> 仅做交流学习使用

NDT

> 第十一章
> https://gitee.com/suyunzzz/pcl_example_code/tree/master/%E7%AC%AC%E5%8D%81%E4%B8%80%E7%AB%A0

如何使用NDT（书11.2.3）

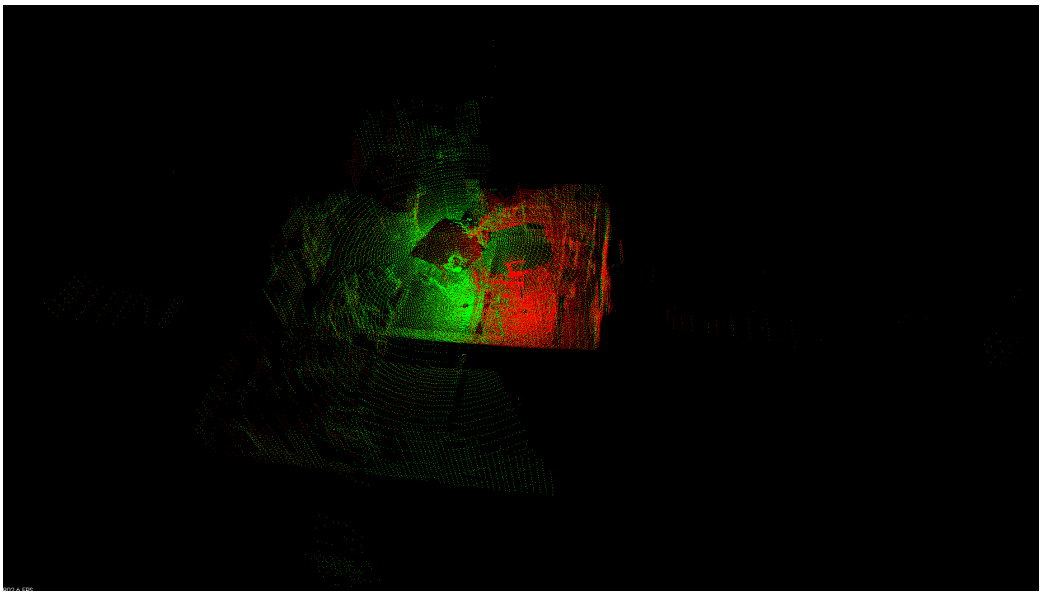
代码

```
#include <iostream>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/registration/ndt.h>
#include <pcl/filters/approximate_voxel_grid.h>
#include <pcl/visualization/pcl_visualizer.h>
```

```cpp
#include <boost/thread/thread.hpp>
int
main (int argc, char** argv)
{
  //加载房间的第一次扫描
  pcl::PointCloud<pcl::PointXYZ>::Ptr target_cloud (new pcl::PointCloud<pcl::PointXYZ>);
  if (pcl::io::loadPCDFile<pcl::PointXYZ> ("room_scan1.pcd", *target_cloud) == -1)
  {
  PCL_ERROR ("Couldn't read file room_scan1.pcd \n");
  return (-1);
  }
  std::cout << "Loaded " << target_cloud->size () << " data points from room_scan1.pcd" << std::endl;
  //加载从新视角得到的房间的第二次扫描
  pcl::PointCloud<pcl::PointXYZ>::Ptr input_cloud (new pcl::PointCloud<pcl::PointXYZ>);
  if (pcl::io::loadPCDFile<pcl::PointXYZ> ("room_scan2.pcd", *input_cloud) == -1)
  {
  PCL_ERROR ("Couldn't read file room_scan2.pcd \n");
  return (-1);
  }
  std::cout << "Loaded " << input_cloud->size () << " data points from room_scan2.pcd" << std::endl;
  //将输入的扫描过滤到原始尺寸的大概10%以提高匹配的速度。
  pcl::PointCloud<pcl::PointXYZ>::Ptr filtered_cloud (new pcl::PointCloud<pcl::PointXYZ>);
  pcl::ApproximateVoxelGrid<pcl::PointXYZ> approximate_voxel_filter;
  approximate_voxel_filter.setLeafSize (0.2, 0.2, 0.2);
  approximate_voxel_filter.setInputCloud (input_cloud);
  approximate_voxel_filter.filter (*filtered_cloud);
  std::cout << "Filtered cloud contains " << filtered_cloud->size ()
  << " data points from room_scan2.pcd" << std::endl;
  //初始化正态分布变换（NDT）
  pcl::NormalDistributionsTransform<pcl::PointXYZ, pcl::PointXYZ> ndt;
  //设置依赖尺度NDT参数
  //为终止条件设置最小转换差异
  ndt.setTransformationEpsilon (0.01);
  //为More-Thuente线搜索设置最大步长
  ndt.setStepSize (0.1);
  //设置NDT网格结构的分辨率（VoxelGridCovariance）
  ndt.setResolution (1.0);
  //设置匹配迭代的最大次数
  ndt.setMaximumIterations (35);
  // 设置要配准的点云
  ndt.setInputCloud (filtered_cloud);
  //设置点云配准目标
  ndt.setInputTarget (target_cloud);
  //设置使用机器人测距法得到的初始对准估计结果
  Eigen::AngleAxisf init_rotation (0.6931, Eigen::Vector3f::UnitZ ());
  Eigen::Translation3f init_translation (1.79387, 0.720047, 0);
  Eigen::Matrix4f init_guess = (init_translation * init_rotation).matrix ();
```

```
54  //计算需要的刚体变换以便将输入的点云匹配到目标点云
55  pcl::PointCloud<pcl::PointXYZ>::Ptr output_cloud (new pcl::PointCloud<pcl::PointXYZ>);
56  ndt.align (*output_cloud, init_guess);
57  std::cout << "Normal Distributions Transform has converged:" << ndt.hasConverged ()
58  << " score: " << ndt.getFitnessScore () << std::endl;
59  //使用创建的变换对未过滤的输入点云进行变换
60  pcl::transformPointCloud (*input_cloud, *output_cloud, ndt.getFinalTransformation ());
61  //保存转换的输入点云
62  pcl::io::savePCDFileASCII ("room_scan2_transformed.pcd", *output_cloud);
63  // 初始化点云可视化界面
64  boost::shared_ptr<pcl::visualization::PCLVisualizer>
65  viewer_final (new pcl::visualization::PCLVisualizer ("3D Viewer"));
66  viewer_final->setBackgroundColor (0, 0, 0);
67  //对目标点云着色（红色）并可视化
68  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ>
69  target_color (target_cloud, 255, 0, 0);
70  viewer_final->addPointCloud<pcl::PointXYZ> (target_cloud, target_color, "target cloud");
71  viewer_final->setPointCloudRenderingProperties (pcl::visualization::PCL_VISUALIZER_POINT_
SIZE,
72  1, "target cloud");
73  //对转换后的目标点云着色（绿色）并可视化
74  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ>
75  output_color (output_cloud, 0, 255, 0);
76  viewer_final->addPointCloud<pcl::PointXYZ> (output_cloud, output_color, "output cloud");
77  viewer_final->setPointCloudRenderingProperties (pcl::visualization::PCL_VISUALIZER_POINT_
SIZE,
78  1, "output cloud");
79  // 启动可视化
80  viewer_final->addCoordinateSystem (1.0);
81  viewer_final->initCameraParameters ();
82  //等待直到可视化窗口关闭。
83  while (!viewer_final->wasStopped ())
84  {
85  viewer_final->spinOnce (100);
86  boost::this_thread::sleep (boost::posix_time::microseconds (100000));
87  }
88  return (0);
89 }
90
```

可视化



刚性物体的位姿估计（书11.2.5）

代码

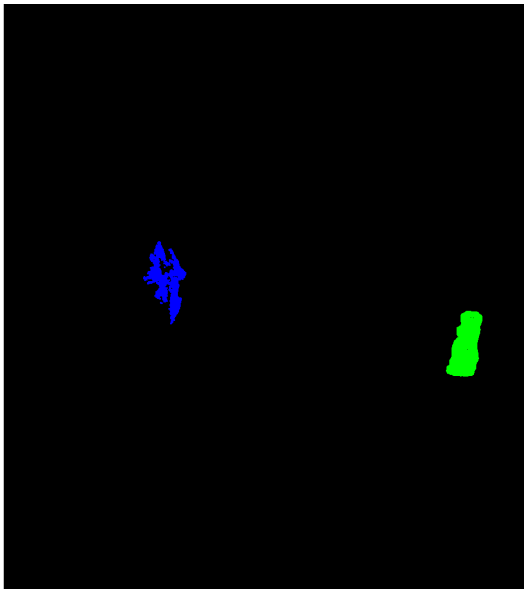
```
#include <Eigen/Core>
#include <pcl/point_types.h>
#include <pcl/point_cloud.h>
#include <pcl/common/time.h>
#include <pcl/console/print.h>
#include <pcl/features/normal_3d.h>
#include <pcl/features/fpfh.h>
#include <pcl/filters/filter.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/io/pcd_io.h>
#include <pcl/registration/icp.h>
#include <pcl/registration/sample_consensus_prerejective.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/visualization/pcl_visualizer.h>

// Types
typedef pcl::PointNormal PointNT;
typedef pcl::PointCloud<PointNT> PointCloudT;
typedef pcl::FPFHSignature33 FeatureT;
typedef pcl::FPFHEstimation<PointNT,PointNT,FeatureT> FeatureEstimationT;
typedef pcl::PointCloud<FeatureT> FeatureCloudT;
typedef pcl::visualization::PointCloudColorHandlerCustom<PointNT> ColorHandlerT;

// Align a rigid object to a scene with clutter and occlusions
int
main (int argc, char **argv)
{
 // Point clouds
 PointCloudT::Ptr object (new PointCloudT);
 PointCloudT::Ptr object_aligned (new PointCloudT);
 PointCloudT::Ptr scene (new PointCloudT);
```

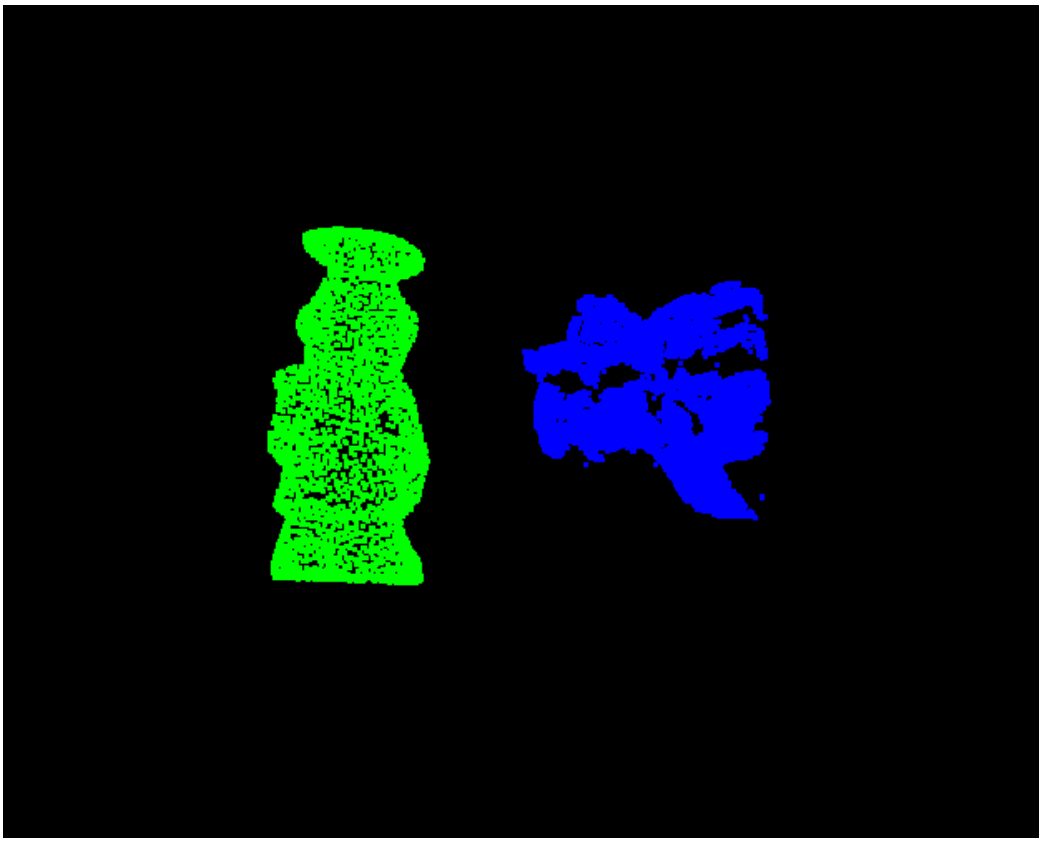
```
32  FeatureCloudT::Ptr object_features (new FeatureCloudT);
33  FeatureCloudT::Ptr scene_features (new FeatureCloudT);
34
35  // Get input object and scene
36  if (argc != 3)
37  {
38  pcl::console::print_error ("Syntax is: %s object.pcd scene.pcd\n", argv[0]);
39  return (1);
40  }
41
42  // 加载目标物体和场景点云
43  pcl::console::print_highlight ("Loading point clouds...\n");
44  if (pcl::io::loadPCDFile<PointNT> (argv[1], *object) < 0 ||
45  pcl::io::loadPCDFile<PointNT> (argv[2], *scene) < 0)
46  {
47  pcl::console::print_error ("Error loading object/scene file!\n");
48  return (1);
49  }
50
51  // 下采样
52  pcl::console::print_highlight ("Downsampling...\n");
53  pcl::VoxelGrid<PointNT> grid;
54  const float leaf = 0.005f;
55  grid.setLeafSize (leaf, leaf, leaf);
56  grid.setInputCloud (object);
57  grid.filter (*object);
58  grid.setInputCloud (scene);
59  grid.filter (*scene);
60
61  // 估计场景法线
62  pcl::console::print_highlight ("Estimating scene normals...\n");
63  pcl::NormalEstimation<PointNT,PointNT> nest;
64  nest.setRadiusSearch (0.01);
65  nest.setInputCloud (scene);
66  nest.compute (*scene);
67
68  // 特征估计
69  pcl::console::print_highlight ("Estimating features...\n");
70  FeatureEstimationT fest;
71  fest.setRadiusSearch (0.025);
72  fest.setInputCloud (object);
73  fest.setInputNormals (object);
74  fest.compute (*object_features);
75  fest.setInputCloud (scene);
76  fest.setInputNormals (scene);
77  fest.compute (*scene_features);
78
79  // 实施配准
80  pcl::console::print_highlight ("Starting alignment...\n");
```

```
  pcl::SampleConsensusPrerejective<PointNT,PointNT,FeatureT> align;
  align.setInputSource (object);
  align.setSourceFeatures (object_features);
  align.setInputTarget (scene);
  align.setTargetFeatures (scene_features);
  align.setMaximumIterations (50000); // 采样一致性迭代次数
  align.setNumberOfSamples (3); // 创建假设所需的样本数
  align.setCorrespondenceRandomness (5); // 使用的临近特征点的数目
  align.setSimilarityThreshold (0.9f); // 多边形边长度相似度阈值
  align.setMaxCorrespondenceDistance (2.5f * 0.005); // 判断是否为内点的距离阈值
  align.setInlierFraction (0.25f); //接受位姿假设所需的内点比例
  {
  pcl::ScopeTime t("Alignment");
  align.align (*object_aligned);
  }

  if (align.hasConverged ())
  {
  // Print results
   printf ("\n");
   Eigen::Matrix4f transformation = align.getFinalTransformation ();
   pcl::console::print_info (" | %6.3f %6.3f %6.3f | \n", transformation (0,0), transformat
ion (0,1), transformation (0,2));
   pcl::console::print_info ("R = | %6.3f %6.3f %6.3f | \n", transformation (1,0), transfor
mation (1,1), transformation (1,2));
   pcl::console::print_info (" | %6.3f %6.3f %6.3f | \n", transformation (2,0), transformat
ion (2,1), transformation (2,2));
   pcl::console::print_info ("\n");
   pcl::console::print_info ("t = < %0.3f, %0.3f, %0.3f >\n", transformation (0,3), transfo
rmation (1,3), transformation (2,3));
   pcl::console::print_info ("\n");
   pcl::console::print_info ("Inliers: %i/%i\n", align.getInliers ().size (), object->size
());

   // Show alignment
   pcl::visualization::PCLVisualizer visu("点云库PCL学习教程第二版-鲁棒位姿估计");
   int v1(0),v2(0);
   visu.createViewPort(0,0,0.5,1,v1);
   visu.createViewPort(0.5,0,1,1,v2);
   visu.setBackgroundColor(255,255,255,v1);
   visu.addPointCloud (scene, ColorHandlerT (scene, 0.0, 255.0, 0.0), "scene",v1);
   visu.addPointCloud (object_aligned, ColorHandlerT (object_aligned, 0.0, 0.0, 255.0), "ob
ject_aligned",v1);

   visu.addPointCloud(object,ColorHandlerT (object, 0.0, 255.0, 0.0), "object_before_aligne
d",v2);
   visu.addPointCloud(scene,ColorHandlerT (scene, 0.0, 0.0, 255.0), "scene_v2",v2);
   visu.setPointCloudRenderingProperties(pcl::visualization::PCL_VISUALIZER_POINT_SIZE,3,"s
cene");
   visu.setPointCloudRenderingProperties(pcl::visualization::PCL_VISUALIZER_POINT_SIZE,3,"o
bject_aligned");
```

```
  visu.setPointCloudRenderingProperties(pcl::visualization::PCL_VISUALIZER_POINT_SIZE,3,"object_before_aligned");
  visu.setPointCloudRenderingProperties(pcl::visualization::PCL_VISUALIZER_POINT_SIZE,3,"scene_v2");
  visu.spin ();
  }
  else
  {
  pcl::console::print_error ("Alignment failed!\n");
  return (1);
  }

  return (0);
}
```

可视化





如何将扫描数据与模板对象进行配准(11.3.1)

代码

```
#include <limits>
```

```cpp
#include <fstream>
#include <vector>
#include <Eigen/Core>
#include <pcl/point_types.h>
#include <pcl/point_cloud.h>
#include <pcl/io/pcd_io.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/filters/passthrough.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/features/normal_3d.h>
#include <pcl/features/fpfh.h>
#include <pcl/registration/ia_ransac.h>

class FeatureCloud
{
 public:
  // A bit of shorthand
  typedef pcl::PointCloud<pcl::PointXYZ> PointCloud;
  typedef pcl::PointCloud<pcl::Normal> SurfaceNormals;
  typedef pcl::PointCloud<pcl::FPFHSignature33> LocalFeatures;
  typedef pcl::search::KdTree<pcl::PointXYZ> SearchMethod;

  FeatureCloud () :
  search_method_xyz_ (new SearchMethod),
  normal_radius_ (0.02f),
  feature_radius_ (0.02f)
  {}

  ~FeatureCloud () {}

  // Process the given cloud
  void
  setInputCloud (PointCloud::Ptr xyz)
  {
  xyz_ = xyz;
  processInput ();
  }

  // Load and process the cloud in the given PCD file
  void
  loadInputCloud (const std::string &pcd_file)
  {
  xyz_ = PointCloud::Ptr (new PointCloud);
  pcl::io::loadPCDFile (pcd_file, *xyz_);
  processInput ();
  }

  // Get a pointer to the cloud 3D points
  PointCloud::Ptr
```

```
51  getPointCloud () const
52  {
53  return (xyz_);
54  }
55
56  // Get a pointer to the cloud of 3D surface normals
57  SurfaceNormals::Ptr
58  getSurfaceNormals () const
59  {
60  return (normals_);
61  }
62
63  // Get a pointer to the cloud of feature descriptors
64  LocalFeatures::Ptr
65  getLocalFeatures () const
66  {
67  return (features_);
68  }
69
70  protected:
71  // Compute the surface normals and local features
72  void
73  processInput ()
74  {
75  computeSurfaceNormals ();
76  computeLocalFeatures ();
77  }
78
79  // Compute the surface normals
80  void
81  computeSurfaceNormals ()
82  {
83  normals_ = SurfaceNormals::Ptr (new SurfaceNormals);
84
85  pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> norm_est;
86  norm_est.setInputCloud (xyz_);
87  norm_est.setSearchMethod (search_method_xyz_);
88  norm_est.setRadiusSearch (normal_radius_);
89  norm_est.compute (*normals_);
90  }
91
92  // Compute the local feature descriptors
93  void
94  computeLocalFeatures ()
95  {
96  features_ = LocalFeatures::Ptr (new LocalFeatures);
97
98  pcl::FPFHEstimation<pcl::PointXYZ, pcl::Normal, pcl::FPFHSignature33> fpfh_est;
99  fpfh_est.setInputCloud (xyz_);
```

```
 fpfh_est.setInputNormals (normals_);
 fpfh_est.setSearchMethod (search_method_xyz_);
 fpfh_est.setRadiusSearch (feature_radius_);
 fpfh_est.compute (*features_);
 }

 private:
 // Point cloud data
 PointCloud::Ptr xyz_;
 SurfaceNormals::Ptr normals_;
 LocalFeatures::Ptr features_;
 SearchMethod::Ptr search_method_xyz_;

 // Parameters
 float normal_radius_;
 float feature_radius_;
};

class TemplateAlignment
{
 public:

 // A struct for storing alignment results
 struct Result
 {
 float fitness_score;
 Eigen::Matrix4f final_transformation;
 EIGEN_MAKE_ALIGNED_OPERATOR_NEW
 };

 TemplateAlignment () :
 min_sample_distance_ (0.05f),
 max_correspondence_distance_ (0.01f*0.01f),
 nr_iterations_ (500)
 {
 // Intialize the parameters in the Sample Consensus Intial Alignment (SAC-IA) algorithm
 sac_ia_.setMinSampleDistance (min_sample_distance_);
 sac_ia_.setMaxCorrespondenceDistance (max_correspondence_distance_);
 sac_ia_.setMaximumIterations (nr_iterations_);
 }

 ~TemplateAlignment () {}

 // Set the given cloud as the target to which the templates will be aligned
 void
 setTargetCloud (FeatureCloud &target_cloud)
 {
 target_ = target_cloud;
```

```
148   sac_ia_.setInputTarget (target_cloud.getPointCloud ());
149   sac_ia_.setTargetFeatures (target_cloud.getLocalFeatures ());
150   }
151
152   // Add the given cloud to the list of template clouds
153   void
154   addTemplateCloud (FeatureCloud &template_cloud)
155   {
156   templates_.push_back (template_cloud);
157   }
158
159   // Align the given template cloud to the target specified by setTargetCloud ()
160   void
161   align (FeatureCloud &template_cloud, TemplateAlignment::Result &result)
162   {
163   sac_ia_.setInputCloud (template_cloud.getPointCloud ());
164   sac_ia_.setSourceFeatures (template_cloud.getLocalFeatures ());
165
166   pcl::PointCloud<pcl::PointXYZ> registration_output;
167   sac_ia_.align (registration_output);
168
169   result.fitness_score = (float) sac_ia_.getFitnessScore (max_correspondence_distance_);
170   result.final_transformation = sac_ia_.getFinalTransformation ();
171   }
172
173   // Align all of template clouds set by addTemplateCloud to the target specified by setTa
rgetCloud ()
174   void
175   alignAll (std::vector<TemplateAlignment::Result, Eigen::aligned_allocator<Result> > &res
ults)
176   {
177   results.resize (templates_.size ());
178   for (size_t i = 0; i < templates_.size (); ++i)
179   {
180   align (templates_[i], results[i]);
181   }
182   }
183
184   // Align all of template clouds to the target cloud to find the one with best alignment
score
185   int
186   findBestAlignment (TemplateAlignment::Result &result)
187   {
188   // Align all of the templates to the target cloud
189   std::vector<Result, Eigen::aligned_allocator<Result> > results;
190   alignAll (results);
191
192   // Find the template with the best (lowest) fitness score
193   float lowest_score = std::numeric_limits<float>::infinity ();
```

```
194   int best_template = 0;
195   for (size_t i = 0; i < results.size (); ++i)
196   {
197   const Result &r = results[i];
198   if (r.fitness_score < lowest_score)
199   {
200   lowest_score = r.fitness_score;
201   best_template = (int) i;
202   }
203   }
204
205   // Output the best alignment
206   result = results[best_template];
207   return (best_template);
208   }
209
210   private:
211   // A list of template clouds and the target to which they will be aligned
212   std::vector<FeatureCloud> templates_;
213   FeatureCloud target_;
214
215   // The Sample Consensus Initial Alignment (SAC-IA) registration routine and its paramete
rs
216   pcl::SampleConsensusInitialAlignment<pcl::PointXYZ, pcl::PointXYZ, pcl::FPFHSignature33>
sac_ia_;
217   float min_sample_distance_;
218   float max_correspondence_distance_;
219   int nr_iterations_;
220 };
221
222 // Align a collection of object templates to a sample point cloud
223 int
224 main (int argc, char **argv)
225 {
226   if (argc < 3)
227   {
228   printf ("No target PCD file given!\n");
229   return (-1);
230   }
231
232   // Load the object templates specified in the object_templates.txt file
233   std::vector<FeatureCloud> object_templates;
234   std::ifstream input_stream (argv[1]);
235   object_templates.resize (0);
236   std::string pcd_filename;
237   while (input_stream.good ())
238   {
239   std::getline (input_stream, pcd_filename);
240   if (pcd_filename.empty () || pcd_filename.at (0) == '#') // Skip blank lines or comments
```

```
241   continue;
242
243   FeatureCloud template_cloud;
244   template_cloud.loadInputCloud (pcd_filename);
245   object_templates.push_back (template_cloud);
246   }
247   input_stream.close ();
248
249   // Load the target cloud PCD file
250   pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::PointXYZ>);
251   pcl::io::loadPCDFile (argv[2], *cloud);
252
253   // Preprocess the cloud by...
254   // ...removing distant points
255   const float depth_limit = 1.0;
256   pcl::PassThrough<pcl::PointXYZ> pass;
257   pass.setInputCloud (cloud);
258   pass.setFilterFieldName ("z");
259   pass.setFilterLimits (0, depth_limit);
260   pass.filter (*cloud);
261
262   // ... and downsampling the point cloud
263   const float voxel_grid_size = 0.005f;
264   pcl::VoxelGrid<pcl::PointXYZ> vox_grid;
265   vox_grid.setInputCloud (cloud);
266   vox_grid.setLeafSize (voxel_grid_size, voxel_grid_size, voxel_grid_size);
267   vox_grid.filter (*cloud);
268
269   // Assign to the target FeatureCloud
270   FeatureCloud target_cloud;
271   target_cloud.setInputCloud (cloud);
272
273   // Set the TemplateAlignment inputs
274   TemplateAlignment template_align;
275   for (size_t i = 0; i < object_templates.size (); ++i)
276   {
277   template_align.addTemplateCloud (object_templates[i]);
278   }
279   template_align.setTargetCloud (target_cloud);
280
281   // Find the best template alignment
282   TemplateAlignment::Result best_alignment;
283   int best_index = template_align.findBestAlignment (best_alignment);
284   const FeatureCloud &best_template = object_templates[best_index];
285
286   // Print the alignment fitness score (values less than 0.00002 are good)
287   printf ("Best fitness score: %f\n", best_alignment.fitness_score);
288
289   // Print the rotation matrix and translation vector
```

```
  Eigen::Matrix3f rotation = best_alignment.final_transformation.block<3,3>(0, 0);
  Eigen::Vector3f translation = best_alignment.final_transformation.block<3,1>(0, 3);

  printf ("\n");
  printf (" | %6.3f %6.3f %6.3f | \n", rotation (0,0), rotation (0,1), rotation (0,2));
  printf ("R = | %6.3f %6.3f %6.3f | \n", rotation (1,0), rotation (1,1), rotation (1,2));
  printf (" | %6.3f %6.3f %6.3f | \n", rotation (2,0), rotation (2,1), rotation (2,2));
  printf ("\n");
  printf ("t = < %0.3f, %0.3f, %0.3f >\n", translation (0), translation (1), translation (2));

  // Save the aligned template for visualization
  pcl::PointCloud<pcl::PointXYZ> transformed_cloud;
  pcl::transformPointCloud (*best_template.getPointCloud (), transformed_cloud, best_alignment.final_transformation);
  pcl::io::savePCDFileBinary ("output.pcd", transformed_cloud);

  return (0);
}
```

可视化

分割

> 第十二章
> https://gitee.com/suyunzzz/pcl_example_code/tree/master/%E7%AC%AC%E5%8D%81%E4%BA%8C%E7%AB%A0

平面分割

代码

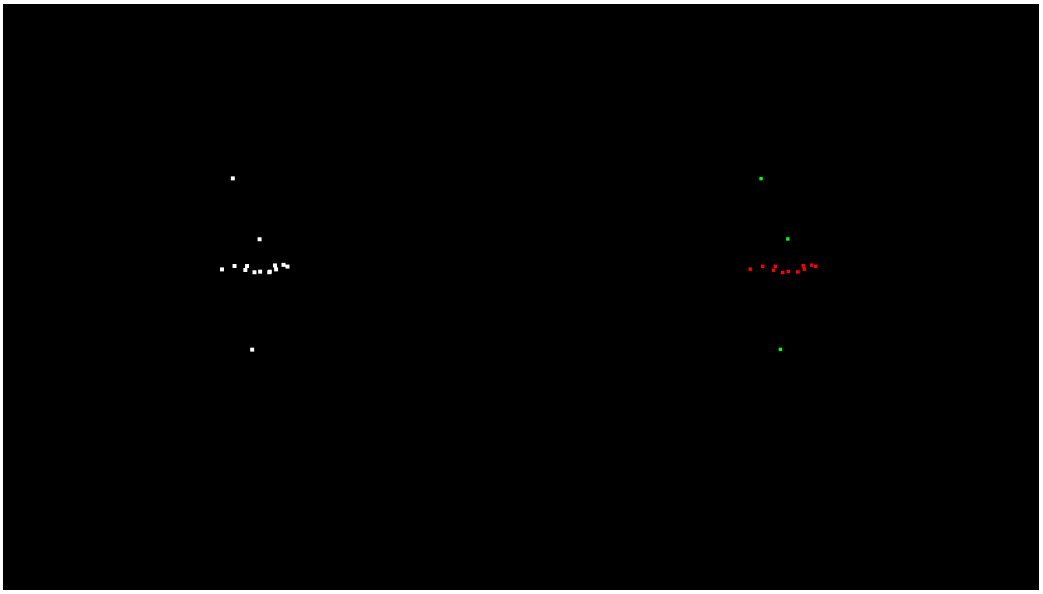
```
#include <iostream>
#include <pcl/ModelCoefficients.h>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/sample_consensus/method_types.h>
```

```
#include <pcl/sample_consensus/model_types.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/visualization/pcl_visualizer.h>
#include <pcl/filters/extract_indices.h>


// TODO 增加IndiceExtract模块来提取点


int
main (int argc, char** argv)
{
 pcl::PointCloud<pcl::PointXYZ> cloud;
//����������
cloud.width = 15;
 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 + 1.0f);
 cloud.points[i].y = 1024 * rand () / (RAND_MAX + 1.0f);
 cloud.points[i].z = 1.0;
 }
//���ü�������
cloud.points[0].z = 2.0;
cloud.points[3].z = -2.0;
cloud.points[6].z = 4.0;
 std::cerr << "Point cloud data: " << cloud.points.size () <<" points" << 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::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
//□□□□□□□□□□
 pcl::SACSegmentation<pcl::PointXYZ> seg;
 //��w����
 seg.setOptimizeCoefficients (true);
//��������
seg.setModelType (pcl::SACMODEL_PLANE);
 seg.setMethodType (pcl::SAC_RANSAC);
 seg.setDistanceThreshold (0.01);
 seg.setInputCloud (cloud.makeShared ());
 seg.segment (*inliers, *coefficients);

if (inliers->indices.size () == 0)
 {
```

```
 PCL_ERROR ("Could not estimate a planar model for the given dataset.");
 return (-1);
 }
std::cerr << "Model coefficients: " << coefficients->values[0] << " "
 <<coefficients->values[1] << " "
 <<coefficients->values[2] << " "
 <<coefficients->values[3] <<std::endl;

std::cerr << "Model inliers: " << inliers->indices.size () << std::endl;
for (size_t i = 0; i < inliers->indices.size (); ++i){
 std::cerr << inliers->indices[i] << " " <<cloud.points[inliers->indices[i]].x << " "
 <<cloud.points[inliers->indices[i]].y << " "
 <<cloud.points[inliers->indices[i]].z << std::endl;
}

// 提取
pcl::PointCloud<pcl::PointXYZ> cloud_plane;
pcl::ExtractIndices<pcl::PointXYZ> ex;
ex.setInputCloud(cloud.makeShared());
ex.setIndices(inliers);
ex.filter(cloud_plane);

// 提取2
pcl::PointCloud<pcl::PointXYZ> no_plane;
ex.setNegative(true);
ex.filter(no_plane);

// visualize
boost::shared_ptr<pcl::visualization::PCLVisualizer>
viewer_final (new pcl::visualization::PCLVisualizer ("3D Viewer"));
viewer_final->setBackgroundColor (0, 0, 0);

int v1;
viewer_final->createViewPort(0, 0, 0.5, 1.0, v1);
pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> white(cloud.makeShared(),
255, 255, 255);
viewer_final->addPointCloud(cloud.makeShared(), white, "raw" , v1);

int v2;
viewer_final->createViewPort(0.5, 0, 1.0, 1.0, v2);
pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ>
red(cloud_plane.makeShared(), 255, 0, 0);
pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ>
green(no_plane.makeShared(), 0, 255, 0);
viewer_final->addPointCloud(cloud_plane.makeShared(), red, "plane", v2);
viewer_final->addPointCloud(no_plane.makeShared(), green, "no_plane", v2);

while(!viewer_final->wasStopped()){
 viewer_final->spinOnce();
```

```
}


return (0);
}

```

可视化



圆柱体拟合

代码

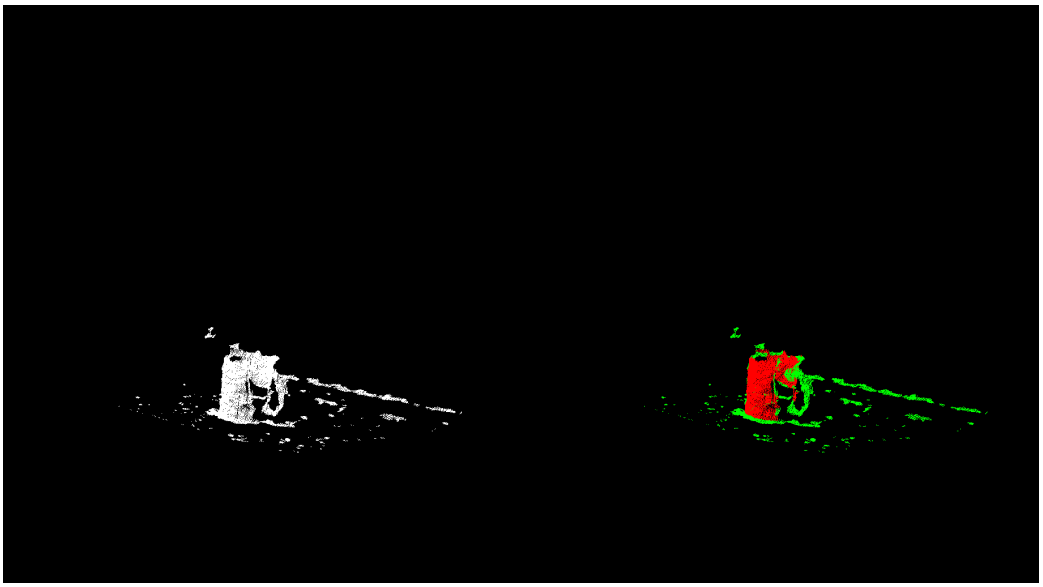
```
#include <pcl/ModelCoefficients.h>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/filters/extract_indices.h>
#include <pcl/filters/passthrough.h>
#include <pcl/features/normal_3d.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/segmentation/sac_segmentation.h>
 #include <pcl/visualization/pcl_visualizer.h>

 typedef pcl::PointXYZ PointT;


 void visualize_two_cloud(const pcl::PointCloud<PointT>& cloud1, const pcl::PointCloud<PointT>& cloud2_inliner, const pcl::PointCloud<PointT>& cloud2_outliner ){
  // visualize
  boost::shared_ptr<pcl::visualization::PCLVisualizer>
  viewer_final (new pcl::visualization::PCLVisualizer ("3D Viewer"));
  viewer_final->setBackgroundColor (0, 0, 0);

  int v1;
  viewer_final->createViewPort(0, 0, 0.5, 1.0, v1);
```

```
  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ>
white(cloud1.makeShared(), 255, 255, 255);
  viewer_final->addPointCloud(cloud1.makeShared(), white, "raw" , v1);

  int v2;
  viewer_final->createViewPort(0.5, 0, 1.0, 1.0, v2);
  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> red(cloud2_inliner.makeSh
ared(), 255, 0, 0);
  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> green(cloud2_outliner.mak
eShared(), 0, 255, 0);
  viewer_final->addPointCloud(cloud2_inliner.makeShared(), red, "plane", v2);
  viewer_final->addPointCloud(cloud2_outliner.makeShared(), green, "no_plane", v2);

  while(!viewer_final->wasStopped()){
  viewer_final->spinOnce();
  }
 }

 int
 main (int argc, char** argv)
 {
  // All the objects needed
  pcl::PCDReader reader;
  pcl::PassThrough<PointT> pass;
  pcl::NormalEstimation<PointT, pcl::Normal> ne;
  pcl::SACSegmentationFromNormals<PointT, pcl::Normal> seg;
  pcl::PCDWriter writer;
  pcl::ExtractIndices<PointT> extract;
  pcl::ExtractIndices<pcl::Normal> extract_normals;
  pcl::search::KdTree<PointT>::Ptr tree (new pcl::search::KdTree<PointT> ());

  // Datasets
  pcl::PointCloud<PointT>::Ptr cloud (new pcl::PointCloud<PointT>);
  pcl::PointCloud<PointT>::Ptr cloud_filtered (new pcl::PointCloud<PointT>);
  pcl::PointCloud<pcl::Normal>::Ptr cloud_normals (new pcl::PointCloud<pcl::Normal>);
  pcl::PointCloud<PointT>::Ptr cloud_filtered2 (new pcl::PointCloud<PointT>);
  pcl::PointCloud<pcl::Normal>::Ptr cloud_normals2 (new pcl::PointCloud<pcl::Normal>);
  pcl::ModelCoefficients::Ptr coefficients_plane (new pcl::ModelCoefficients), coefficients
_cylinder (new pcl::ModelCoefficients);
  pcl::PointIndices::Ptr inliers_plane (new pcl::PointIndices), inliers_cylinder (new
pcl::PointIndices);

  // Read in the cloud data
  reader.read ("table_scene_mug_stereo_textured.pcd", *cloud);
  std::cerr << "PointCloud has: " << cloud->points.size () << " data points." << std::endl;

  // Build a passthrough filter to remove spurious NaNs
  pass.setInputCloud (cloud);
  pass.setFilterFieldName ("z");
  pass.setFilterLimits (0, 1.5);
```

```
  pass.filter (*cloud_filtered);
  std::cerr << "PointCloud after filtering has: " << cloud_filtered->points.size () << " da
ta points." << std::endl;

  // Estimate point normals
  ne.setSearchMethod (tree);
  ne.setInputCloud (cloud_filtered);
  ne.setKSearch (50);
  ne.compute (*cloud_normals);

  // Create the segmentation object for the planar model and set all the parameters
  seg.setOptimizeCoefficients (true);
  seg.setModelType (pcl::SACMODEL_NORMAL_PLANE);
  seg.setNormalDistanceWeight (0.1);
  seg.setMethodType (pcl::SAC_RANSAC);
  seg.setMaxIterations (100);
  seg.setDistanceThreshold (0.03);
  seg.setInputCloud (cloud_filtered);
  seg.setInputNormals (cloud_normals);
  // Obtain the plane inliers and coefficients
  seg.segment (*inliers_plane, *coefficients_plane);
  std::cerr << "Plane coefficients: " << *coefficients_plane << std::endl;

  // Extract the planar inliers from the input cloud
  extract.setInputCloud (cloud_filtered);
  extract.setIndices (inliers_plane);
  extract.setNegative (false);

  // Write the planar inliers to disk
  pcl::PointCloud<PointT>::Ptr cloud_plane (new pcl::PointCloud<PointT> ());
  extract.filter (*cloud_plane);
  std::cerr << "PointCloud representing the planar component: " << cloud_plane->points.size
() << " data points." << std::endl;
  writer.write ("table_scene_mug_stereo_textured_plane.pcd", *cloud_plane, false);

   // Remove the planar inliers, extract the rest
   extract.setNegative (true);
   extract.filter (*cloud_filtered2);
   extract_normals.setNegative (true);
   extract_normals.setInputCloud (cloud_normals);
   extract_normals.setIndices (inliers_plane);
   extract_normals.filter (*cloud_normals2);

   // Create the segmentation object for cylinder segmentation and set all the parameters
   seg.setOptimizeCoefficients (true);
   seg.setModelType (pcl::SACMODEL_CYLINDER);
   seg.setMethodType (pcl::SAC_RANSAC);
   seg.setNormalDistanceWeight (0.1);
   seg.setMaxIterations (10000);
```

```
  seg.setDistanceThreshold (0.05);
  seg.setRadiusLimits (0, 0.1);
  seg.setInputCloud (cloud_filtered2);
  seg.setInputNormals (cloud_normals2);

  // Obtain the cylinder inliers and coefficients
  seg.segment (*inliers_cylinder, *coefficients_cylinder);
  std::cerr << "Cylinder coefficients: " << *coefficients_cylinder << std::endl;

  // Write the cylinder inliers to disk
  extract.setInputCloud (cloud_filtered2);
  extract.setIndices (inliers_cylinder);
  extract.setNegative (false);
  pcl::PointCloud<PointT>::Ptr cloud_cylinder (new pcl::PointCloud<PointT> ());
  extract.filter (*cloud_cylinder);
  if (cloud_cylinder->points.empty ())
  std::cerr << "Can't find the cylindrical component." << std::endl;
  else
  {
  std::cerr << "PointCloud representing the cylindrical component: " << cloud_cylinder->points.size () << " data points." << std::endl;
  writer.write ("table_scene_mug_stereo_textured_cylinder.pcd", *cloud_cylinder, false);
  }

  //negative
  pcl::PointCloud<PointT> cloud_outliner;
  extract.setNegative(true);
  extract.filter(cloud_outliner);

  visualize_two_cloud(*cloud_filtered2, *cloud_cylinder, cloud_outliner);


  return (0);
 }

```

可视化



聚类提取

代码

```
#include <pcl/ModelCoefficients.h>
#include <pcl/point_types.h>
#include <pcl/io/pcd_io.h>
#include <pcl/filters/extract_indices.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/features/normal_3d.h>
#include <pcl/kdtree/kdtree.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
 #include <pcl/segmentation/sac_segmentation.h>
 #include <pcl/segmentation/extract_clusters.h>
 #include <pcl/visualization/pcl_visualizer.h>


 int
 main (int argc, char** argv)
 {
  // Read in the cloud data
  pcl::PCDReader reader;
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::PointXYZ>), cloud_f
(new pcl::PointCloud<pcl::PointXYZ>);
  reader.read ("table_scene_lms400.pcd", *cloud);
  std::cout << "PointCloud before filtering has: " << cloud->points.size () << " data point
s." << std::endl; //*

  // Create the filtering object: downsample the dataset using a leaf size of 1cm
  pcl::VoxelGrid<pcl::PointXYZ> vg;
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new pcl::PointCloud<pcl::PointXYZ>);
  vg.setInputCloud (cloud);
  vg.setLeafSize (0.01f, 0.01f, 0.01f);
  vg.filter (*cloud_filtered);
```

```
  std::cout << "PointCloud after filtering has: " << cloud_filtered->points.size () << " da
ta points." << std::endl; //*

  // Create the segmentation object for the planar model and set all the parameters
  pcl::SACSegmentation<pcl::PointXYZ> seg;
  pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
  pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_plane (new pcl::PointCloud<pcl::PointXYZ> ());
  pcl::PCDWriter writer;
  seg.setOptimizeCoefficients (true);
  seg.setModelType (pcl::SACMODEL_PLANE);
  seg.setMethodType (pcl::SAC_RANSAC);
  seg.setMaxIterations (100);
  seg.setDistanceThreshold (0.02);

  int i=0, nr_points = (int) cloud_filtered->points.size ();
  while (cloud_filtered->points.size () > 0.3 * nr_points)
  {
  // Segment the largest planar component from the remaining cloud
  seg.setInputCloud (cloud_filtered);
  seg.segment (*inliers, *coefficients);
  if (inliers->indices.size () == 0)
  {
  std::cout << "Could not estimate a planar model for the given dataset." << std::endl;
  break;
  }

  // Extract the planar inliers from the input cloud
  pcl::ExtractIndices<pcl::PointXYZ> extract;
  extract.setInputCloud (cloud_filtered);
  extract.setIndices (inliers);
  extract.setNegative (false);

  // Write the planar inliers to disk
  extract.filter (*cloud_plane);
  std::cout << "PointCloud representing the planar component: " << cloud_plane->points.size
() << " data points." << std::endl;

  // Remove the planar inliers, extract the rest
  extract.setNegative (true);
  extract.filter (*cloud_f);
  cloud_filtered = cloud_f;
  }

  // Creating the KdTree object for the search method of the extraction
  pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZ>);
  tree->setInputCloud (cloud_filtered);

  std::vector<pcl::PointIndices> cluster_indices;
```

```
77  pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
78  ec.setClusterTolerance (0.02); // 2cm
79  ec.setMinClusterSize (100);
80  ec.setMaxClusterSize (25000);
81  ec.setSearchMethod (tree);
82  ec.setInputCloud (cloud_filtered);
83  ec.extract (cluster_indices);
84
85  // visualize
86  boost::shared_ptr<pcl::visualization::PCLVisualizer>
87  viewer_final (new pcl::visualization::PCLVisualizer ("3D Viewer"));
88  viewer_final->setBackgroundColor (0, 0, 0);
89
90  int v1;
91  viewer_final->createViewPort(0, 0, 0.5, 1.0, v1);
92  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> white(cloud_filtered,
255, 255, 255);
93  viewer_final->addPointCloud(cloud_filtered, white, "cloud_filtered" , v1);
94
95  int v2;
96  viewer_final->createViewPort(0.5, 0, 1.0, 1.0, v2);
97  // pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> red(cloud_plane.makeSh
ared(), 255, 0, 0);
98  // pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> green(no_plane.makeSha
red(), 0, 255, 0);
99  // viewer_final->addPointCloud(cloud_plane.makeShared(), red, "plane", v2);
100  // viewer_final->addPointCloud(no_plane.makeShared(), green, "no_plane", v2);
101
102
103
104
105  int j = 0;
106  for (std::vector<pcl::PointIndices>::const_iterator it = cluster_indices.begin (); it !=
cluster_indices.end (); ++it)
107  {
108  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_cluster (new pcl::PointCloud<pcl::PointXYZ>);
109  for (std::vector<int>::const_iterator pit = it->indices.begin (); pit != it->indices.end
(); pit++)
110  cloud_cluster->points.push_back (cloud_filtered->points[*pit]); //*
111  cloud_cluster->width = cloud_cluster->points.size ();
112  cloud_cluster->height = 1;
113  cloud_cluster->is_dense = true;
114
115
116
117  std::cout << "PointCloud representing the Cluster: " << cloud_cluster->points.size () <<
" data points." << std::endl;
118  std::stringstream ss;
119  ss << "cloud_cluster_" << j << ".pcd";
120  writer.write<pcl::PointXYZ> (ss.str (), *cloud_cluster, false); //*
```

```
121   j++;
122
123
124   // visualize
125   int c1 = rand()%255;
126   int c2 = rand()%255;
127   int c3 = rand()%255;
128
129   pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> color(cloud_cluster, c1,
c2, c3);
130
131   viewer_final->addPointCloud(cloud_cluster, color, ss.str(), v2);
132
133   }
134
135
136   while(!viewer_final->wasStopped()){
137   viewer_final->spinOnce();
138   }
139
140   return (0);
141 }
142
```

可视化