloud_mapping => C library

- easy to use for simple stuff

  void **computeCentroid** (const sensor_msgs::PointCloud &points, sensor_msgs::PointCloud &centroid);

- hard for more “complicated” things

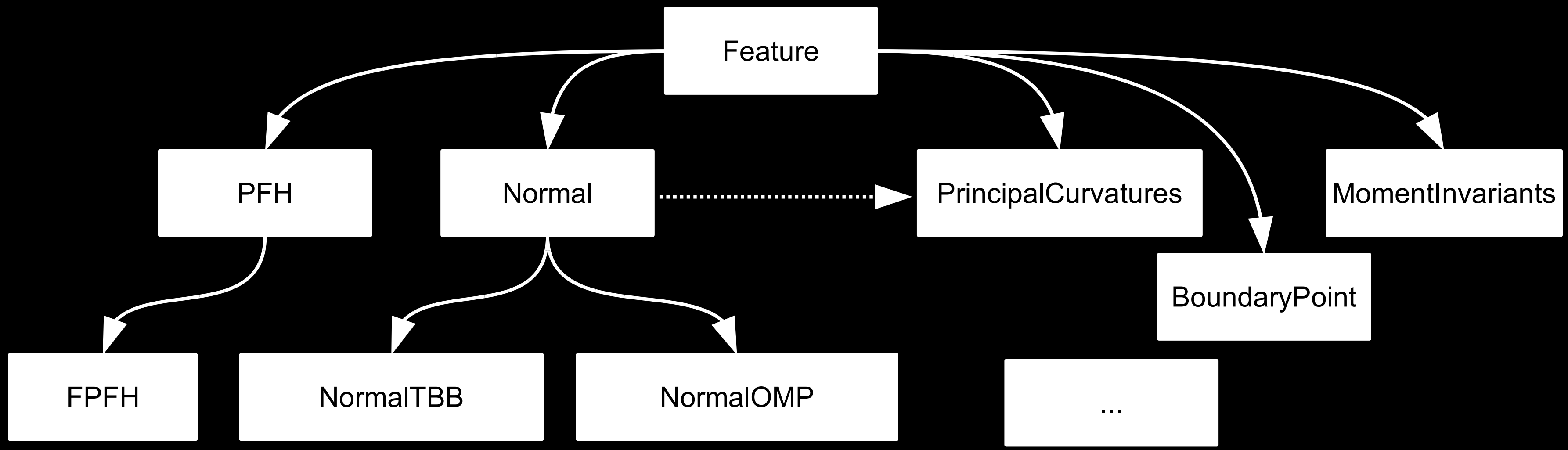
  void **computeOrganizedPointCloudNormalsWithFiltering** (sensor_msgs::PointCloud &points, const sensor_msgs::PointCloud &surface, int k, int downsample_factor, int width, int height,double max_z, double min_angle, double max_angle, const geometry_msgs::Point32 &viewpoint);

- no way to store state, hard to reuse code as “building blocks”, too many nodes
- worst possible example: N versions of the same thing (e.g., same planar detector copied in 6 packages!) => maintainability is an issue! If 1 copy is updated, the rest are left behind!

Why PCL? Why PointCloud2?

III. more on reusability, architecture

- Algorithmically:
  - ➔ door detection = table detection = wall detection = ...
  - ➔ the only thing that changes is: parameters (constraints)!
- Inheritance simplifies development and testing (C++ rocks!)



Why PCL? Why PointCloud2?

III. more on reusability, architecture

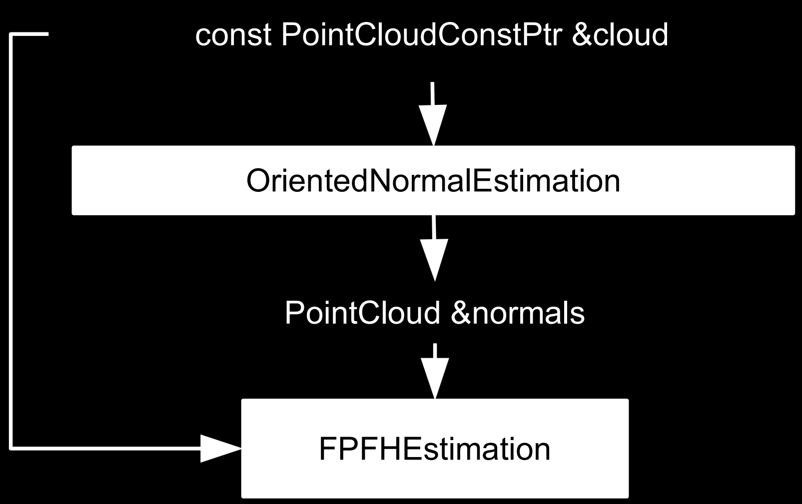
- Algorithmically:
  - ➔ door detection = table detection = wall detection = ...
  - ➔ the only thing that changes is: parameters (constraints)!
- Inheritance simplifies development and testing (C++ rocks!)

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

Why PCL? Why PointCloud2?

IV. PointCloud is the largest data type in ROS. Need ways to pass data without sacrificing reusability, performance, etc...

- Ideally, all in *shared memory*! N processing building blocks, joint access.



Solutions!

I. sensor_msgs/PointCloud2
  - binary blob + channel/field description

II. PCL
  - fully templated modern C++ library, SSE optimizations (Eigen backend + custom), OpenMP and TBB enabled
  - data passing through Boost shared pointers

III. Nodelets
  - "nodes within nodes"
  - same ROS API as nodes, dynamically loadable
  - optimizations for zero-copy Boost shared_ptr passing

sensor_msgs/PointCloud2

```
#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
```

sensor_msgs/PointCloud2

```
#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
```

sensor_msgs/PointCloud2

- 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
```
- 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
```

sensor_msgs/PointCloud2

- Binary blobs are hard to work with
- custom converter, Publisher/Subscriber, transport tools, filters, etc, similar to images (the whole nine yards)
- templated types: PointCloud2 → PointCloud<**T**>
- examples of **T**:
  - ```
    struct PointXYZ
    {
      float x;
      float y;
      float z;
    }
    ```
  - ```
    struct Normal
    {
      float normal[3];
      float curvature;
    }
    ```

Nodelet

- dynamically loadable C++ ROS entity
- uses ROS API for communication
  - Publish/Subscribe
  - parameters
  - remappings
- walks like a node, talks like a node, except:
  - ***nodelets*** reside in the same process, while
  - each ***node*** is a separate process
  - because it's single-process, communication can be efficient!!! Instead of publishing data, we publish Boost shared pointers to data! (roscpp optimizations)

Nodelet Implementation Example

```
class Plus : public nodelet::Nodelet
{
  void init()
  {
    private_nh_.getParam("value", value_);
    pub=private_nh_.advertise<std_msgs::Float64>("out", 10);
    sub=private_nh_.subscribe("in", 10, &Plus::callback, this);
  };
  void callback(const std_msgs::Float64::ConstPtr& input)
  {
    boost::shared_ptr<std_msgs::Float64> output;
    output = boost::make_shared<std_msgs::Float64> ();
    output->data= input->data + value_;
    pub.publish(output);
  };
  ros::Publisher pub;
  ros::Subscriber sub;
  double value_;
};
```

Dynamic Loading with Pluginlib

- Library declaration
  PLUGINLIB_DECLARE_CLASS
  (nodelet_tutorial_math, Plus, nodelet_tutorial_math::Plus, nodelet::Nodelet);
- Manifest Export

```
<export>
  <nodelet plugin="${prefix}/nodelet_math.xml"/>
</export>
```

- Plugin description file

```
<library path="lib/libnodelet_math">
  <class name="nodelet_tutorial_math/Plus"
         type="nodelet_tutorial_math::Plus"
         base_class_type="nodelet::Nodelet">
  <description>Add a value and republish. </description>
  </class>
</library>
```

Running from the command line

```
$ rosrun nodelet standalone_nodelet
$ rosrun nodelet nodelet foo_name nodelet_tutorial_math/Plus standalone_nodelet
```

- This will run a nodelet manager then call into it to dynamically load a nodelet.

Nodelets in Launch Files

- An example of launching a nodelet from a file

```
<launch>
  <node pkg="nodelet"
        type="standalone_nodelet"
        name="nodelet_manager"/>
  <node pkg="nodelet"
        type="nodelet"
        name="$(anon nodelet)"
 args="Plus3 nodelet_tutorial_math/Plus nodelet_manager">
    <param name="value" type="double" value="2.5"/>
    <remap from="Plus3/in" to="Plus2/out"/>
  </node>
</launch>
```

PCL

- Features:
  - complete Eigen backend (patches go upstream)
  - SSE, OpenMP, TBB optimizations (future CUDA)
  - fully templated on the point type

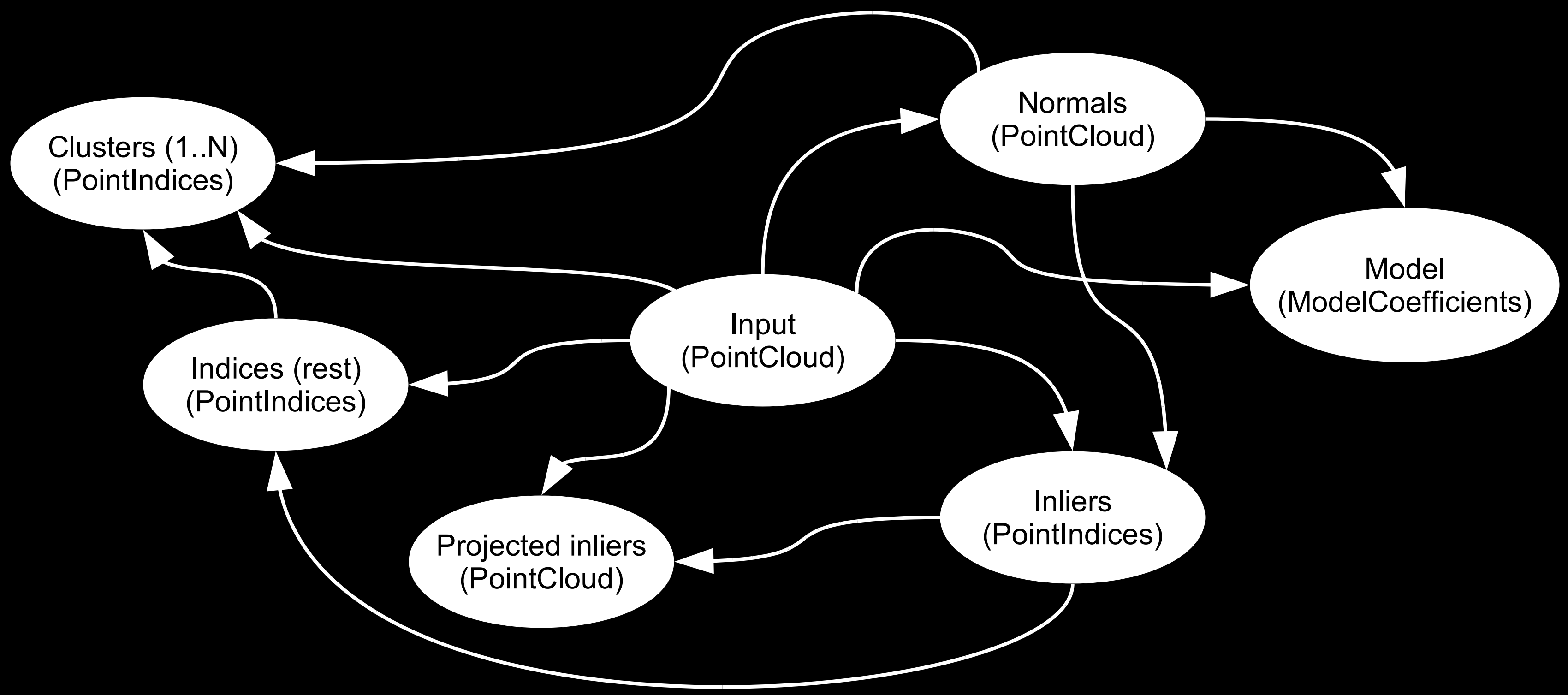
```
template <typename PointT>
class Foo: public boost::enable_shared_from_this <Foo<PointT> >, public nodelet::Nodelet
{
  typedef  point_cloud::PointCloud<PointT> PointCloud;
…
}
PointCloud<pcl::PointXYZ> cloud;
cloud.points[i].x = ...
Foo<pcl::PointXYZ> bar;
```

PCL

- Structure:
  - C++ libraries, with a few simple C-style methods: 6 in progress + more soon
  - libPCL_features, libPCL_surface, libPCL_filters, libPCL_segmentation, libPCL_io – tested/testing and documented
  - libPCL_registration – design in progress
  - Features, segmentation, io → unit tested
  - C++ classes are templated building blocks (nodelets!) via inheritance and init()
  - “standard” node examples still exist

PCL

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



PCL

- Misc, stats:
  - Approx 77 classes, over 25k lines of code (10-15% done by my estimation in terms of the functionality that I want ☺ - oh well, need more coders!)
  - External dependencies (for now) on eigen, cminpack, ANN, FLANN, TBB
  - Internal dependencies (excluding the obvious) on dynamic_reconfigure, message_filters
  - ...

Usage examples

- Simple downsampling and filtering

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

  <!-- Downsample the data -->
  <node pkg="nodelet" type="nodelet" name="foo" args="voxel_gridVoxelGrid 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
     filter_field_name: "z"  # The field name that holds distance values (for filtering)
     filter_limit_min: 0.8   # points closer than 0.8m from the viewpoint will not be considered
     filter_limit_max: 5.0   # points closer than 5.0m from the viewpoint will not be considered
     use_indices: false      # false by default
    </rosparam>
  </node>
…
</launch>
```

Usage examples

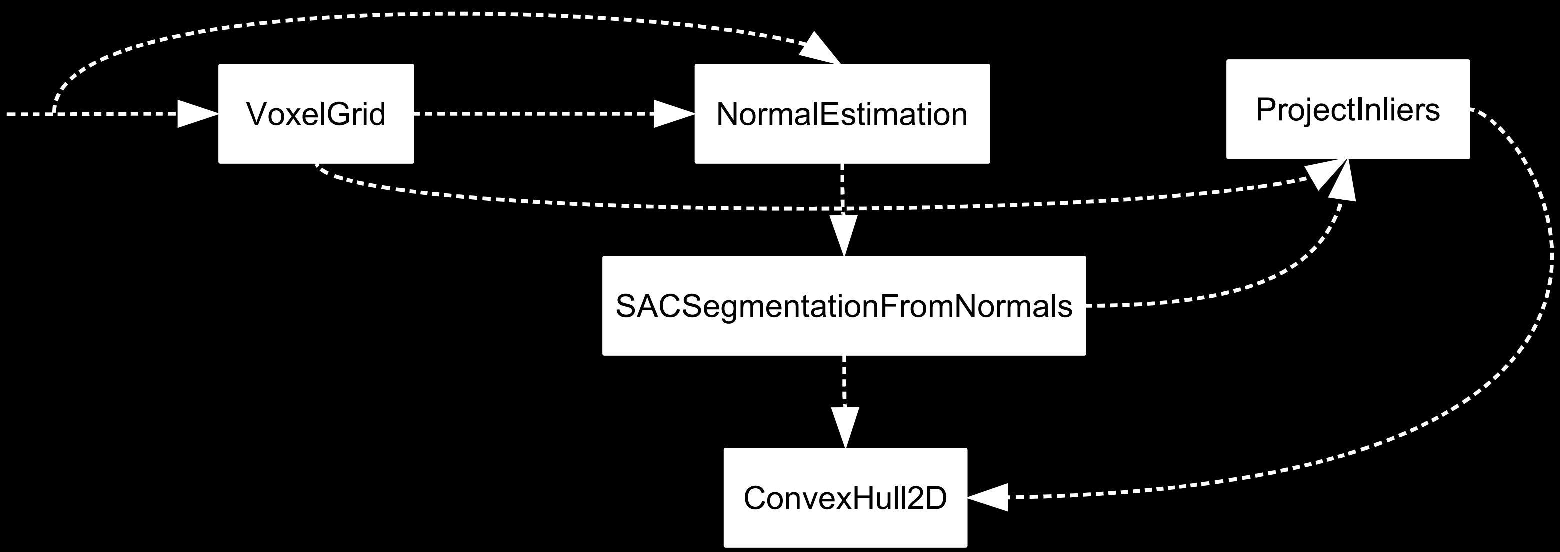
- Normal estimation

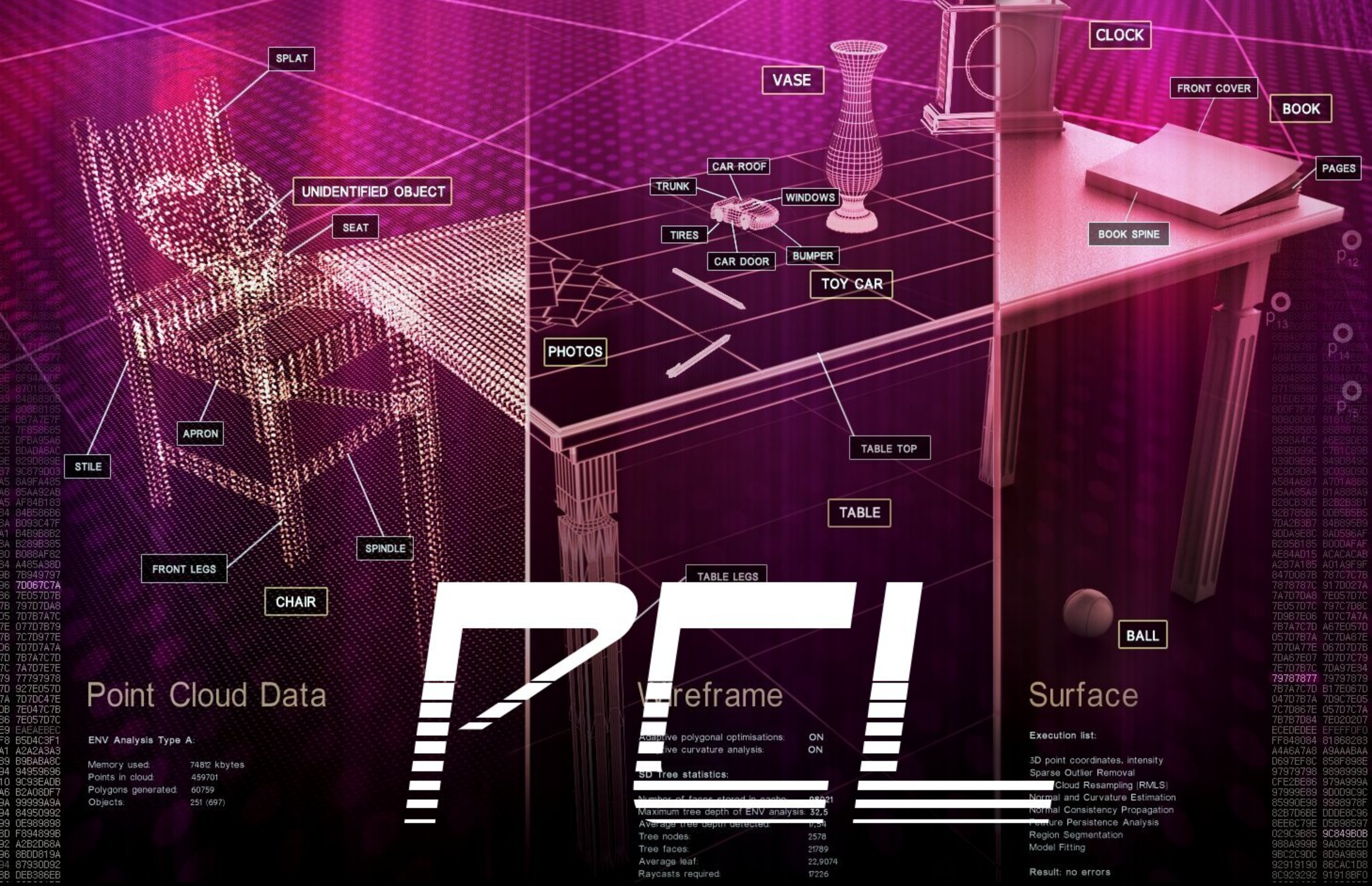
```
<launch>
  <node pkg="nodelet" type="standalone_nodelet" name="pcl_manager" output="screen" />
   <!-- Estimate point normals -->
<node pkg="nodelet" type="nodelet" name="foo" args="normal_estimation NormalEstimation pcl_manager">
    <remap from="/normal_estimation/input"   to="/voxel_grid/output" />
    <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

      # -[ Optional parameters
      use_indices: false    # false by default
      use_surface: false    # false by default
    </rosparam>
  </node>
…
</launch>
```

Usage examples

- (improved) table segmentation (constrained!)





Point Cloud Library

Thanks to Tully Foote, Patrick Mihelich, Josh Faust, Kurt Konolige from Willow Garage, and our colleagues at TUM!