目錄

简介	1.1
1.PCL的编译与安装	1.2
2.PCD(点云数据)文件格式	1.3
3.常见输入输出	1.4
4.可用Point类型(上)	1.5
5.可用Point类型(下)	1.6
6.利用KDTree近邻搜索	1.7
7.点云可视化	1.8
8.滤波	1.9
9.表面法线	1.10
10.关键点	1.11
11.特征描述	1.12
12.一个完整的实例	1.13
13.精配准	1.14
14.曲面重建	1.15
15.曲面分割	1.16
附录1：新建项目	1.17
附录2：常用函数(1)	1.18
附录2：常用函数(2)	1.19
附录2：常用函数(3)	1.20
附录3：Eigen函数表	1.21
附录4：新建QT界面项目	1.22
附录5：常用模型库	1.23

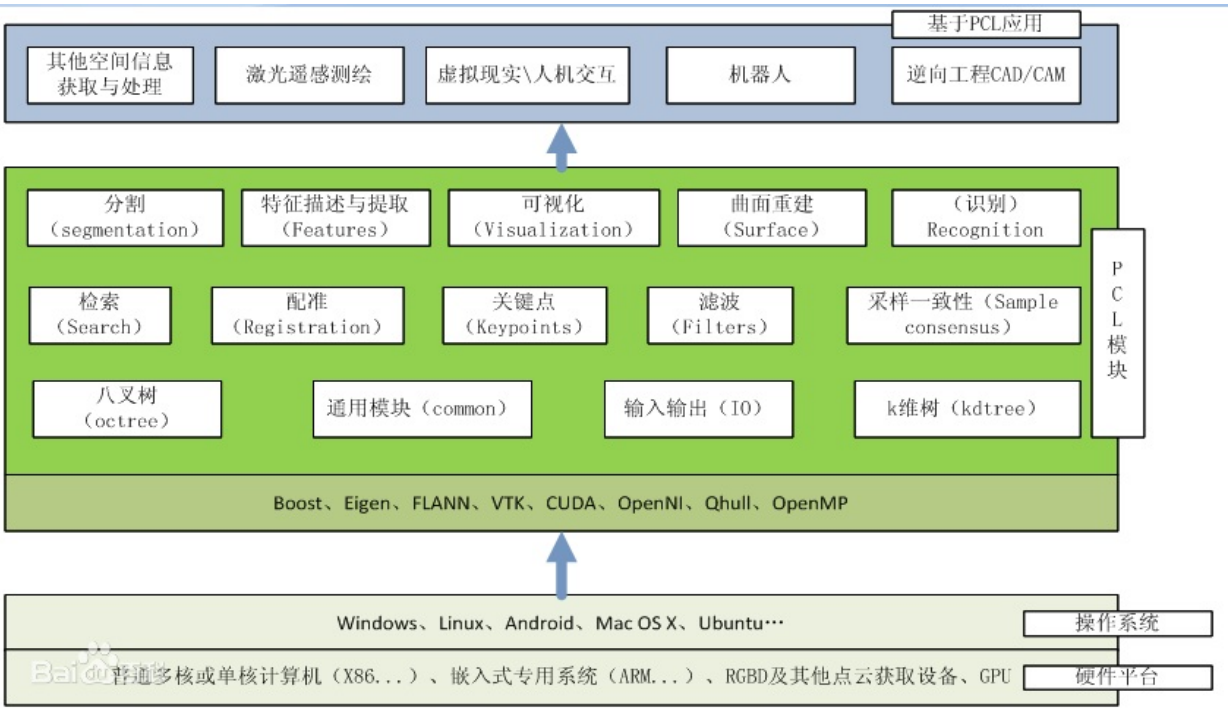
PCL Notes

学习**PCL**的一点个人总结。

> PCL（Point Cloud Library）是在吸收了前人点云相关研究基础上建立起来的大型跨平台开源C++编程库，它实现了大量点云相关的通用算法和高效数据结构，涉及到点云获取、滤波、分割、配准、检索、特征提取、识别、追踪、曲面重建、可视化等。支持多种操作系统平台，可在Windows、Linux、Android、Mac OS X、部分嵌入式实时系统上运行。如果说OpenCV是2D信息获取与处理的结晶，那么PCL就在3D信息获取与处理上具有同等地位，PCL是BSD授权方式，可以免费进行商业和学术应用。

*

> 如图PCL架构图所示，对于3D点云处理来说，PCL完全是一个的模块化的现代C++模板库。其基于以下第三方库：Boost、Eigen、FLANN、VTK、CUDA、OpenNI、Qhull，实现点云相关的获取、滤波、分割、配准、检索、特征提取、识别、追踪、曲面重建、可视化等。



- 相关网站

英文官网

中文论坛

英文论坛

- 参考书籍

  点云库PCL学习教程，朱德海，北京航空航天大学出版社

- 在线阅读地址

https://www.gitbook.com/book/mnewbie/pcl-notes/details

PCL的编译与安装

pcl需要第三方库的支持：**Boost**、**Eigen**、**FLANN**、**Qhull**、**VTK**、**OpenNI2**、**[QT**、**CUDA]**

- 工具
  VS
  cmake
- 编译**Boost**库

若需要mpi支持，首先下载安装mpi，然后到
boost_1_60_0\tools\build\src\tools\mpi.jam中修改下面几行：

249-251 line

```
local microsoft_mpi_sdk_path = "C:\\Program Files (x86)\\Microso
ft SDKs\\MPI" ;
local microsoft_mpi_path = "C:\\Program Files\\Microsoft MPI" ;
if [ GLOB $(microsoft_mpi_sdk_path)\\Include : mpi.h ]
```

260-262 line

```
options = <include>$(microsoft_mpi_sdk_path)/Include
<address-model>64:<library-path>$(microsoft_mpi_sdk_path)/Lib/x6
4
<library-path>$(microsoft_mpi_sdk_path)/Lib/x86
```

268 line

```
.mpirun = "\"$(microsoft_mpi_path)\\Bin\\mpiexec.exe"\" ;
```

然后以管理员身份运行VS自带的cmd(“VS2013 x86 本机工具命令提示”)，进入boost文件夹，运行bootstrap.bat,运行结束后会生成project-config.jam，打开并在第四行加上：using mpi ;（注意“ ;”前面有一个空格！） 接下来还是用cmd进入boost文件夹，运行如下命令编译boost：

```
Win32：
b2.exe toolset=msvc-12.0 address-model=32 --build-dir=build\x86
install --prefix="C:\Program Files (x86)\Boost" -j8
X64:
b2.exe toolset=msvc-12.0 address-model=64 --build-dir=build\x64
install --prefix="C:\Program Files\Boost" -j8
```

- 编译**Eigen**库

使用cmake，分别设置eigen的source和build路径（source路径是含有CMakeLists.txt的文件夹，其实就是source的根目录），如build不存在，点击Configure会提示新建build文件夹，选择vs编译器，这里注意Configure时有CMAKE_INSTALL_PREFIX这个选项，默认为C:\Program Files\Eigen．这里的路径即为该软件最后的安装路径(也是环境变量中要设置的EIGEN_ROOT的路径，可设置为你想要的其它路径，后边的FLANN，QHULL，VTK也是一样道理)．然后Generate。

以管理员身份运行VS（否则install时会失败），打开bulid文件夹下的eigen.sln工程，待加载完文件后，VS->生成->批生成->勾选ALL_BUILD的Debug和 Release完成生成，完成后同理生成INSTALL(Debug & Release)。可以看到eigen安装路径中出现include文件夹。最后在环境变量中建立EIGEN_ROOT变量，值为eigen的安装路径。

- 编译**qhull**库

使用cmake，分别设置qhull的source和build路径，选择vs编译器．注意根据需要修改CMAKE_INSTALL_PREFIX，然后添加一个entry：

> Name: CMAKE_DEBUG_POSTFIX
> Type: STRING
> Value: -d

修改完后再次点Configure，然后Generate。以管理员身份运行VS并打开qhull.sln工程文件，待加载完文件后，完成后生成ALL_BUILD（debug & release），然后生成INSTALL(Debug & Release)。完成后可以看到qhull安装路径中出现include和lib文件夹。最后在环境变量中建立QHULL_ROOT变量，值为qhull安装地址。

- 编译**flann**库

使用cmake，分别设置Flann的source和build路径，选择vs编译器．注意根据需要修改CMAKE_INSTALL_PREFIX，然后去掉BULID_MATLAB_BINDINGS和BULID_PYTHON_BINDINGS的勾选，不bulid它们。然后添加一个entry：

> Name: CMAKE_DEBUG_POSTFIX
> Type: STRING
> Value: -gd

修改完后再次点Configure，然后Generate。在C:\flann\src\cpp\flann\util(源码)中找到serialization.h文件 在92行BASIC_TYPE_SERIALIZER(bool)之后加入以下代码：

```
#ifdef _MSC_VER

BASIC_TYPE_SERIALIZER( unsigned __int64 );//注意此处__int64是两个下划线连一起

#endif
```

修改完后以管理员身份运行VS并打开flann.sln工程文件，待加载完文件后，（Debug & Release）生成all_build，完成后生成install(Debug & Release)。完成后可以看到flann安装路径下出现include和lib文件夹。最后在环境变量中建立FLANN_ROOT变量，值为flann安装路径。

- 编译**QT**

```
configure -platform win32-msvc2015 -confirm-license -opensource -debug-and-release -opengl desktop -prefix "" -nomake examples
```

将qt的bin目录添加到环境变量path中。

注：如果需要编译64位qt，只需要打开vs64位命令行即可，还需要一些其他第三方软件，百度即可。

- 编译**VTK**库

使用cmake，分别设置VTK的source和build路径，选择vs编译器．注意根据需要修改CMAKE_INSTALL_PREFIX选项，然后添加一个entry：

> Name: CMAKE_DEBUG_POSTFIX
> Type: STRING
> Value: -gd

修改完后再次点Configure，然后Generate。以管理员身份运行VS并打开VTK.sln工程文件，待加载完文件后，生成ALL_BUILD（debug & release），完成后生成install（debug & release）。完成后可以看到VTK文件夹中出现include和lib文件夹。最后在环境变量中建立VTK_ROOT，为VTK安装路径。

如果需要编译QT支持插件，在cmake时勾选VTK_Group_Qt，VTK_RENDERING_BACKEND可以是OpenGL也可以是OpenGL2，我使用OpenGL2。 Configure 后可能会出现报错，将VTK_QT_VERSION 修改成 5，QT_QMAKE_EXECUTABLE 为QTDIR\bin目录下的qmake.exe。点击 Configure 后仍然会报错，修改 Qt5_DIR 路径为 QTDIR\lib\cmake\Qt5，再次 Configure 后若出现红色部分为NOTFOUND根据名字自行添加。Configure 后再点击 Generate完成二进制生成。然后打开sln生成ALL_Build 和 Install。

> 添加了qt INSTALL会出错！原因是QVTKWidgetPlugin.dll也加了-gd，vs找不到，去掉即可。

生成完成后，在VTKBuildDIR\plugins\designer 中找到 QVTKWidgetPlugin.dll 插件，将其复制到 QtCreatorDIR\bin\plugins\designer目录中，用于在 Qt Creator的designer中显示 QVTKWidget 控件。

注：如果编译qt插件，目录中不能包含中文路径，否则报错。建议其他的也都不要中文路径。

- **OpenNI2**

下载安装即可。

- **CUDA**

如果需要编译Kinfu等内容，需要Nvida显卡、CUDA支持，CUDA下载安装即可。

- 编译**PCL**库

同样使用cmake，打开CMakeLists.txt，按照提示添加第三方的库文件，勾选自己需要编译的内容即可。
注意：

> 在win10上用vs2015编译PCL1.8的时候，编译到visualization模块时,pcl_visulalizer.cpp如下语句会报错。
>
> ```
> if (!pcl::visualization::getColormapLUT (static_cast<LookUpTableRepresentationProperties>(value), table))
>             break;
> 解决方案：
> 将所有的
> static_cast<LookUpTableRepresentationProperties>(value)
> 修改成
> static_cast<LookUpTableRepresentationProperties>(int(value))
> ```
>
> pcl1.8.0 用eigen3.3编译失败，ndt2d无法编译，换3.2可以
>
> （3）带nurbs编译(用于曲面拟合):
> （http://pointclouds.org/documentation/tutorials/bspline_fitting.php#bspline-fitting）
> Please note that the modules for NURBS and B-splines are not enabled by default. Make sure you enable “BUILD_surface_on_nurbs” in your ccmake configuration, by setting it to ON.
> If your license permits, also enable “USE_UMFPACK” for sparse linear solving. This requires SuiteSparse (libsuitesparse-dev in Ubuntu) which is faster, allows more degrees of freedom (i.e. control points) and more data points.
> windows下需要编译SuiteSparse库才能勾选“USE_UMFPACK”，比较麻烦，可以不用勾选。

- 添加环境变量

将以上所有编译的库中含有bin目录的添加到path中即可，OpenNI2添加Tools目录。

参考：

VS编译PCL1.8.0

VTK7.0&QTCreator5.6环境配置教程.pdf

- 现有**exe**安装

一路next即可，添加环境变量同上。

PCD（点云数据）文件格式

- **pcd**文件数据举例

```
# .PCD v.7 - Point Cloud Data file format
VERSION .7
FIELDS x y z rgb
SIZE 4 4 4 4
TYPE F FFF
COUNT 1 1 1 1
WIDTH 213
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS 213
DATA ascii
0.93773 0.33763 0 4.2108e+06
0.90805 0.35641 0 4.2108e+06
```

- 文件格式详解

PCD版本

在点云库（PCL）1.0版本发布之前，PCD文件格式有不同的修订号。这些修订号用PCD_Vx来编号（例如，PCD_V5、PCD_V6、PCD_V7等等），代表PCD文件的0.x版本号。然而PCL中PCD文件格式的正式发布是0.7版本（PCD_V7）。

文件头格式

每一个PCD文件包含一个文件头，它确定和声明文件中存储的点云数据的某种特性。PCD文件头必须用ASCII码来编码。PCD文件中指定的每一个文件头字段以及ascii点数据都用一个新行（\n）分开了，从0.7版本开始，PCD文件头包含下面的字段：

·VERSION –指定PCD文件版本

·FIELDS –指定一个点可以有的每一个维度和字段的名字。例如： FIELDS x y z # XYZ data FIELDS x y z rgb # XYZ + colors FIELDS x y z normal_xnormal_y normal_z # XYZ + surface normals FIELDS j1 j2 j3 # moment invariants ...

·SIZE –用字节数指定每一个维度的大小。例如： unsigned char/char has 1 byte unsigned short/short has 2 bytes unsignedint/int/float has 4 bytes double has 8 bytes

·TYPE –用一个字符指定每一个维度的类型。现在被接受的类型有： I –表示有符号类型int8（char）、int16（short）和int32（int）； U – 表示无符号类型 uint8（unsigned char）、uint16（unsigned short）和uint32（unsigned int）； F –表示浮点类型。

·COUNT –指定每一个维度包含的元素数目。例如，x这个数据通常有一个元素，但是像VFH这样的特征描述子就有308个。实际上这是在给每一点引入n维直方图描述符的方法，把它们当做单个的连续存储块。默认情况下，如果没有COUNT，所有维度的数目被设置成1。

·WIDTH –用点的数量表示点云数据集的宽度。根据是有序点云还是无序点云，WIDTH有两层解释： 1)它能确定无序数据集的点云中点的个数（和下面的POINTS一样）； 2)它能确定有序点云数据集的宽度（一行中点的数目）。 注意：有序点云数据集，意味着点云是类似于图像（或者矩阵）的结构，数据分为行和列。这种点云的实例包括立体摄像机和时间飞行摄像机生成的数据。有序数据集的优势在于，预先了解相邻点（和像素点类似）的关系，邻域操作更加高效，这样就加速了计算并降低了PCL中某些算法的成本。 例如： WIDTH 640 # 每行有640个点

·HEIGHT –用点的数目表示点云数据集的高度。类似于WIDTH ，HEIGHT也有两层解释： 1)它表示有序点云数据集的高度（行的总数）； 2)对于无序数据集它被设置成1（被用来检查一个数据集是有序还是无序）。 有序点云例子： WIDTH 640 # 像图像一样的有序结构，有640行和480列， HEIGHT 480 # 这样该数据集中共有640*480=307200个点 无序点云例子： WIDTH 307200 HEIGHT 1 # 有307200个点的无序点云数据集

·VIEWPOINT–指定数据集中点云的获取视点。VIEWPOINT有可能在不同坐标系之间转换的时候应用，在辅助获取其他特征时也比较有用，例如曲面法线，在判断方向一致性时，需要知道视点的方位， 视点信息被指定为平移（txtytz）+四元数（qwqxqyqz）。默认值是： VIEWPOINT 0 0 0 1 0 0 0

·POINTS–指定点云中点的总数。从0.7版本开始，该字段就有点多余了，因此有可能在将来的版本中将它移除。 例子： POINTS 307200 #点云中点的总数为307200

·DATA –指定存储点云数据的数据类型。从0.7版本开始，支持两种数据类型：ascii和二进制。查看下一节可以获得更多细节。 注意：文件头最后一行（DATA）的下一个字节就被看成是点云的数据部分了，它会被解释为点云数据。 警告：PCD文件的文件头部分必须以上面的顺序精确指定，也就是如下顺序： VERSION、FIELDS、SIZE、TYPE、COUNT、WIDTH、HEIGHT、VIEWPOINT、POINTS、DATA 之间用换行隔开。

- 数据存储类型

在0.7版本中，.PCD文件格式用两种模式存储数据： 如果以ASCII形式，每一点占据一个新行： p_1 p_2 ... p_n 注意：从PCL 1.0.1版本开始，用字符串“nan”表示NaN，此字符表示该点的值不存在或非法等。

- 其他文件格式

PLY是一种多边形文件格式，由Stanford大学的Turk等人设计开发；

STL是3D Systems公司创建的模型文件格式，主要应用于CAD、CAM领域；

OBJ是从几何学上定义的文件格式，首先由Wavefront Technologies开发；

X3D是符合ISO标准的基于XML的文件格式，表示3D计算机图形数据；

其他许多种格式。

数据输入输出

- 从**PCD**文件中读取与保存点云文件

```
#include <iostream> //标准c++库输入输出相关头文件
#include <pcl/io/pcd_io.h> // pcd读写相关头文件
#include <pcl/point_types.h> // pcl中支持的点类型头文件

// 定义点云格式，具体见下章
typedef pcl::PointXYZ PointT;

int main(int argc, char** argv)
{
    // 定义点云
    pcl::PointCloud<PointT>::Ptr cloud(new pcl::PointCloud<Point
T>);

    // 读取点云，失败返回-1
    if (pcl::io::loadPCDFile<PointT>("readName.pcd", *cloud) ==
-1)
    {
        PCL_ERROR("couldn't read file\n");
        return (-1);
    }

    // 输出点云大小 cloud->width * cloud->height
    std::cout << "点云大小：" << cloud->size() << std::endl;

    // 保存点云文件
    pcl::io::savePCDFile("saveName.pcd", *cloud);

    system("pause"); // windows命令行窗口暂停
    return (0);
}
```

- 读取与保存**PLY**文件

后缀命名为.ply格式文件，常用的点云数据文件。ply文件不仅可以存储点数据，而且可以存储网格数据. 用编辑器打开一个ply文件，观察表头，如果表头element face的值为0,则表示该文件为点云文件，如果element face的值为某一正整数N，则表示该文件为网格文件，且包含N个网格.所以利用pcl读取 ply 文件，不能一味用 `pcl::PointCloud<PointT>::Ptr cloud (new pcl::PointCloud<PintT>)` 来读取。在读取ply文件时候，首先要分清该文件是点云还是网格类文件。如果是点云文件，则按照一般的点云类去读取即可，如果ply文件是网格类，则需要:

```
#include <pcl/io/ply_io.h>

pcl::PolygonMesh mesh;
pcl::io::loadPLYFile("readName.ply", mesh);
pcl::io::savePLYFile("saveName.ply", mesh);
```

可用Point类型(上)

为了涵盖能想到的所有可能的情况，PCL中定义了大量的point类型。下面是一小段，在point_types.hpp中有完整目录，这个列表很重要，因为在定义你自己的类型之前，需要了解已有的类型，如果你需要的类型，已经存在于PCL，那么就不需要重复定义了。

- 目录
  - PointXYZ
  - PointXYZI
  - PointXYZRGBA
  - PointXYZRGB
  - PointXY
  - InterestPoint
  - Normal
  - PointNormal
  - PointXYZRGBNormal
  - PointXYZINormal

- **PointXYZ–**成员变量: **float x, y, z;**

PointXYZ是使用最常见的一个点数据类型，因为它只包含三维xyz坐标信息，这三个浮点数附加一个浮点数来满足存储对齐，用户可利用points[i].data[0]，或者points[i].x访问点的x坐标值。

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
```

- **PointXYZI–成员变量: float x, y, z, intensity;**

PointXYZI是一个简单的XYZ坐标加intensity的point类型，理想情况下，这四个变量将新建单独一个结构体，并且满足存储对齐，然而，由于point的大部分操作会把data[4]元素设置成0或1（用于变换），不能让intensity与xyz在同一个结构体中，如果这样的话其内容将会被覆盖。例如，两个点的点积会把他们的第四个元素设置成0，否则该点积没有意义，等等。因此，对于兼容存储对齐，用三个额外的浮点数来填补intensity，这样在存储方面效率较低，但是符合存储对齐要求，运行效率较高。

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    struct
    {
        float intensity;
    };
    float data_c[4];
};
```

- **PointXYZRGBA–成员变量: float x, y, z; uint32_t rgba;**

除了rgba信息被包含在一个整型变量中，其它的和PointXYZI类似。

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    struct
    {
        uint32_t rgba;
    };
    float data_c[4];
};
```

- **PointXYZRGB - float x, y, z, rgb;**

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    struct
    {
        float rgb;
    };
    float data_c[4];
};
```

- **PointXY-float x, y;**

简单的二维x-y point结构

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

- **InterestPoint-float x, y, z, strength;**

除了strength表示关键点的强度的测量值，其它的和PointXYZI类似。

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    struct
    {
        float strength;
    };
    float data_c[4];
};
```

- **Normal - float normal[3], curvature;**

另一个最常用的数据类型，Normal结构体表示给定点所在样本曲面上的法线方向，以及对应曲率的测量值（通过曲面块特征值之间关系获得——查看NormalEstimation类API以便获得更多信息，后续章节有介绍），由于在PCL中对曲面法线的操作很普遍，还是用第四个元素来占位，这样就兼容SSE和高效计算，例如，用户访问法向量的第一个坐标，可以通过points[i].data_n[0]或者points[i].normal[0]或者points[i].normal_x，再一次强调，曲率不能被存储在同一个结构体中，因为它会被普通的数据操作覆盖掉。

```
union
{
    float data_n[4];
    float normal[3];
    struct
    {
        float normal_x;
        float normal_y;
        float normal_z;
    };
}
union
{
    struct
    {
        float curvature;
    };
    float data_c[4];
};
```

- **PointNormal - float x, y, z; float normal[3], curvature;**

PointNormal是存储XYZ数据的point结构体，并且包括采样点对应法线和曲率。

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    float data_n[4];
    float normal[3];
    struct
    {
        float normal_x;
        float normal_y;
        float normal_z;
    };
};
union
{
    struct
    {
        float curvature;
    };
    float data_c[4];
};
```

- **PointXYZRGBNormal - float x, y, z, rgb, normal[3], curvature;**

PointXYZRGBNormal存储XYZ数据和RGB颜色的point结构体，并且包括曲面法线和曲率

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    float data_n[4];
    float normal[3];
    struct
    {
        float normal_x;
        float normal_y;
        float normal_z;
    };
}
union
{
    struct
    {
        float rgb;
        float curvature;
    };
    float data_c[4];
};
```

- **PointXYZINormal - float x, y, z, intensity, normal[3], curvature;**

PointXYZINormal存储XYZ数据和强度值的point结构体，并且包括曲面法线和曲率。

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    float data_n[4];
    float normal[3];
    struct
    {
        float normal_x;
        float normal_y;
        float normal_z;
    };
}
union
{
    struct
    {
        float intensity;
        float curvature;
    };
    float data_c[4];
};
```

可用Point类型(下)

- 目录

  - PointWithRange
  - PointWithViewpoint
  - MomentInvariants
  - PrincipalRadiiRSD
  - Boundary
  - PrincipalCurvatures
  - BounPFHSignature125dary
  - FPFHSignature33
  - VFHSignature308
  - Narf36
  - BorderDescription
  - IntensityGradient
  - Histogram
  - PointWithScale
  - PointSurfel

- **PointWithRange - float x, y, z (union with float point[4]), range;**

PointWithRange除了range包含从所获得的视点到采样点的距离测量值之外，其它与PointXYZI类似。

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    struct
    {
        float range;
    };
    float data_c[4];
};
```

- **PointWithViewpoint - float x, y, z, vp_x, vp_y, vp_z;**

ointWithViewpoint除了vp_x、vp_y和vp_z以三维点表示所获得的视点之外，其它与PointXYZI一样。

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    struct
    {
        float vp_x;
        float vp_y;
        float vp_z;
    };
    float data_c[4];
};
```

- **MomentInvariants - float j1, j2, j3;**

MomentInvariants是一个包含采样曲面上面片的三个不变矩的point类型，描述面片上质量的分布情况。查看MomentInvariantsEstimation以获得更多信息。

```
struct
{
    float j1,j2,j3;
};
```

- **PrincipalRadiiRSD - float r_min, r_max;**

PrincipalRadiiRSD是一个包含曲面块上两个RSD半径的point类型，查看RSDEstimation以获得更多信息。

```
struct
{
    float r_min,r_max;
};
```

- **Boundary - uint8_t boundary_point;**

Boundary存储一个点是否位于曲面边界上的简单point类型，查看BoundaryEstimation以获得更多信息。

```
struct
{
    uint8_t boundary_point;
};
```

- **PrincipalCurvatures - float principal_curvature[3], pc1, pc2;**

PrincipalCurvatures包含给定点主曲率的简单point类型。查看PrincipalCurvaturesEstimation以获得更多信息。

```
struct
{
    union
    {
        float principal_curvature[3];
        struct
        {
            float principal_curvature_x;
            float principal_curvature_y;
            float principal_curvature_z;
        };
    };
    float pc1;
    float pc2;
};
```

- **PFHSignature125 - float pfh[125];**

PFHSignature125包含给定点的PFH（点特征直方图）的简单point类型,查看PFHEstimation以获得更多信息。

```
struct
{
    float histogram[125];
};
```

- **FPFHSignature33 - float fpfh[33];**

FPFHSignature33包含给定点的FPFH（快速点特征直方图）的简单point类型，查看FPFHEstimation以获得更多信息。

```
struct
{
    float histogram[33];
};
```

- **VFHSignature308 - float vfh[308];**

VFHSignature308包含给定点VFH（视点特征直方图）的简单point类型，查看VFHEstimation以获得更多信息。

```
struct
{
    float histogram[308];
};
```

- **Narf36 - float x, y, z, roll, pitch, yaw; float descriptor[36];**

Narf36包含给定点NARF（归一化对齐半径特征）的简单point类型，查看NARFEstimation以获得更多信息。

```
struct
{
    float x,y,z,roll,pitch,yaw;
    float descriptor[36];
};
```

- **BorderDescription - int x, y; BorderTraits traits;**

BorderDescription包含给定点边界类型的简单point类型，看BorderEstimation以获得更多信息。

```
struct
{
    int x,y;
    BorderTraitstraits;
};
```

- **IntensityGradient - float gradient[3];**

IntensityGradient包含给定点强度的梯度point类型，查看IntensityGradientEstimation以获得更多信息。

```
struct
{
    union
    {
        float gradient[3];
        struct
        {
            float gradient_x;
            float gradient_y;
            float gradient_z;
        };
    };
};
```

- **Histogram - float histogram[N];**

Histogram用来存储一般用途的n维直方图。

```
template<int N>
struct Histogram
{
    float histogram[N];
};
```

- **PointWithScale - float x, y, z, scale;**

PointWithScale除了scale表示某点用于几何操作的尺度（例如，计算最近邻所用的球体半径，窗口尺寸等等），其它的和PointXYZI一样。

```
struct
{
    union
    {
        float data[4];
        struct
        {
            float x;
            float y;
            float z;
        };
    };
    float scale;
};
```

- **PointSurfel - float x, y, z, normal[3], rgba, radius, confidence, curvature;**

PointSurfel存储XYZ坐标、曲面法线、RGB信息、半径、可信度和曲面曲率的复杂point类型。

```
union
{
    float data[4];
    struct
    {
        float x;
        float y;
        float z;
    };
};
union
{
    float data_n[4];
    float normal[3];
    struct
    {
        float normal_x;
        float normal_y;
        float normal_z;
    };
};
union
{
    struct
    {
        uint32_trgba;
        float radius;
        float confidence;
        float curvature;
    };
    float data_c[4];
};
```

以上内容来自http://www.pclcn.org/study/news.php?lang=cn&class3=105

利用KDTree近邻搜索

k-d树（k-dimensional树的简称），是一种分割k维数据空间的数据结构。主要应用于多维空间关键数据的搜索（如：范围搜索和最近邻搜索）。K-D树是二进制空间分割树的特殊的情况。

索引结构中相似性查询有两种基本的方式：一种是范围查询（range searches），另一种是K近邻查询（K-neighbor searches）。范围查询就是给定查询点和查询距离的阈值，从数据集中找出所有与查询点距离小于阈值的数据；K近邻查询是给定查询点及正整数K，从数据集中找到距离查询点最近的K个数据，当K=1时，就是最近邻查询（nearest neighbor searches）。

PCL中类 `pcl::KdTree<PointT>` 是kd-tree数据结构的实现。并且提供基于FLANN进行快速搜索的一些相关子类与包装类。具体可以参考相应的API。下面给出2个类的具体用法。

- **pcl::search::KdTree < PointT >**

`pcl::search::KdTree<PointT>` 是 `pcl::search::Search< PointT >` 的子类，是 `pcl::KdTree<PointT>` 的包装类。包含(1) k 近邻搜索；(2) 邻域半径搜索。

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

#include <pcl/search/kdtree.h> // 包含kdtree头文件

typedef pcl::PointXYZ PointT;

int main()
{
    pcl::PointCloud<PointT>::Ptr cloud(new pcl::PointCloud<Point
T>);
    pcl::io::loadPCDFile("read.pcd", *cloud);

    // 定义KDTree对象
    pcl::search::KdTree<PointT>::Ptr kdtree(new pcl::search::KdT
ree<PointT>);
```

```
    kdtree->setInputCloud(cloud); // 设置要搜索的点云，建立KDTree

    std::vector<int> indices; // 存储查询近邻点索引
    std::vector<float> distances; // 存储近邻点对应距离的平方

    PointT point = cloud->points[0]; // 初始化一个查询点

    // 查询距point最近的k个点
    int k = 10;
    int size = kdtree->nearestKSearch(point, k, indices, distanc
es);

    std::cout << "search point : " << size << std::endl;

    // 查询point半径为radius邻域球内的点
    double radius = 2.0;
    size = kdtree->radiusSearch(point, radius, indices, distance
s);

    std::cout << "search point : " << size << std::endl;

    system("pause");
    return 0;
}
```

注意： 搜索结果默认是按照距离point点的距离从近到远排序；如果InputCloud中含有point点，搜索结果的的第一个点是point本身。

- **pcl::KdTreeFLANN < PointT >**

`pcl::KdTreeFLANN<PointT>` 是 `pcl::KdTree<PointT>` 的子类，可以实现同样的功能。

```
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
// 包含相关头文件
#include <pcl/kdtree/kdtree_flann.h>

typedef pcl::PointXYZ PointT;

int main()
{
    pcl::PointCloud<PointT>::Ptr cloud(new pcl::PointCloud<Point
T>);
    pcl::io::loadPCDFile("read.pcd", *cloud);

    pcl::KdTreeFLANN<pcl::PointXYZ> kdtree; //创建KDtree
    kdtree.setInputCloud(cloud); // 设置要搜索的点云，建立KDTree

    std::vector<int> indices; // 存储查询近邻点索引
    std::vector<float> distances; // 存储近邻点对应距离的平方

    PointT point = cloud->points[0]; // 初始化一个查询点

    // 查询距point最近的k个点
    int k = 10;
    int size = kdtree.nearestKSearch(point, k, indices, distance
s);

    std::cout << "search point : " << size << std::endl;

    // 查询point半径为radius邻域球内的点
    double radius = 2.0;
    size = kdtree.radiusSearch(point, radius, indices, distances
);

    std::cout << "search point : " << size << std::endl;

    system("pause");
    return 0;
}
```

点云可视化

可视化（Visualization）是利用计算机图形学和图像处理技术，将数据转换成图形或图像在屏幕显示出来，并且进行交互处理的理论、方法和技术。

PCL中pcl_visualization库中提供了可视化相关的数据结构和组件，其主要是为了可视化其他模块算法处理后的结果，可直观的反馈给用户。其依赖于pcl_common、pcl_range_image、pcl_kdtree、pcl_IO模块以及VTK外部开源可视化库。下面给出2个常用的可视化类。

- **pcl::visualization::PCLVisualizer**

PCLVisualizer是PCL可视化3D点云的主要类。其内部实现了添加各种3D对象以及交互的实现等，比其他类实现的功能更齐全。

基础显示功能： 显示点云、网格、设置颜色、连线

```
#include <pcl/io/pcd_io.h>
#include <pcl/io/ply_io.h>
#include <pcl/point_types.h>
// 包含相关头文件
#include <pcl/visualization/pcl_visualizer.h>

typedef pcl::PointXYZ PointT;

int main()
{
    // 读取点云
    pcl::PointCloud<PointT>::Ptr cloud1(new pcl::PointCloud<Poin
tT>);
    pcl::io::loadPCDFile("read1.pcd", *cloud1);

    pcl::PointCloud<PointT>::Ptr cloud2(new pcl::PointCloud<Poin
tT>);
    pcl::io::loadPCDFile("read2.pcd", *cloud2);

    // 定义对象
    pcl::visualization::PCLVisualizer viewer;
```

```
    //设置背景颜色，默认黑色
    viewer.setBackgroundColor(100, 100, 100); // rgb

    // --- 显示点云数据 ----
    // "cloud1" 为显示id，默认cloud,显示多个点云时用默认会报警告。
    viewer.addPointCloud(cloud1, "cloud1");

    pcl::visualization::PointCloudColorHandlerCustom<PointT> red
(cloud2, 255, 0, 0); // rgb
    // 将点云设置颜色，默认白色
    viewer.addPointCloud(cloud2, red, "cloud2");

    // 将两个点连线
    PointT temp1 = cloud1->points[0];
    PointT temp2 = cloud1->points[1];
    viewer.addLine(temp1, temp2, "line0");
    // 同样可以设置线的颜色，
    //viewer.addLine(temp1, temp2, 255，0，0， "line0");

    // --- 显示网格数据 ---
    pcl::PolygonMesh mesh;
    pcl::io::loadPLYFile("read.ply", mesh);

    viewer.addPolygonMesh(mesh);


    // 开始显示2种方法,任选其一
    // 1. 阻塞式
    viewer.spin();

    // 2. 非阻塞式
    while (!viewer.wasStopped())
    {
        viewer.spinOnce(100);
        boost::this_thread::sleep(boost::posix_time::microsecond
s(100000));
        // 可添加其他操作
    }

    system("pause");
```

```
    return 0;
}
```

高级功能：设置回掉函数进行交互、显示区域分割

```
+ 按键事件
```

```
#include <pcl/io/pcd_io.h>
#include <pcl/visualization/pcl_visualizer.h>

// 回掉函数所用数据结构
struct callback_args {
    bool *isShow;
    pcl::PointCloud<pcl::PointXYZ>::Ptr orgin_points;
    pcl::visualization::PCLVisualizer::Ptr viewerPtr;
};

// 按键事件回掉函数
void kb_callback(const pcl::visualization::KeyboardEvent& event,
 void* args)
{
    if (event.keyDown() && event.getKeyCode() == 'a')
    {
        std::cout << "a has pressed" << std::endl;
        struct callback_args* data = (struct callback_args *)arg
s;
        if (*(data->isShow))
        {
            data->viewerPtr->removePointCloud("cloud");
            *(data->isShow) = false;
            std::cout << "remove" << std::endl;
        }
        else {
            data->viewerPtr->addPointCloud(data->orgin_points, "
cloud");
            *(data->isShow) = true;
            std::cout << "add" << std::endl;
        }
```

```
    }
}

int main(int argc, char** argv)
{
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointClou
d<pcl::PointXYZ>);
    pcl::io::loadPCDFile("bunny.pcd", *cloud);
    pcl::console::print_highlight("load cloud !\n");

    // 定义对象
    pcl::visualization::PCLVisualizer::Ptr viewer(new pcl::visua
lization::PCLVisualizer);
    viewer->addPointCloud(cloud, "cloud");

    // 初始化参数
    bool isShow = true;
    struct callback_args kb_args;
    kb_args.isShow = &isShow;
    kb_args.orgin_points = cloud;
    kb_args.viewerPtr = viewer;

    // 设置回掉函数
    viewer->registerKeyboardCallback(kb_callback, (void*)&kb_arg
s);

    viewer->spin();

    return 0;
}
```

```
+ 点选取事件
```

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

struct callback_args {
```

```
    // structure used to pass arguments to the callback function
    pcl::PointCloud<pcl::PointXYZ>::Ptr clicked_points_3d;
    pcl::visualization::PCLVisualizer::Ptr viewerPtr;
};

void pp_callback(const pcl::visualization::PointPickingEvent& ev
ent, void* args)
{
    struct callback_args* data = (struct callback_args *)args;

    if (event.getPointIndex() == -1)
        return;
    int index = event.getPointIndex();
    std::cout << "index: " << index << std::endl;
    pcl::PointXYZ current_point;
    event.getPoint(current_point.x, current_point.y, current_poi
nt.z);
    data->clicked_points_3d->points.push_back(current_point);
    // Draw clicked points in red:
    pcl::visualization::PointCloudColorHandlerCustom<pcl::PointX
YZ> red(data->clicked_points_3d, 255, 0, 0);
    data->viewerPtr->removePointCloud("clicked_points");
    data->viewerPtr->addPointCloud(data->clicked_points_3d, red,
 "clicked_points");
    data->viewerPtr->setPointCloudRenderingProperties(pcl::visua
lization::PCL_VISUALIZER_POINT_SIZE, 10, "clicked_points");
    std::cout << current_point.x << " " << current_point.y << "
" << current_point.z << std::endl;
}

int main(int argc, char** argv)
{
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointClou
d<pcl::PointXYZ>);
    pcl::io::loadPCDFile("bunny.pcd", *cloud);
    pcl::console::print_highlight("load cloud !\n");

    pcl::visualization::PCLVisualizer viewer;

    pcl::visualization::PointCloudColorHandlerCustom<pcl::PointX
```

```
YZ> green(cloud, 0, 255, 0);
    viewer.addPointCloud(cloud, green, "cloud");

    // Add point picking callback to viewer:
    struct callback_args cb_args;
    pcl::PointCloud<pcl::PointXYZ>::Ptr clicked_points_3d(new pc
l::PointCloud<pcl::PointXYZ>);
    cb_args.clicked_points_3d = clicked_points_3d;
    cb_args.viewerPtr = pcl::visualization::PCLVisualizer::Ptr(&
viewer);
    viewer.registerPointPickingCallback(pp_callback, (void*)&cb_
args);
    pcl::console::print_highlight("Shift+click on three floor po
ints, then press 'Q'...\n");

    // Spin until 'Q' is pressed:
    viewer.spin();

    system("pause");
    return 0;
}
```

```
+ 区域选取事件
```

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

struct callback_args{
    // structure used to pass arguments to the callback function
    pcl::PointCloud<pcl::PointXYZ>::Ptr orgin_points;
    pcl::PointCloud<pcl::PointXYZ>::Ptr chosed_points_3d;
    pcl::visualization::PCLVisualizer::Ptr viewerPtr;
};

void ap_callback(const pcl::visualization::AreaPickingEvent& eve
nt, void* args)
{
```

```
    struct callback_args* data = (struct callback_args *)args;
    std::vector<int> indiecs;

    if (!event.getPointsIndices(indiecs))
        return;
    for (int i = 0; i < indiecs.size(); ++i)
    {
        data->chosed_points_3d->push_back(data->orgin_points->po
ints[indiecs[i]]);
    }

    // Draw clicked points in red:
    pcl::visualization::PointCloudColorHandlerCustom<pcl::PointX
YZ> red(data->chosed_points_3d, 255, 0, 0);
    data->viewerPtr->removePointCloud("chosed_points");
    data->viewerPtr->addPointCloud(data->chosed_points_3d, red,
"chosed_points");
    data->viewerPtr->setPointCloudRenderingProperties(pcl::visua
lization::PCL_VISUALIZER_POINT_SIZE, 5, "chosed_points");
    std::cout << "selected " << indiecs.size() << " points , now
 sum is " << data->chosed_points_3d->size() << std::endl;
}

int main(int argc, char** argv)
{
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointClou
d<pcl::PointXYZ>);
    pcl::io::loadPCDFile("rabbit.pcd", *cloud);
    pcl::console::print_highlight("load cloud !\n");

    pcl::visualization::PCLVisualizer viewer;

    pcl::visualization::PointCloudColorHandlerCustom<pcl::PointX
YZ> green(cloud, 0, 255, 0);
    viewer.addPointCloud(cloud, green, "cloud");

    // Add point picking callback to viewer:
    struct callback_args cb_args;
    cb_args.orgin_points = cloud;
    pcl::PointCloud<pcl::PointXYZ>::Ptr chosed_points_3d(new pcl
```

```
::PointCloud<pcl::PointXYZ>);
    cb_args.chosed_points_3d = chosed_points_3d;
    cb_args.viewerPtr = pcl::visualization::PCLVisualizer::Ptr(&
viewer);
    viewer.registerAreaPickingCallback(ap_callback, (void*)&cb_a
rgs);
    pcl::console::print_highlight("press x enter slected model,
then press 'qQ'...\n");

    // Spin until 'Q' is pressed:
    viewer.spin();

    system("pause");
    return 0;
}
```

```
+ 显示区域分割
```

pcl可以将显示区域分割，从(xmin,ymin)到(xmax,ymax)一个矩形区域，范围是（0，1）。左下角(0,0)，右上角(1,1)。之前所有的函数都支持区域显示。

```
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
// 包含相关头文件
#include <pcl/visualization/pcl_visualizer.h>

typedef pcl::PointXYZ PointT;

int main()
{
    // 读取点云
    pcl::PointCloud<PointT>::Ptr cloud(new pcl::PointCloud<Point
T>);
    pcl::io::loadPCDFile("read.pcd", *cloud);

    // 定义对象
    pcl::visualization::PCLVisualizer viewer;

    int v1(1); // viewport
    viewer.createViewPort(0.0, 0.0, 0.5, 1.0, v1);
    viewer.setBackgroundColor(255, 0, 0, v1);
    viewer.addPointCloud(cloud, "cloud1", v1);;

    int v2(2);// viewport
    viewer.createViewPort(0.5, 0.0, 1.0, 1.0, v1);
    viewer.setBackgroundColor(0, 255, 0, v2);
    viewer.addPointCloud(cloud, "cloud2", v2);;

    viewer.spin();

    system("pause");
    return 0;
}
```

- **pcl::visualization::CloudViewer**

```
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
// 包含相关头文件
#include <pcl/visualization/cloud_viewer.h>

typedef pcl::PointXYZ PointT;

int main()
{
    // 读取点云
    pcl::PointCloud<PointT>::Ptr cloud(new pcl::PointCloud<Point
T>);
    pcl::io::loadPCDFile("read.pcd", *cloud);

    pcl::visualization::CloudViewer viewer("simple cloud viewer"
);
    viewer.showCloud(cloud);
    while (!viewer.wasStopped())
    {
        // todo::
    }

    system("pause");
    return 0;
}
```

滤波

PCL中总结了集中需要进行点云滤波处理的情况，分别如下： (1) 点云数据密度不规则需要平滑。 (2) 因为遮挡等问题造成离群点需要去除。 (3) 大量数据需要进行下采样。 (4) 噪声数据需要去除。

PCL点云格式分为有序点云和无序点云，针对有序点云提供了双边滤波、高斯滤波、中值滤波等，针对无序点云提供了体素栅格、随机采样等。

下边给出两个下采样类的应用实例。（VoxelGrid、UniformSampling）

- **VoxelGrid、UniformSampling**

```
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
// 包含相关头文件
#include <pcl/filters/voxel_grid.h>
#include <pcl/filters/uniform_sampling.h>

typedef pcl::PointXYZ PointT;

int main(int argc, char** argv)
{
    // 读取点云
    pcl::PointCloud<PointT>::Ptr cloud(new pcl::PointCloud<Point
T>);
    pcl::io::loadPCDFile(argv[1], *cloud);
    std::cout << "original cloud size : " << cloud->size() << st
d::endl;

    // 使用体素化网格(VoxelGrid)进行下采样
    pcl::VoxelGrid<PointT> grid; //创建滤波对象
    const float leaf = 0.005f;
    grid.setLeafSize(leaf, leaf, leaf); // 设置体素体积
    grid.setInputCloud(cloud); // 设置点云
    pcl::PointCloud<PointT>::Ptr voxelResult(new pcl::PointCloud
<PointT>);
    grid.filter(*voxelResult); // 执行滤波，输出结果
```

```
    std::cout << "voxel downsample size :" << voxelResult->size(
) << std::endl;

    // 使用UniformSampling进行下采样
    pcl::UniformSampling<PointT> uniform_sampling;
    uniform_sampling.setInputCloud(cloud);
    double radius = 0.005f;
    uniform_sampling.setRadiusSearch(radius);
    pcl::PointCloud<PointT>::Ptr uniformResult(new pcl::PointClo
ud<PointT>);
    uniform_sampling.filter(*uniformResult);
    std::cout << "UniformSampling size :" << uniformResult->size
() << std::endl;

    system("pause");
    return 0;
}
```

双边滤波PCL提供了两种方法，FastBilateralFilter 和 BilateralFilter。
FastBilateralFilter需要有序点云数据，BilateralFilter需要带强度的点云数据。

- 双边滤波

```
#include <iostream>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/filters/impl/bilateral.hpp>
#include <pcl/visualization/pcl_visualizer.h>

void bilateralFilter(pcl::PointCloud<pcl::PointXYZI>::Ptr &input
, pcl::PointCloud<pcl::PointXYZI>::Ptr& output)
{
    pcl::search::KdTree<pcl::PointXYZI>::Ptr tree1(new pcl::sear
ch::KdTree<pcl::PointXYZI>);
    // Apply the filter
    pcl::BilateralFilter<pcl::PointXYZI> fbf;
    fbf.setInputCloud(input);
    fbf.setSearchMethod(tree1);
    fbf.setStdDev(0.1);
    fbf.setHalfSize(0.1);
```

```
    fbf.filter(*output);
}
int main(int argc, char** argv)
{
    pcl::PointCloud<pcl::PointXYZI>::Ptr cloud(new pcl::PointClo
ud<pcl::PointXYZI>); // 需要PointXYZI
    pcl::PointCloud<pcl::PointXYZI>::Ptr cloud_filtered(new pcl:
:PointCloud<pcl::PointXYZI>);

    // Fill in the cloud data
    pcl::PCDReader reader;
    // Replace the path below with the path where you saved your
 file
    reader.read<pcl::PointXYZI>(argv[1], *cloud);

    bilateralFilter(cloud, cloud_filtered);

    /*pcl::visualization::PCLVisualizer viewer;
    viewer.addPointCloud(cloud_filtered);
    viewer.spin();*/

    return (0);
}
```

- 剔除离群点

参考：http://www.pointclouds.org/documentation/tutorials/statistical_outlier.php

表面法线

表面法线是几何体表面的重要属性，属于特征描述范畴，由于下章关键点需要此操作，所以提前给出用法。

```
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
// 包含相关头文件
#include <pcl/features/normal_3d.h>
#include <pcl/visualization/pcl_visualizer.h>

typedef pcl::PointXYZ PointT;
typedef pcl::PointNormal PointNT; // 也可以pcl::Normal,但无法用PCL Visualizer显示。

int main(int argc, char** argv)
{
    // 读取点云
    pcl::PointCloud<PointT>::Ptr cloud(new pcl::PointCloud<PointT>);
    pcl::io::loadPCDFile(argv[1], *cloud);

    // 计算法向量
    pcl::NormalEstimation<PointT, PointNT> nest;
    //nest.setRadiusSearch(0.01); // 设置拟合时邻域搜索半径，最好用模型分辨率的倍数
    nest.setKSearch(50); // 设置拟合时采用的点数
    nest.setInputCloud(cloud);
    pcl::PointCloud<PointNT>::Ptr normals(new pcl::PointCloud<PointNT>);
    nest.compute(*normals);

    for (size_t i = 0; i < cloud->points.size(); ++i)
    {    // 生成时只生成了法向量，没有将原始点云信息拷贝，为了显示需要复制原信息
        // 也可用其他方法进行连接，如：pcl::concatenateFields
        normals->points[i].x = cloud->points[i].x;
        normals->points[i].y = cloud->points[i].y;
```

```
        normals->points[i].z = cloud->points[i].z;
    }

    // 显示
    pcl::visualization::PCLVisualizer viewer;
    viewer.addPointCloud(cloud, "cloud");
    int level = 100; // 多少条法向量集合显示成一条
    float scale = 0.01; // 法向量长度
    viewer.addPointCloudNormals<PointNT>(normals, level, scale,
"normals");

    viewer.spin();

    system("pause");
    return 0;
}
```

提取关键点

关键点也称为兴趣点，它是2D图像或3D点云或曲面模型上，可以通过定义检测标准来获取的具有稳定性、区别性的点集。从技术上来说，关键点的数量相比于原始点云或图像数据量上减小很多，与局部特征描述子结合一起，组成关键点描述子常用来形成原始数据的紧凑表示，而且不失代表性和描述性，从而加快后续识别、追踪等对数据的处理速度。故而，关键点提取就成为2D与3D信息处理中不可或缺的关键技术。

PCL中pcl_keypoints库目前提供几种常用的关键点检测算法，下面给出几种常用算法的实例。

- **ISSKeypoint3D**

参考文献： Yu Zhong, “Intrinsic shape signatures: A shape descriptor for 3D object recognition,”Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on ,vol., no., pp.689-696, Sept. 27 2009-Oct. 4 2009

```
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
// 包含相关头文件
#include <pcl/keypoints/iss_3d.h>

#include "resolution.h" // 用于计算模型分辨率

typedef pcl::PointXYZ PointT;

int main(int argc, char** argv)
{
    // 读取点云
    pcl::PointCloud<PointT>::Ptr cloud(new pcl::PointCloud<Point
T>);
    pcl::io::loadPCDFile(argv[1], *cloud);
    std::cout << "original cloud size : " << cloud->size() << st
d::endl;

    double resolution = computeCloudResolution(cloud);
```

```
    pcl::search::KdTree<pcl::PointXYZ>::Ptr tree(new pcl::search
::KdTree<pcl::PointXYZ>());

    pcl::ISSKeypoint3D<PointT, PointT> iss_detector;
    iss_detector.setSearchMethod(tree);
    iss_detector.setSalientRadius(6 * resolution);
    iss_detector.setNonMaxRadius(4 * resolution);
    iss_detector.setThreshold21(0.975);
    iss_detector.setThreshold32(0.975);
    iss_detector.setMinNeighbors(5);
    iss_detector.setNumberOfThreads(4);
    iss_detector.setInputCloud(cloud);

    pcl::PointCloud<PointT>::Ptr keys(new pcl::PointCloud<PointT
>);
    iss_detector.compute(*keys);
    std::cout << "key points size : " << keys->size() << std::en
dl;

    system("pause");
    return 0;
}
```

- **HarrisKeypoint3D**

HarrisKeypoint3D是对2D的Harris提取关键点算法的一个三维扩展。

```
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
// 包含相关头文件
#include <pcl/keypoints/harris_3d.h>

#include "resolution.h" // 用于计算模型分辨率

typedef pcl::PointXYZ PointT;

int main(int argc, char** argv)
{
    // 读取点云
```

```
    pcl::PointCloud<PointT>::Ptr cloud(new pcl::PointCloud<Point
T>);
    pcl::io::loadPCDFile(argv[1], *cloud);
    std::cout << "original cloud size : " << cloud->size() << st
d::endl;

    double resolution = computeCloudResolution(cloud);

    pcl::search::KdTree<pcl::PointXYZ>::Ptr tree(new pcl::search
::KdTree<pcl::PointXYZ>());

    pcl::HarrisKeypoint3D<PointT, pcl::PointXYZI> detector;
    pcl::PointCloud<pcl::PointXYZI>::Ptr keypoints_temp(new pcl:
:PointCloud<pcl::PointXYZI>);
    detector.setNonMaxSupression(true);
    detector.setRadiusSearch(10 * resolution);
    detector.setThreshold(1E-6);
    detector.setSearchMethod(tree); // 不写也可以，默认构建kdtree
    detector.setInputCloud(cloud);
    detector.compute(*keypoints_temp);
    pcl::console::print_highlight("Detected %d points !\n", keyp
oints_temp->size());
    pcl::PointCloud<PointT>::Ptr keys(new pcl::PointCloud<pcl::P
ointXYZ>);
    pcl::copyPointCloud(*keypoints_temp, *keys);

    system("pause");
    return 0;
}
```

- **SIFTKeypoint**

SIFTKeypoint是对2D的sift算法的一个扩展。参考：David G. Lowe, "Distinctive image features from scale-invariant keypoints," International Journal of Computer Vision, 60, 2 (2004), pp. 91-110.

```
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
// 包含相关头文件
```

```
#include <pcl/features/normal_3d.h>
#include <pcl/keypoints/sift_keypoint.h>

#include "resolution.h" // 用于计算模型分辨率

typedef pcl::PointXYZ PointT;

int main(int argc, char** argv)
{
    // 读取点云
    pcl::PointCloud<PointT>::Ptr cloud(new pcl::PointCloud<Point
T>);
    pcl::io::loadPCDFile(argv[1], *cloud);
    std::cout << "original cloud size : " << cloud->size() << st
d::endl;

    double resolution = computeCloudResolution(cloud); // 模型分
辨率

    // 法向量
    pcl::NormalEstimation<pcl::PointXYZ, pcl::PointNormal> ne;
    pcl::PointCloud<pcl::PointNormal>::Ptr cloud_normals(new pcl
::PointCloud<pcl::PointNormal>);
    pcl::search::KdTree<pcl::PointXYZ>::Ptr tree_n(new pcl::sear
ch::KdTree<pcl::PointXYZ>());
    ne.setInputCloud(cloud);
    ne.setSearchMethod(tree_n);
    //    ne.setRadiusSearch(10 * resolution);
    ne.setKSearch(50);
    ne.compute(*cloud_normals);

    // 拷贝数据
    for (size_t i = 0; i < cloud_normals->points.size(); ++i)
    {
        cloud_normals->points[i].x = cloud->points[i].x;
        cloud_normals->points[i].y = cloud->points[i].y;
        cloud_normals->points[i].z = cloud->points[i].z;
    }

    // sift参数
```

```
    const float min_scale = 0.001f;
    const int n_octaves = 5; //3
    const int n_scales_per_octave = 6; //4
    const float min_contrast = 0.001f;

    // 使用法向量作为强度计算关键点，还可以是rgb、z值或者自定义，具体参看A
PI
    pcl::SIFTKeypoint<pcl::PointNormal, PointT> sift; //PointT
可以是 pcl::PointWithScale包含尺度信息
    pcl::PointCloud<PointT> keys;
    pcl::search::KdTree<pcl::PointNormal>::Ptr tree(new pcl::sea
rch::KdTree<pcl::PointNormal>());
    sift.setSearchMethod(tree);
    sift.setScales(min_scale, n_octaves, n_scales_per_octave);
    sift.setMinimumContrast(min_contrast);
    sift.setInputCloud(cloud_normals);
    sift.compute(keys);
    std::cout << "No of SIFT points in the result are " << keys.
points.size() << std::endl;

    system("pause");
    return 0;
}
```

点云特征描述

3D点云特征描述与提取是点云信息处理中的最基础也是最关键的一部分，点云的识别、分割、重采样、配准、曲面重建等处理大部分算法，都严重依赖特征描述与提取的结果。从尺度上来分，一般分为局部特征描述和全局特征描述。http://robotica.unileon.es/index.php/PCL/OpenNI_tutorial_4:_3D_object_recognition_(descriptors)有各种描述子较为详细的介绍。 下面给出常用的PFH、FPFH、SHOT、RoPS使用实例。

- **PFH**

参考文献： (1) R.B. Rusu, N. Blodow, Z.C. Marton, M. Beetz. Aligning Point Cloud Views using Persistent Feature Histograms. In Proceedings of the 21st IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Nice, France, September 22-26 2008. (2) R.B. Rusu, Z.C. Marton, N. Blodow, M. Beetz. Learning Informative Point Classes for the Acquisition of Object Model Maps. In Proceedings of the 10th International Conference on Control, Automation, Robotics and Vision (ICARCV), Hanoi, Vietnam, December 17-20 2008.

```
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
// 包含相关头文件
#include <pcl/features/normal_3d.h>
#include <pcl/features/pfh.h>

#include "resolution.h" // 用于计算模型分辨率

typedef pcl::PointXYZ PointT;
typedef pcl::Normal PointNT;
typedef pcl::PFHSignature125 FeatureT;

int main(int argc, char** argv)
{
    // 读取点云
    pcl::PointCloud<PointT>::Ptr cloud(new pcl::PointCloud<Point
T>);
    pcl::io::loadPCDFile(argv[1], *cloud);
```

```
    // 读取关键点，也可以用之前提到的方法计算
    pcl::PointCloud<PointT>::Ptr keys(new pcl::PointCloud<PointT
>);
    pcl::io::loadPCDFile(argv[2], *keys);

    double resolution = computeCloudResolution(cloud);

    // 法向量
    pcl::NormalEstimation<PointT, PointNT> nest;
    //    nest.setRadiusSearch(10 * resolution);
    nest.setKSearch(10);
    nest.setInputCloud(cloud);
    nest.setSearchSurface(cloud);
    pcl::PointCloud<PointNT>::Ptr normals(new pcl::PointCloud<pc
l::Normal>);
    nest.compute(*normals);
    std::cout << "compute normal\n";

    // 关键点计算PFH描述子
    pcl::PointCloud<FeatureT>::Ptr features(new pcl::PointCloud<
FeatureT>);
    pcl::PFHEstimation<PointT, PointNT, FeatureT> fest;
    fest.setRadiusSearch(18 * resolution);
    fest.setSearchSurface(cloud);
    fest.setInputCloud(keys);
    fest.setInputNormals(normals);
    fest.compute(*features);
    std::cout << "compute feature\n";

    system("pause");
    return 0;
}
```

- **FPFH**

参考文献： (1) R.B. Rusu, N. Blodow, M. Beetz. Fast Point Feature Histograms (FPFH) for 3D Registration. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), Kobe, Japan, May 12-17 2009. (2) R.B.

Rusu, A. Holzbach, N. Blodow, M. Beetz. Fast Geometric Point Labeling using Conditional Random Fields. In Proceedings of the 22nd IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), St. Louis, MO, USA, October 11-15 2009.

相关类用法与PFH类似，只需要引入 `#include <pcl/features/fpfh.h>`；将 `typedef pcl::PFHSignature125 FeatureT` 替换为 `typedef pcl::FPFHSignature33 FeatureT`；将 `pcl::PFHEstimation<PointT, PointNT, FeatureT> fest;` 替换为 `pcl::FPFHEstimation<PointT, PointNT, FeatureT> fest;` 即可。

- **SHOT**

参考文献： (1) F. Tombari, S. Salti, L. Di Stefano Unique Signatures of Histograms for Local Surface Description. In Proceedings of the 11th European Conference on Computer Vision (ECCV), Heraklion, Greece, September 5-11 2010. (2) F. Tombari, S. Salti, L. Di Stefano A Combined Texture-Shape Descriptor For Enhanced 3D Feature Matching. In Proceedings of the 18th International Conference on Image Processing (ICIP), Brussels, Belgium, September 11-14 2011.

```
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
// 包含相关头文件
#include <pcl/features/shot.h>
#include <pcl/features/normal_3d.h>

#include "resolution.h" // 用于计算模型分辨率

typedef pcl::PointXYZ PointT;
typedef pcl::Normal PointNT;
typedef pcl::SHOT352 FeatureT;

int main(int argc, char** argv)
{
    // 读取点云
    pcl::PointCloud<PointT>::Ptr cloud(new pcl::PointCloud<Point
T>);
    pcl::io::loadPCDFile(argv[1], *cloud);
```

```
    // 读取关键点，也可以用之前提到的方法计算
    pcl::PointCloud<PointT>::Ptr keys(new pcl::PointCloud<PointT
>);
    pcl::io::loadPCDFile(argv[2], *keys);

    double resolution = computeCloudResolution(cloud);

    // 法向量
    pcl::NormalEstimation<PointT, PointNT> nest;
    //    nest.setRadiusSearch(5*resolution);
    nest.setKSearch(10);
    pcl::PointCloud<pcl::Normal>::Ptr cloud_normal(new pcl::Poin
tCloud<pcl::Normal>);
    nest.setInputCloud(cloud);
    nest.setSearchSurface(cloud);
    nest.compute(*cloud_normal);
    std::cout << "compute normal\n";

    pcl::SHOTEstimation<pcl::PointXYZ, pcl::Normal, pcl::SHOT352
> shot;
    shot.setRadiusSearch(18 * resolution);
    shot.setInputCloud(keys);
    shot.setSearchSurface(cloud);
    shot.setInputNormals(cloud_normal);
    //shot.setInputReferenceFrames(lrf);  //也可以自己传入局部坐标系
    pcl::PointCloud<FeatureT>::Ptr features(new pcl::PointCloud<
FeatureT>);
    shot.compute(*features);
    std::cout << "compute feature\n";

    system("pause");
    return 0;
}
```

- **RoPS**

参考文献： "Rotational Projection Statistics for 3D Local Surface Description and Object Recognition" by Yulan Guo, Ferdous Sohel, Mohammed Bennamoun, Min Lu and Jianwei Wan.

由于RoPS是基于网格数据，所以如果输入的是点云数据需要先进行网格化处理。

```
#include <pcl/io/pcd_io.h>
#include <pcl/features/rops_estimation.h>
#include <pcl/features/normal_3d.h>
#include <pcl/surface/gp3.h>

int main(int argc, char** argv)
{
    // 加载点云
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointClou
d<pcl::PointXYZ>);
    pcl::io::loadPCDFile(argv[1], *cloud);

    // 加载关键点
    pcl::PointCloud<pcl::PointXYZ>::Ptr key_points(new pcl::Poin
tCloud<pcl::PointXYZ>);
    pcl::io::loadPCDFile(argv[2], *key_points);

    // 计算法向量
    pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> n;
    pcl::PointCloud<pcl::Normal>::Ptr normals(new pcl::PointClou
d<pcl::Normal>);
    pcl::search::KdTree<pcl::PointXYZ>::Ptr tree(new pcl::search
::KdTree<pcl::PointXYZ>);
    tree->setInputCloud(cloud);
    n.setInputCloud(cloud);
    n.setSearchMethod(tree);
    n.setKSearch(20);
    n.compute(*normals);

    // 连接数据
    pcl::PointCloud<pcl::PointNormal>::Ptr cloud_with_normals(ne
w pcl::PointCloud<pcl::PointNormal>);
    pcl::concatenateFields(*cloud, *normals, *cloud_with_normals
);
    //* cloud_with_normals = cloud + normals

    // ---- rops基于网格，所以要先将pcd点云数据重建网格 ---
```

```
    // Create search tree*
    pcl::search::KdTree<pcl::PointNormal>::Ptr tree2(new pcl::se
arch::KdTree<pcl::PointNormal>);
    tree2->setInputCloud(cloud_with_normals);

    pcl::GreedyProjectionTriangulation<pcl::PointNormal> gp3;//
Initialize objects
    pcl::PolygonMesh triangles;
    // Set the maximum distance between connected points (maximu
m edge length)
    gp3.setSearchRadius(0.025);
    gp3.setMu(2.5); // Set typical values for the parameters
    gp3.setMaximumNearestNeighbors(100);
    gp3.setMaximumSurfaceAngle(M_PI / 4); // 45 degrees
    gp3.setMinimumAngle(M_PI / 18); // 10 degrees
    gp3.setMaximumAngle(2 * M_PI / 3); // 120 degrees
    gp3.setNormalConsistency(false);
    gp3.setInputCloud(cloud_with_normals);
    gp3.setSearchMethod(tree2);
    gp3.reconstruct(triangles);    // Get result

    // ----- rops 描述-------
    // 由于pcl_1.8.0中rops还没有定义好的结构，所以采用pcl::Histogram存
储描述子
    pcl::ROPSEstimation<pcl::PointXYZ, pcl::Histogram<135>> rops
;
    rops.setInputCloud(key_points);
    rops.setSearchSurface(cloud);
    rops.setNumberOfPartitionBins(5);
    rops.setNumberOfRotations(3);
    rops.setRadiusSearch(0.01);
    rops.setSupportRadius(0.01);
    rops.setTriangles(triangles.polygons);
    rops.setSearchMethod(pcl::search::KdTree<pcl::PointXYZ>::Ptr
(new pcl::search::KdTree < pcl::PointXYZ>));
    //feature size = number_of_rotations * number_of_axis_to_rot
ate_around * number_of_projections * number_of_central_moments
    //unsigned int feature_size = number_of_rotations_ * 3 * 3 *
 5; //计算出135
    pcl::PointCloud<pcl::Histogram<135>> description;
```

```
    rops.compute(description);  // 结果计算的是描述子。。需传入inputcloud和surface
    std::cout << "size is " << description.points.size()<<std::endl;
    //pcl::io::savePCDFile("rops_des.pcd", description); // 此句出错！！pcl::Histogram没有对应的保存方法

    system("pause");
    return 0;
}
```

一个完整的实例

该代码中完成了提取关键点、特征描述、计算匹配对与旋转平移矩阵、显示等操作。

```
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
// 包含相关头文件
#include <pcl/keypoints/harris_3d.h>
#include <pcl/features/normal_3d.h>
#include <pcl/features/shot.h>
#include <pcl/registration/correspondence_estimation.h>
#include <pcl/visualization/pcl_visualizer.h>
/*
#include <pcl/common/transforms.h>
#include <pcl/registration/transformation_estimation_svd.h>
*/
#include "resolution.h" // 用于计算模型分辨率
#include "getTransformation.h" //计算旋转平移矩阵

typedef pcl::PointXYZ PointT;
typedef pcl::Normal PointNT;
typedef pcl::SHOT352 FeatureT;

// 获取Harris关键点
void getHarrisKeyPoints(const pcl::PointCloud<PointT>::Ptr &clou
d, double resolution,
    pcl::PointCloud<PointT>::Ptr &keys)
{
    pcl::HarrisKeypoint3D<PointT, pcl::PointXYZI> detector;
    pcl::PointCloud<pcl::PointXYZI>::Ptr keypoints_temp(new pcl:
:PointCloud<pcl::PointXYZI>);
    detector.setNonMaxSupression(true);
    detector.setRadiusSearch(10 * resolution);
    detector.setThreshold(1E-6);
    detector.setInputCloud(cloud);
    detector.compute(*keypoints_temp);
    pcl::console::print_highlight("Detected %d points !\n", keyp
```

```
oints_temp->size());
    pcl::copyPointCloud(*keypoints_temp, *keys);
}

void getFeatures(const pcl::PointCloud<PointT>::Ptr &cloud, cons
t pcl::PointCloud<PointT>::Ptr &keys,
    double resolution, pcl::PointCloud<FeatureT>::Ptr features)
{
    // 法向量
    pcl::NormalEstimation<PointT, PointNT> nest;
    nest.setKSearch(10);
    pcl::PointCloud<PointNT>::Ptr cloud_normal(new pcl::PointClo
ud<PointNT>);
    nest.setInputCloud(cloud);
    nest.setSearchSurface(cloud);
    nest.compute(*cloud_normal);
    std::cout << "compute normal\n";

    pcl::SHOTEstimation<PointT, PointNT, FeatureT> shot;
    shot.setRadiusSearch(18 * resolution);
    shot.setInputCloud(keys);
    shot.setSearchSurface(cloud);
    shot.setInputNormals(cloud_normal);
    shot.compute(*features);
    std::cout << "compute feature\n";
}

int main(int argc, char** argv)
{
    // 读取点云
    pcl::PointCloud<PointT>::Ptr cloud_src(new pcl::PointCloud<P
ointT>);
    pcl::io::loadPCDFile(argv[1], *cloud_src);

    pcl::PointCloud<PointT>::Ptr cloud_tgt(new pcl::PointCloud<P
ointT>);
    pcl::io::loadPCDFile(argv[2], *cloud_tgt);

    // 计算模型分辨率
    double resolution = computeCloudResolution(cloud_src);
```

```
    // 提取关键点
    pcl::PointCloud<PointT>::Ptr keys_src(new pcl::PointCloud<pc
l::PointXYZ>);
    getHarrisKeyPoints(cloud_src, resolution, keys_src);

    pcl::PointCloud<PointT>::Ptr keys_tgt(new pcl::PointCloud<pc
l::PointXYZ>);
    getHarrisKeyPoints(cloud_tgt, resolution, keys_tgt);

    // 特征描述
    pcl::PointCloud<FeatureT>::Ptr features_src(new pcl::PointCl
oud<FeatureT>);
    getFeatures(cloud_src, keys_src, resolution, features_src);

    pcl::PointCloud<FeatureT>::Ptr features_tgt(new pcl::PointCl
oud<FeatureT>);
    getFeatures(cloud_tgt, keys_tgt, resolution, features_tgt);

    // 计算对应匹配关系
    pcl::registration::CorrespondenceEstimation<FeatureT, Featur
eT> cor_est;
    cor_est.setInputCloud(features_src);
    cor_est.setInputTarget(features_tgt);
    boost::shared_ptr<pcl::Correspondences> cor(new pcl::Corresp
ondences);
    cor_est.determineReciprocalCorrespondences(*cor);
    std::cout << "compute Correspondences " << cor->size() << st
d::endl;

    // 计算旋转平移矩阵
    Eigen::Matrix4f transformation(Eigen::Matrix4f::Identity());
    getTransformation(keys_src,keys_tgt, resolution, *cor, trans
formation);

    /*
    // 也可以不用关键点对应关系直接获取旋转平移矩阵
    Eigen::Matrix4f transformation(Eigen::Matrix4f::Identity());
    pcl::registration::TransformationEstimationSVD<pcl::PointXYZ
, pcl::PointXYZ> svd;
```

```
    svd.estimateRigidTransformation(*cloud_src, *cloud_tgt, tran
sformation);
    */

    // 显示
    pcl::visualization::PCLVisualizer viewer;
    viewer.addPointCloud(cloud_src, "cloud_src"); // 显示点云
    viewer.addPointCloud(cloud_tgt, "cloud_tgt");

    pcl::visualization::PointCloudColorHandlerCustom<PointT> red
_src(keys_src, 255, 0, 0);
    pcl::visualization::PointCloudColorHandlerCustom<PointT> red
_tgt(keys_tgt, 255, 0, 0);
    viewer.addPointCloud(keys_src, red_src, "keys_src"); //显示关
键点，红色，加粗
    viewer.addPointCloud(keys_tgt, red_tgt, "keys_tgt");
    viewer.setPointCloudRenderingProperties(pcl::visualization::
PCL_VISUALIZER_POINT_SIZE, 4, "keys_src");
    viewer.setPointCloudRenderingProperties(pcl::visualization::
PCL_VISUALIZER_POINT_SIZE, 4, "keys_tgt");

    for (size_t i = 0; i < cor->size(); ++i) // 显示关键点匹配关系
    {
        PointT temp1 = keys_src->points[cor->at(i).index_query];
        PointT temp2 = keys_tgt->points[cor->at(i).index_match];
        std::stringstream ss;
        ss << "line_" << i;
        viewer.addLine(temp1, temp2, ss.str());
    }

    pcl::PointCloud<PointT>::Ptr cloud_trans(new pcl::PointCloud
<pcl::PointXYZ>);
    pcl::transformPointCloud(*cloud_src, *cloud_trans, transform
ation); // 将原点云旋转

    pcl::visualization::PointCloudColorHandlerCustom<PointT> gre
en_trans(cloud_trans, 0, 255, 0);
    viewer.addPointCloud(cloud_trans, green_trans, "cloud_trans"
);
```

```
    viewer.spin();

    system("pause");
    return 0;
}
```

精配准

精配准大多采用ICP(Iterative Closest Point)算法或者其变种算法完成。ICP算法本质上是基于最小二乘法的最优配准方法。该算法重复进行选择对应关系点对，计算最优刚体变换，直到满足正确配准的收敛精度要求。

改进的ICP可以采用GPU对迭代进行加速，或者针对最近点选取上采用Point to Point、Point to Plane、Point to Projection等一些方式完成，或者针对收敛函数做一些改变。

PCL中也提供了ICP算法和一些改进算法。

- **ICP**

参考：http://pointclouds.org/documentation/tutorials/iterative_closest_point.php#iterative-closest-point

- **GeneralizedIterativeClosestPoint**

论文：http://www.robots.ox.ac.uk/~avsegal/resources/papers/Generalized_ICP.pdf

```
pcl::GeneralizedIterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ> gicp;

gicp.setInputSource(cloud_src);
gicp.setInputTarget(cloud_tgt);

gicp.setMaximumIterations(100);
gicp.setTransformationEpsilon(1e-6);
gicp.setEuclideanFitnessEpsilon(0.1);
gicp.setMaxCorrespondenceDistance(0.01);

pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_final(new pcl::PointCloud<pcl::PointXYZ>);
gicp.align(*cloud_final);
```

- **Sparse ICP**

参考:http://lgg.epfl.ch/sparseicp

曲面重建

曲面重建可以用于逆向工程、数据可视化、自动化建模等领域。PCL中目前实现了多种基于点云的曲面重建算法，如：泊松曲面重建、贪婪投影三角化、移动立方体、GridProjection、EarClipping等算法。可以参考http://pointclouds.org/documentation/tutorials/相关内容、CSDN博客http://blog.csdn.net/xuezhisdc/article/details/51034359/。

还有一个特殊的B样条拟合(B-splines)，需要在pcl编译时特殊支持。OrganizedFastMesh需要输入有序的点云。 而移动最小二乘(mls)虽然放在surface模块下，此类并不能输出拟合后的表面，不能生成Mesh或者Triangulations，只是将点云进行了MLS的映射，使得输出的点云更加平滑。

- 贪婪投影三角化算法

贪婪投影三角化算法对有向点云进行三角化，具体方法是先将有向点云投影到某一局部二维坐标平面内，再在坐标平面内进行平面内的三角化，再根据平面内三位点的拓扑连接关系获得一个三角网格曲面模型。

贪婪投影三角化算法原理是处理一系列可以使网格“生长扩大”的点（边缘点），延伸这些点直到所有符合几何正确性和拓扑正确性的点都被连上。该算法的优点是可以处理来自一个或者多个扫描仪扫描得到并且有多个连接处的散乱点云。但该算法也有一定的局限性，它更适用于采样点云来自于表面连续光滑的曲面并且点云密度变化比较均匀的情况。

```
#include <pcl/point_types.h>
#include <pcl/io/pcd_io.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/features/normal_3d.h>
#include <pcl/surface/gp3.h>
#include <pcl/visualization/pcl_visualizer.h>
#include "resolution.h"

int main(int argc, char** argv)
{
    // Load input file into a PointCloud<T> with an appropriate
type
```

```
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointClou
d<pcl::PointXYZ>);
    pcl::PCLPointCloud2 cloud_blob;
    pcl::io::loadPCDFile(argv[1], cloud_blob);
    pcl::fromPCLPointCloud2(cloud_blob, *cloud);
    //* the data should be available in cloud

    double resolution = computeCloudResolution(cloud);

    // Normal estimation*
    pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> n;
    pcl::PointCloud<pcl::Normal>::Ptr normals(new pcl::PointClou
d<pcl::Normal>);
    pcl::search::KdTree<pcl::PointXYZ>::Ptr tree(new pcl::search
::KdTree<pcl::PointXYZ>);
    tree->setInputCloud(cloud);
    n.setInputCloud(cloud);
    n.setSearchMethod(tree);
    n.setKSearch(20);
    n.compute(*normals);
    // normals should not contain the point normals + surface cu
rvatures

    // Concatenate the XYZ and normal fields*
    pcl::PointCloud<pcl::PointNormal>::Ptr cloud_with_normals(ne
w pcl::PointCloud<pcl::PointNormal>);
    pcl::concatenateFields(*cloud, *normals, *cloud_with_normals
);
    // cloud_with_normals = cloud + normals

    // Create search tree*
    pcl::search::KdTree<pcl::PointNormal>::Ptr tree2(new pcl::se
arch::KdTree<pcl::PointNormal>);
    tree2->setInputCloud(cloud_with_normals);

    // Initialize objects
    pcl::GreedyProjectionTriangulation<pcl::PointNormal> gp3;
    pcl::PolygonMesh triangles;

    // Set the maximum distance between connected points (maximu
```

```
m edge length)
    gp3.setSearchRadius(5 * resolution);  //设置连接点之间的最大距离（最大边长）用于确定k近邻的球半径【默认值 0】
    // Set typical values for the parameters
    gp3.setMu(2.5); //设置最近邻距离的乘子，以得到每个点的最终搜索半径【默认值 0】
    gp3.setMaximumNearestNeighbors(100); //设置搜索的最近邻点的最大数量
    gp3.setMaximumSurfaceAngle(M_PI / 4); // 45 degrees（pi）最大平面角
    gp3.setMinimumAngle(M_PI / 18);  // 10 degrees 每个三角的最小角度
    gp3.setMaximumAngle(2 * M_PI / 3); // 120 degrees 每个三角的最大角度
    gp3.setNormalConsistency(false); //如果法向量一致，设置为true

    // Get result
    gp3.setInputCloud(cloud_with_normals);
    gp3.setSearchMethod(tree2);
    gp3.reconstruct(triangles);

    // Additional vertex information
    std::vector<int> parts = gp3.getPartIDs();
    std::vector<int> states = gp3.getPointStates();

    // show
    pcl::visualization::PCLVisualizer viewer;
    viewer.addPolygonMesh(triangles, "mesh");
    viewer.spin();

    // Finish
    return (0);
}
```

- 泊松曲面重建

泊松曲面重建基于泊松方程。根据泊松方程使用矩阵迭代求出近似解，采用移动立方体算法提取等值面，对所测数据点集重构出被测物体的模型，泊松方程在边界处的误差为零，因此得到的模型不存在假的表面框。

(http://blog.csdn.net/jennychenhit/article/details/52126156?locationNum=8)

```
#include <pcl/surface/poisson.h>

pcl::Poisson<pcl::PointNormal> pn;
pn.setConfidence(false);
pn.setDegree(2);
pn.setDepth(8);
pn.setIsoDivide(8);
pn.setManifold(false);
pn.setOutputPolygons(false);
pn.setSamplesPerNode(3.0);
pn.setScale(1.25);
pn.setSolverDivide(8);
pn.setSearchMethod(tree2);
pn.setInputCloud(cloud_with_normals);
pcl::PolygonMesh mesh;
pn.performReconstruction(mesh);
(其他同上)
```

- 移动立方体

MarchingCubes(移动立方体)算法是目前三围数据场等值面生成中最常用的方法。它实际上是一个分而治之的方法，把等值面的抽取分布于每个体素中进行。对于每个被处理的体素，以三角面片逼近其内部的等值面片。每个体素是一个小立方体，构造三角面片的处理过程对每个体素都“扫描”一遍，就好像一个处理器在这些体素上移动一样，由此得名移动立方体算法。

```
#include <pcl/surface/marching_cubes_hoppe.h>

pcl::MarchingCubes<pcl::PointNormal>::Ptr mc(new pcl::MarchingCu
besHoppe<pcl::PointNormal>);
//设置MarchingCubes对象的参数
mc->setIsoLevel(0.0f);
mc->setGridResolution(5, 5, 5);
mc->setPercentageExtendGrid(0.0f);
mc->setSearchMethod(tree2);
mc->setInputCloud(cloud_with_normals);

pcl::PolygonMesh mesh;//执行重构，结果保存在mesh中
mc->reconstruct(mesh);
(其他同上)
```

- **B**样条拟合

参考：http://pointclouds.org/documentation/tutorials/bspline_fitting.php#bspline-fitting

- 移动最小二乘

虽说此类放在了Surface下面，但是通过反复的研究与使用，发现此类并不能输出拟合后的表面，不能生成Mesh或者Triangulations，只是将点云进行了MLS的映射，使得输出的点云更加平滑，进行上采样和计算法向量。

```
#include <iostream>
#include <pcl/point_types.h>
#include <pcl/io/pcd_io.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/surface/mls.h>
#include <pcl/visualization/pcl_visualizer.h>
#include "resolution.h"

int main(int argc, char** argv)
{
    // Load input file into a PointCloud<T> with an appropriate
type
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointClou
```

```
d<pcl::PointXYZ>());
    // Load bun0.pcd -- should be available with the PCL archive
 in test
    pcl::io::loadPCDFile(argv[1], *cloud);

    double resolution = computeCloudResolution(cloud);

    // Create a KD-Tree
    pcl::search::KdTree<pcl::PointXYZ>::Ptr tree(new pcl::search
::KdTree<pcl::PointXYZ>);

    // Output has the PointNormal type in order to store the nor
mals calculated by MLS
    pcl::PointCloud<pcl::PointNormal> mls_points;

    // Init object (second point type is for the normals, even i
f unused)
    pcl::MovingLeastSquares<pcl::PointXYZ, pcl::PointNormal> mls
;
    mls.setInputCloud(cloud);
    mls.setComputeNormals(true); //我们都知道表面重构时需要估计点云的
法向量，这里MLS提供了一种方法来估计点云法向量。（如果是true的话注意输出数据
格式）。
    mls.setPolynomialFit(true); //对于法线的估计是有多项式还是仅仅依靠
切线。
    mls.void setPolynomialOrder(3); //MLS拟合曲线的阶数，这个阶数在构
造函数里默认是2，但是参考文献给出最好选择3或者4
    mls.setSearchMethod(tree);
    mls.setSearchRadius(10 * resolution);
    // Reconstruct
    mls.process(mls_points);

    // Save output
    //pcl::io::savePCDFile("bun0-mls.pcd", mls_points);
    // show
    pcl::visualization::PCLVisualizer viewer_1;
    viewer_1.addPointCloud(cloud, "cloud");
    viewer_1.spin();

    pcl::visualization::PCLVisualizer viewer_2;
```

```
    pcl::PointCloud<pcl::PointXYZ>::Ptr result(new pcl::PointClo
ud<pcl::PointXYZ>);
    pcl::copyPointCloud(mls_points, *result);
    viewer_2.addPointCloud(result, "mls_points");
    viewer_2.spin();

    return 0;
}
```

曲面分割

参考：http://pointclouds.org/documentation/tutorials/#segmentation-tutorial

文献综述： 刘进,武仲科,周明全.点云模型分割及应用技术综述[J].计算机科学,2011,38(04):21-24+71

新建项目

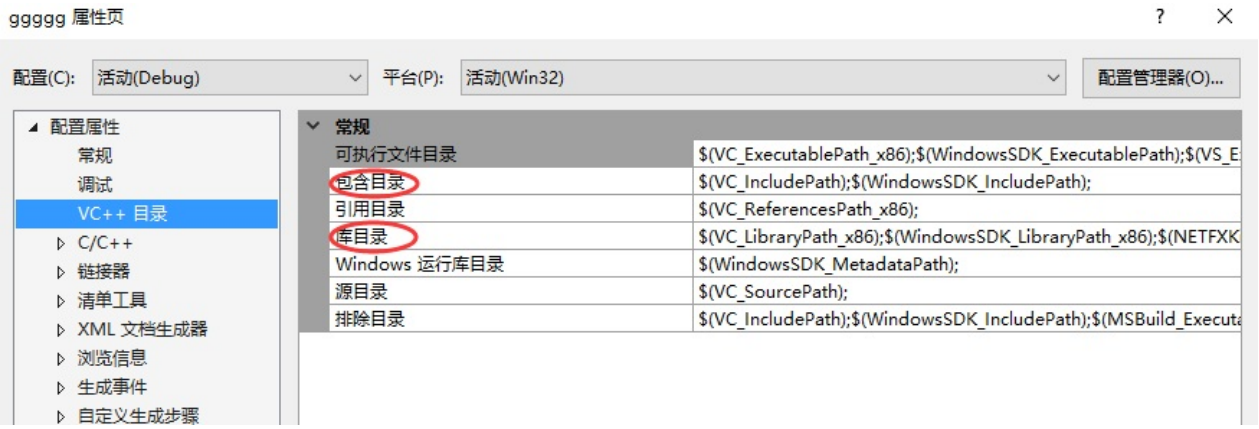
- **Cmake**创建项目

cmake创建VS项目实质就是根据CMakeLists.txt文件，搜索相关的环境变量查找文件编译时所需的路径及文件。如果编译成功则按照所选版本规则生成vs所需的.sln、.vcxproj等文件。

- **VS**直接新建项目

1、新建C++空项目 2、新建所需的源文件、头文件.cpp和.h文件。 3、项目上右键->属性，添加“附加包含目录”、“附加库目录”、“附加依赖项”

添加所有库的include目录、lib目录



添加所需的lib文件

注：此处可以使用属性表添加。

4、vs2013以上运行项目时，会报unsafe之类的错误，将检查关闭即可。

检查改成“否”

5、如果运行需要添加参数，可以在调试“命令参数”处添加，添加后，可在main函数中通过argv参数获取。

常用函数(1)

给出一些可能会用到的工具函数。代码来自
https://segmentfault.com/a/1190000007125502

- 时间计算

pcl中计算程序运行时间有很多函数，其中利用控制台的时间计算是：

```
#include <pcl/console/time.h>

pcl::console::TicToc time; time.tic();

+程序段 +

cout<<time.toc()/1000<<"s"<<endl;
```

就可以以秒输出“程序段”的运行时间。

- **pcl::PointCloud::Ptr**和**pcl::PointCloud**的两个类相互转换

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

pcl::PointCloud<pcl::PointXYZ>::Ptr cloudPointer(new pcl::PointC
loud<pcl::PointXYZ>);
pcl::PointCloud<pcl::PointXYZ> cloud;
cloud = *cloudPointer;
cloudPointer = cloud.makeShared();
```

- 如何查找点云的**x**，**y**，**z**的极值？

```
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/common/common.h>
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud;
cloud = pcl::PointCloud<pcl::PointXYZ>::Ptr (new pcl::PointCloud
<pcl::PointXYZ>);
pcl::io::loadPCDFile<pcl::PointXYZ> ("your_pcd_file.pcd", *cloud
);
pcl::PointXYZ minPt, maxPt;
pcl::getMinMax3D (*cloud, minPt, maxPt);
```

- 知道需要保存点的索引，从原点云中拷贝点到新点云

```
#include <pcl/io/pcd_io.h>
#include <pcl/common/impl/io.hpp>
#include <pcl/point_types.h>
#include <pcl/point_cloud.h>

pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pc
l::PointXYZ>);
pcl::io::loadPCDFile<pcl::PointXYZ>("C:\office3-after21111.pcd",
 *cloud);
pcl::PointCloud<pcl::PointXYZ>::Ptr cloudOut(new pcl::PointCloud
<pcl::PointXYZ>);
std::vector<int > indexs = { 1, 2, 5 };
pcl::copyPointCloud(*cloud, indexs, *cloudOut);
```

- 如何从点云里删除和添加点

```
#include <pcl/io/pcd_io.h>
#include <pcl/common/impl/io.hpp>
#include <pcl/point_types.h>
#include <pcl/point_cloud.h>

pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pc
l::PointXYZ>);
pcl::io::loadPCDFile<pcl::PointXYZ>("C:\office3-after21111.pcd",
 *cloud);
pcl::PointCloud<pcl::PointXYZ>::iterator index = cloud->begin();
cloud->erase(index);//删除第一个
index = cloud->begin() + 5;
cloud->erase(cloud->begin());//删除第5个
pcl::PointXYZ point = { 1, 1, 1 };
//在索引号为5的位置1上插入一点，原来的点后移一位
cloud->insert(cloud->begin() + 5, point);
cloud->push_back(point);//从点云最后面插入一点
std::cout << cloud->points[5].x;//输出1
```

- 链接两个点云字段（两点云大小必须相同）

```
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<p
cl::PointXYZ>);
pcl::io::loadPCDFile("/home/yxg/pcl/pcd/mid.pcd",*cloud);
pcl::NormalEstimation<pcl::PointXYZ,pcl::Normal> ne;
ne.setInputCloud(cloud);
pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::K
dTree<pcl::PointXYZ>());
ne.setSearchMethod(tree);
pcl::PointCloud<pcl::Normal>::Ptr cloud_normals(new pcl::PointCl
oud<pcl::Normal>());
ne.setKSearch(8);
 //ne.setRadisuSearch(0.3);
ne.compute(*cloud_normals);
pcl::PointCloud<pcl::PointNormal>::Ptr cloud_with_nomal (new pcl
::PointCloud<pcl::PointNormal>);
pcl::concatenateFields(*cloud,*cloud_normals,*cloud_with_nomal);
```

- 如何从点云中删除无效点

pcl中的无效点是指：点的某一坐标值为nan.

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

using namespace std;
typedef pcl::PointXYZRGBA point;
typedef pcl::PointCloud<point> CloudType;

int main (int argc,char **argv)
{
  CloudType::Ptr cloud (new CloudType);
  CloudType::Ptr output (new CloudType);

  pcl::io::loadPCDFile(argv[1],*cloud);
  cout<<"size is:"<<cloud->size()<<endl;

  vector<int> indices;
  pcl::removeNaNFromPointCloud(*cloud,*output,indices);
  cout<<"output size:"<<output->size()<<endl;

  pcl::io::savePCDFile("out.pcd",*output);

  return 0;
}
```

- 计算质心

```
Eigen::Vector4f centroid;  //质心
pcl::compute3DCentroid(*cloud_smoothed,centroid); //估计质心的坐标
```

- 从网格提取顶点（将网格转化为点）

```
#include <pcl/io/io.h>
#include <pcl/io/pcd_io.h>
#include <pcl/io/obj_io.h>
#include <pcl/PolygonMesh.h>
#include <pcl/point_cloud.h>
#include <pcl/io/vtk_lib_io.h>//loadPolygonFileOBJ所属头文件；
#include <pcl/io/vtk_io.h>
#include <pcl/io/ply_io.h>
#include <pcl/point_types.h>
using namespace pcl;
int main(int argc,char **argv)
{
   pcl::PolygonMesh mesh;
  //   pcl::io::loadPolygonFileOBJ(argv[1], mesh);
  pcl::io::loadPLYFile(argv[1],mesh);
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new     pcl::PointCl
oud<pcl::PointXYZ>);
  pcl::fromPCLPointCloud2(mesh.cloud, *cloud);
  pcl::io::savePCDFileASCII("result.pcd", *cloud);
  return 0;
}
```

以上代码可以从.obj或.ply面片格式转化为点云类型。

常用函数(2)

以下代码为平时测试所用代码，实现并不严谨，仅供参考！

- 计算模型分辨率

resolution.h

```
#include <pcl/io/pcd_io.h>
#include <pcl/search/kdtree.h>

// This function by Tommaso Cavallari and Federico Tombari, take
n from the tutorial
// http://pointclouds.org/documentation/tutorials/correspondence
_grouping.php
double computeCloudResolution(const pcl::PointCloud<pcl::PointXY
Z>::ConstPtr& cloud)
{
    double resolution = 0.0;
    int numberOfPoints = 0;
    int nres;
    std::vector<int> indices(2);
    std::vector<float> squaredDistances(2);
    pcl::search::KdTree<pcl::PointXYZ> tree;
    tree.setInputCloud(cloud);

    for (size_t i = 0; i < cloud->size(); ++i)
    {
        if (!pcl_isfinite((*cloud)[i].x))
            continue;

        // Considering the second neighbor since the first is th
e point itself.
        nres = tree.nearestKSearch(i, 2, indices, squaredDistanc
es);
        if (nres == 2)
        {
            resolution += sqrt(squaredDistances[1]);
            ++numberOfPoints;
        }
    }
    if (numberOfPoints != 0)
        resolution /= numberOfPoints;

    return resolution;
}
```

- 计算旋转平移矩阵

getTransformation.h

```
#ifndef _GET_TRANSFORMATION_
#define _GET_TRANSFORMATION_

#include <pcl/point_types.h>
#include <pcl/common/distances.h>
#include <boost/random.hpp>
#include <pcl/registration/transformation_estimation.h>
#include <pcl/registration/transformation_estimation_3point.h>
#include <pcl/search/pcl_search.h>

int getTransform(pcl::PointCloud<pcl::PointXYZ>::Ptr& src, pcl::
PointCloud<pcl::PointXYZ>::Ptr& tar,
    double resolution,
    pcl::Correspondences &correspondences, Eigen::Matrix4f & tra
nsform)
{
    int cor_size = correspondences.size();
    if (cor_size < 3)
    {
        pcl::console::print_error("matching less 3");
        return (-1);
    }

    float fitness_score = FLT_MAX;
    boost::mt19937 gen;
    boost::uniform_int<> dist(0, cor_size - 1);
    boost::variate_generator<boost::mt19937&, boost::uniform_int
<>> die(gen, dist);
    pcl::registration::TransformationEstimation<pcl::PointXYZ, p
cl::PointXYZ, float>::Ptr transformationEstimation(new pcl::regi
stration::TransformationEstimation3Point <pcl::PointXYZ, pcl::Po
intXYZ>);
    pcl::search::KdTree<pcl::PointXYZ> tree;
    tree.setInputCloud(tar);
    //    float max_inlier_dist_sqr_ = 0.0064;
    for (int i = 0; i < 10000; ++i)
```

```
    {
        int cor1, cor2, cor3;
        cor1 = die();
        while (true)
        {
            cor2 = die();
            if (cor2 != cor1)
                break;
        }
        while (true)
        {
            cor3 = die();
            if (cor3 != cor1 && cor3 != cor2)
                break;
        }

        pcl::Correspondences correspondences_temp;
        correspondences_temp.push_back(correspondences[cor1]);
        correspondences_temp.push_back(correspondences[cor2]);
        correspondences_temp.push_back(correspondences[cor3]);
        Eigen::Matrix4f transform_temp;
        transformationEstimation->estimateRigidTransformation(*s
rc, *tar, correspondences_temp, transform_temp);

        // 先看三个对应点变换后是否匹配
        pcl::PointCloud<pcl::PointXYZ> src_temp;
        pcl::PointCloud<pcl::PointXYZ> tar_temp;
        src_temp.push_back(src->points[correspondences[cor1].ind
ex_query]);
        tar_temp.push_back(tar->points[correspondences[cor1].ind
ex_match]);
        src_temp.push_back(src->points[correspondences[cor2].ind
ex_query]);
        tar_temp.push_back(tar->points[correspondences[cor2].ind
ex_match]);
        src_temp.push_back(src->points[correspondences[cor3].ind
ex_query]);
        tar_temp.push_back(tar->points[correspondences[cor3].ind
ex_match]);
```

```
        pcl::PointCloud<pcl::PointXYZ> src_transformed;
        pcl::transformPointCloud(src_temp, src_transformed, tran
sform_temp);
        float mse = 0.f;
        for (int k = 0; k < 3; ++k)
        {
            mse += pcl::squaredEuclideanDistance(src_transformed
.points[k], tar_temp.points[k]);
        }
        mse /= 3;
        if (mse > 2* resolution)
            continue;

        // 整体变换、得出评价
        pcl::PointCloud<pcl::PointXYZ> match_transformed;
        pcl::transformPointCloud(*src, match_transformed, transf
orm_temp);

        std::vector <int> ids;
        std::vector <float> dists_sqr;
        float score = 0.f;
        for (int k = 0; k < match_transformed.size(); ++k)
        {
            tree.nearestKSearch(src->points[k], 1, ids, dists_sq
r);
            score += dists_sqr[0];
            //            score += (dists_sqr[0] < max_inlier_di
st_sqr_ ? dists_sqr[0] : max_inlier_dist_sqr_);
        }
        score /= match_transformed.size();
        //        score /= (max_inlier_dist_sqr_ * match_transfo
rmed.size());

        if (score < fitness_score)
        {
            fitness_score = score;
            transform = transform_temp;
        }

    }
```

```
    return 0;
}

#endif
```

- 旋转平移矩阵评估

参考论文： Mian A, Bennamoun M, Owens R. A novel representation and feature matching algorithm for automatic pairwise registration of range images[J]. International Journal of Computer Vision, 66(1), 19–40.

evaluation.h

```
#include <iostream>
#include <pcl/common/transforms.h>
#include <pcl/registration/transformation_estimation_svd.h>

using namespace std;

//compute ideal ratation and translation matrix
Eigen::Matrix4f createIdealTransformationMatrix(float tx, float
ty, float tz, float rx, float ry, float rz)
{
    ////rotating and translating point cloud
    ////m and m1 are both the unit matrixes
    Eigen::Matrix4f m = Eigen::Matrix4f::Identity();
    Eigen::Matrix4f m1 = Eigen::Matrix4f::Identity();
    //tx,ty,tz are translation values of x ,y and z axis
    m1(0, 3) = tx;
    m1(1, 3) = ty;
    m1(2, 3) = tz;
    //cout << "m1" << endl << m1 << endl;
    m = m * m1;   //apply the translation
    //cout << "m" << endl << m << endl;
    //rotating the point cloud along the x axis
    m1 = Eigen::Matrix4f::Identity();
    rx = rx / 180.0 * M_PI;
    m1(1, 1) = cos(rx);
    m1(1, 2) = -sin(rx);
    m1(2, 1) = sin(rx);
```

```
    m1(2, 2) = cos(rx);
    //cout << "m1" << endl << m1 << endl;
    m = m * m1;   //apply the rotation along x axis
    //cout << "m" << endl << m << endl;
    //rotating the point cloud along the y axis
    m1 = Eigen::Matrix4f::Identity();
    ry = ry / 180.0 * M_PI;
    m1(0, 0) = cos(ry);
    m1(0, 2) = sin(ry);
    m1(2, 0) = -sin(ry);
    m1(2, 2) = cos(ry);
    //cout << "m1" << endl << m1 << endl;
    m = m * m1;   //apply the rotation along y axis
    //cout << "m" << endl << m << endl;
    //rotating the point cloud along the z axis
    m1 = Eigen::Matrix4f::Identity();
    rz = rz / 180.0 * M_PI;
    m1(0, 0) = cos(rz);
    m1(0, 1) = -sin(rz);
    m1(1, 0) = sin(rz);
    m1(1, 1) = cos(rz);
    //cout << "m1" << endl << m1 << endl;
    m = m * m1;   //apply the rotation along y axis
    //cout << "m" << endl << m << endl;

    return m;
}

//compute the error of rotation matrix and translation matrix
void computeMatrixError(float rotation, float translation, float
 & rotationError, float &translationError, Eigen::Matrix4f &real
RtMatrix)
{
    Eigen::Matrix4f idealMatrix = Eigen::Matrix4f::Identity();
    idealMatrix = createIdealTransformationMatrix(translation, t
ranslation, translation, rotation, rotation, rotation);
    //cout << "idealMatrix:"<<endl << idealMatrix << endl;


    Eigen::Matrix3f idealRotationMatrix = Eigen::Matrix3f::Ident
```

```
ity();
    Eigen::Vector3f iealTranslationVector = Eigen::Vector3f(0, 0
, 0);

    //getting ideal rotation matrix
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            idealRotationMatrix(i, j) = idealMatrix(i, j);
        }
    }

    //getting ideal translation matrix
    for (int i = 0; i < 3; i++)
    {
        iealTranslationVector(i) = idealMatrix(i, 3);
    }
    cout << "ideal Rotation matrix:" << endl << idealRotationMat
rix << endl;
    cout << "ideal Translatin vector:" << endl << iealTranslatio
nVector << endl;

    cout << "real Matrix:" << endl << realRtMatrix << endl;

    Eigen::Matrix3f realRotationMatrix = Eigen::Matrix3f::Identi
ty();
    Eigen::Vector3f realTranslationVector = Eigen::Vector3f(0, 0
, 0);

    //getting ideal rotation matrix
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            realRotationMatrix(i, j) = realRtMatrix(i, j);
        }
    }
    //getting real translation matrix
    for (int i = 0; i < 3; i++)
```

```
    {
        realTranslationVector(i) = realRtMatrix(i, 3);
    }

    cout << "real Rotation matrix:" << endl << realRotationMatri
x << endl;
    cout << "real Translatin vector:" << endl << realTranslation
Vector << endl;

    //compute rotation error
    Eigen::Matrix3f m = idealRotationMatrix * realRotationMatrix
.inverse();
    Eigen::SelfAdjointEigenSolver<Eigen::Matrix3f> es;
    es.compute(m);
    float tr = es.eigenvalues()(0) + es.eigenvalues()(1) + es.ei
genvalues()(2);
    if (tr > 3) tr = 3;
    if (tr < -3) tr = -3;
    rotationError = acos((tr - 1) / 2) * 180.0 / M_PI;

    //compute translation error
    translationError = (realTranslationVector - iealTranslationV
ector).norm();
}
```

- 剔除边缘点

boundary_points.h

```
#ifndef  _BOUNDARY_POINTS_
#define _BOUNDARY_POINTS_

#include <pcl/search/kdtree.h>
#include <pcl/features/normal_3d.h>
#include <pcl/features/boundary.h>

void computeBoundaryPoints(pcl::PointCloud<pcl::PointXYZ>::Ptr &
cloud,
    double resolution,
    pcl::PointCloud<pcl::PointXYZ>::Ptr &boundary_cloud)
```

```
{
    // compute normals;
    pcl::search::Search<pcl::PointXYZ>::Ptr tree(new pcl::search
::KdTree<pcl::PointXYZ>());
    pcl::PointCloud<pcl::Normal>::Ptr normals(new pcl::PointClou
d<pcl::Normal>);
    pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> normal_est
;
    normal_est.setSearchMethod(tree);
    normal_est.setInputCloud(cloud);
    normal_est.setKSearch(50);
    normal_est.compute(*normals);
    //normal_est.setViewPoint(0,0,0);


    //calculate boundary;
    pcl::PointCloud<pcl::Boundary> boundary;
    pcl::BoundaryEstimation<pcl::PointXYZ, pcl::Normal, pcl::Bou
ndary> boundary_est;
    boundary_est.setInputCloud(cloud);
    boundary_est.setInputNormals(normals);
    boundary_est.setRadiusSearch(5 * resolution);
    boundary_est.setAngleThreshold(M_PI / 4);
    boundary_est.setSearchMethod(pcl::search::KdTree<pcl::PointX
YZ>::Ptr(new pcl::search::KdTree<pcl::PointXYZ>));
    boundary_est.compute(boundary);

    for (size_t i = 0; i < cloud->points.size(); ++i)
    {
        if (boundary.points[i].boundary_point == 1)
            boundary_cloud->push_back(cloud->points[i]);
    }
    std::cout << "boundary size is " << boundary_cloud->points.s
ize() << std::endl;
}

void eliminateBoundary(pcl::PointCloud<pcl::PointXYZ>::Ptr &clou
d,
    pcl::PointCloud<pcl::PointXYZ>::Ptr &keys,
    double resolution, int rate)
```

```
{
    pcl::PointCloud<pcl::PointXYZ>::Ptr boundary_cloud(new pcl::
PointCloud<pcl::PointXYZ>);
    computeBoundaryPoints(cloud, resolution, boundary_cloud);
    pcl::search::KdTree<pcl::PointXYZ> kdtree;
    kdtree.setInputCloud(boundary_cloud);

    std::vector<int> indices;
    std::vector<float> distances;
    float diff = rate * rate *resolution*resolution;
    pcl::PointCloud<pcl::PointXYZ>::Ptr keys_result(new pcl::Poi
ntCloud<pcl::PointXYZ>);
    for (size_t i = 0; i < keys->points.size(); ++i)
    {
        kdtree.nearestKSearch(keys->points[i], 1, indices, dista
nces);
        if (distances[0] > diff)
            keys_result->push_back(keys->points[i]);
    }

    std::cout << "remove " << keys->points.size() - keys_result-
>points.size() << "points" << std::endl;

    keys->clear();
    keys = keys_result;
}

#endif // ! _BOUNDARY_POINTS_
```

- 将点旋转平移

tranform.h

```
#pragma once
#include <pcl/common/transforms.h>

void MyTransformationPoint(pcl::PointXYZ &pt_in, pcl::PointXYZ &
pt_out, float tx, float ty, float tz, float rx, float ry, float
rz)
{
```

```
    ////rotating and translating point cloud
    ////m and m1 are both the unit matrixes
    Eigen::Matrix4f m = Eigen::Matrix4f::Identity();
    Eigen::Matrix4f m1 = Eigen::Matrix4f::Identity();
    //tx,ty,tz are translation values of x ,y and z axis
    m1(0, 3) = tx;
    m1(1, 3) = ty;
    m1(2, 3) = tz;
    //cout << "m1" << endl << m1 << endl;
    m = m * m1;   //apply the translation
    //cout << "m" << endl << m << endl;
    //rotating the point cloud along the x axis
    m1 = Eigen::Matrix4f::Identity();
    rx = rx / 180.0 * M_PI;
    m1(1, 1) = cos(rx);
    m1(1, 2) = -sin(rx);
    m1(2, 1) = sin(rx);
    m1(2, 2) = cos(rx);
    //cout << "m1" << endl << m1 << endl;
    m = m * m1;   //apply the rotation along x axis
    //cout << "m" << endl << m << endl;
    //rotating the point cloud along the y axis
    m1 = Eigen::Matrix4f::Identity();
    ry = ry / 180.0 * M_PI;
    m1(0, 0) = cos(ry);
    m1(0, 2) = sin(ry);
    m1(2, 0) = -sin(ry);
    m1(2, 2) = cos(ry);
    //cout << "m1" << endl << m1 << endl;
    m = m * m1;   //apply the rotation along y axis
    //cout << "m" << endl << m << endl;
    //rotating the point cloud along the z axis
    m1 = Eigen::Matrix4f::Identity();
    rz = rz / 180.0 * M_PI;
    m1(0, 0) = cos(rz);
    m1(0, 1) = -sin(rz);
    m1(1, 0) = sin(rz);
    m1(1, 1) = cos(rz);
    //cout << "m1" << endl << m1 << endl;
    m = m * m1;   //apply the rotation along y axis
```

```
    //cout << "m" << endl << m << endl;

    //pcl::transformPointCloud(src, dst, m);
    Eigen::Matrix<float, 3, 1> pt(pt_in.x, pt_in.y, pt_in.z);
    pt_out.x = static_cast<float> (m(0, 0) * pt.coeffRef(0) + m(
0, 1) * pt.coeffRef(1) + m(0, 2) * pt.coeffRef(2) + m(0, 3));
    pt_out.y = static_cast<float> (m(1, 0) * pt.coeffRef(0) + m(
1, 1) * pt.coeffRef(1) + m(1, 2) * pt.coeffRef(2) + m(1, 3));
    pt_out.z = static_cast<float> (m(2, 0) * pt.coeffRef(0) + m(
2, 1) * pt.coeffRef(1) + m(2, 2) * pt.coeffRef(2) + m(2, 3));
}
```

常用函数(3)

这里给出两种计算局部坐标系(LRF)的方式。

- **BOARDLocalReferenceFrame**

参考文献： A. Petrelli, L. Di Stefano, "On the repeatability of the local reference frame for partial shape matching", 13th International Conference on Computer Vision (ICCV), 2011

```
#include <pcl/io/pcd_io.h>
#include <pcl/features/normal_3d.h>
#include <pcl/impl/point_types.hpp>
#include <pcl/features/board.h>
#include <pcl/search/kdtree.h>

int main()
{
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointClou
d<pcl::PointXYZ>);
    pcl::io::loadPCDFile("rabbit.pcd", *cloud);
    std::cout << "load " << cloud->points.size() << std::endl;

    pcl::PointCloud<pcl::Normal>::Ptr normals(new pcl::PointClou
d<pcl::Normal>);

    pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> ne;
    ne.setInputCloud(cloud);
    ne.setKSearch(10);
    ne.setSearchMethod(pcl::search::KdTree<pcl::PointXYZ>::Ptr(n
ew pcl::search::KdTree<pcl::PointXYZ>));
    ne.compute(*normals);
    std::cout << normals->points.size();

    pcl::BOARDLocalReferenceFrameEstimation<pcl::PointXYZ, pcl::
Normal, pcl::ReferenceFrame> rftest;
    rftest.setInputCloud(cloud);
    rftest.setInputNormals(normals);
```

```
    rftest.setSearchSurface(cloud);
    rftest.setRadiusSearch(0.01);
    rftest.setFindHoles(false);

    pcl::PointCloud<pcl::ReferenceFrame>::Ptr rf(new pcl::PointC
loud<pcl::ReferenceFrame>);
    rftest.compute(*rf);

    system("pause");
    return 0;
}
```

- **SHOTLocalReferenceFrame**

参考文献：shot描述子

```
#include <pcl/io/pcd_io.h>
#include <pcl/features/shot_lrf.h>

int main()
{
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointClou
d<pcl::PointXYZ>);
    pcl::io::loadPCDFile("rabbit.pcd", *cloud);
    std::cout << "load " << cloud->points.size() << std::endl;

    pcl::SHOTLocalReferenceFrameEstimation<pcl::PointXYZ, pcl::R
eferenceFrame>::Ptr lrf_estimator(new pcl::SHOTLocalReferenceFra
meEstimation<pcl::PointXYZ, pcl::ReferenceFrame>);
    lrf_estimator->setRadiusSearch(0.001);
    lrf_estimator->setInputCloud(cloud);
    lrf_estimator->setSearchSurface(cloud);
    pcl::PointCloud<pcl::ReferenceFrame>::Ptr cloud_lrf(new pcl:
:PointCloud<pcl::ReferenceFrame>);
    lrf_estimator->compute(*cloud_lrf);
    std::cout << "compute lrf size: " << cloud_lrf->size() << st
d::endl;

    system("pause");
    return 0;
}
```

Eigen函数表

转自http://www.cnblogs.com/python27/p/EigenQuickRef.html

- 矩阵定义

```
#include <Eigen/Dense>

Matrix<double, 3, 3> A;                 // Fixed rows and cols. Sa
me as Matrix3d.
Matrix<double, 3, Dynamic> B;           // Fixed rows, dynamic col
s.
Matrix<double, Dynamic, Dynamic> C;     // Full dynamic. Same as M
atrixXd.
Matrix<double, 3, 3, RowMajor> E;       // Row major; default is c
olumn-major.
Matrix3f P, Q, R;                       // 3x3 float matrix.
Vector3f x, y, z;                       // 3x1 float matrix.
RowVector3f a, b, c;                    // 1x3 float matrix.
VectorXd v;                             // Dynamic column vector o
f doubles
// Eigen           // Matlab           // comments
x.size()           // length(x)        // vector size
C.rows()           // size(C,1)        // number of rows
C.cols()           // size(C,2)        // number of columns
x(i)               // x(i+1)           // Matlab is 1-based
C(i,j)             // C(i+1,j+1)       //
```

- **Eigen** 基础使用

```
// Basic usage
// Eigen        // Matlab           // comments
x.size()        // length(x)        // vector size
C.rows()        // size(C,1)        // number of rows
C.cols()        // size(C,2)        // number of columns
x(i)            // x(i+1)           // Matlab is 1-based
C(i, j)         // C(i+1,j+1)       //

A.resize(4, 4);   // Runtime error if assertions are on.
B.resize(4, 9);   // Runtime error if assertions are on.
A.resize(3, 3);   // Ok; size didn't change.
B.resize(3, 9);   // Ok; only dynamic cols changed.

A << 1, 2, 3,     // Initialize A. The elements can also be
     4, 5, 6,     // matrices, which are stacked along cols
     7, 8, 9;     // and then the rows are stacked.
B << A, A, A;     // B is three horizontally stacked A's.
A.fill(10);       // Fill A with all 10's.
```

- **Eigen** 特殊矩阵生成

```
// Eigen                            // Matlab
MatrixXd::Identity(rows,cols)       // eye(rows,cols)
C.setIdentity(rows,cols)            // C = eye(rows,cols)
MatrixXd::Zero(rows,cols)           // zeros(rows,cols)
C.setZero(rows,cols)                // C = ones(rows,cols)
MatrixXd::Ones(rows,cols)           // ones(rows,cols)
C.setOnes(rows,cols)                // C = ones(rows,cols)
MatrixXd::Random(rows,cols)         // rand(rows,cols)*2-1
  // MatrixXd::Random returns uniform random numbers in (-1, 1).
C.setRandom(rows,cols)              // C = rand(rows,cols)*2-1
VectorXd::LinSpaced(size,low,high)  // linspace(low,high,size)'
v.setLinSpaced(size,low,high)       // v = linspace(low,high,siz
e)'
```

- **Eigen** 矩阵分块

```
// Matrix slicing and blocks. All expressions listed here are re
```

```
ad/write.
// Templated size versions are faster. Note that Matlab is 1-bas
ed (a size N
// vector is x(1)...x(N)).
// Eigen                           // Matlab
x.head(n)                          // x(1:n)
x.head<n>()                        // x(1:n)
x.tail(n)                          // x(end - n + 1: end)
x.tail<n>()                        // x(end - n + 1: end)
x.segment(i, n)                    // x(i+1 : i+n)
x.segment<n>(i)                    // x(i+1 : i+n)
P.block(i, j, rows, cols)          // P(i+1 : i+rows, j+1 : j+co
ls)
P.block<rows, cols>(i, j)          // P(i+1 : i+rows, j+1 : j+co
ls)
P.row(i)                           // P(i+1, :)
P.col(j)                           // P(:, j+1)
P.leftCols<cols>()                 // P(:, 1:cols)
P.leftCols(cols)                   // P(:, 1:cols)
P.middleCols<cols>(j)              // P(:, j+1:j+cols)
P.middleCols(j, cols)              // P(:, j+1:j+cols)
P.rightCols<cols>()                // P(:, end-cols+1:end)
P.rightCols(cols)                  // P(:, end-cols+1:end)
P.topRows<rows>()                  // P(1:rows, :)
P.topRows(rows)                    // P(1:rows, :)
P.middleRows<rows>(i)              // P(i+1:i+rows, :)
P.middleRows(i, rows)              // P(i+1:i+rows, :)
P.bottomRows<rows>()               // P(end-rows+1:end, :)
P.bottomRows(rows)                 // P(end-rows+1:end, :)
P.topLeftCorner(rows, cols)        // P(1:rows, 1:cols)
P.topRightCorner(rows, cols)       // P(1:rows, end-cols+1:end)
P.bottomLeftCorner(rows, cols)     // P(end-rows+1:end, 1:cols)
P.bottomRightCorner(rows, cols)    // P(end-rows+1:end, end-cols
+1:end)
P.topLeftCorner<rows,cols>()       // P(1:rows, 1:cols)
P.topRightCorner<rows,cols>()      // P(1:rows, end-cols+1:end)
P.bottomLeftCorner<rows,cols>()    // P(end-rows+1:end, 1:cols)
P.bottomRightCorner<rows,cols>()   // P(end-rows+1:end, end-cols
+1:end)
```

- **Eigen** 矩阵元素交换

```
// Of particular note is Eigen's swap function which is highly o
ptimized.
// Eigen                           // Matlab
R.row(i) = P.col(j);               // R(i, :) = P(:, i)
R.col(j1).swap(mat1.col(j2));      // R(:, [j1 j2]) = R(:, [j2,
j1])
```

- **Eigen** 矩阵转置

```
// Views, transpose, etc; all read-write except for .adjoint().
// Eigen                           // Matlab
R.adjoint()                        // R'
R.transpose()                      // R.' or conj(R')
R.diagonal()                       // diag(R)
x.asDiagonal()                     // diag(x)
R.transpose().colwise().reverse(); // rot90(R)
R.conjugate()                      // conj(R)
```

- **Eigen** 矩阵乘积

```
// All the same as Matlab, but matlab doesn't have *= style oper
ators.
// Matrix-vector.  Matrix-matrix.   Matrix-scalar.
y  = M*x;          R  = P*Q;        R  = P*s;
a  = b*M;          R  = P - Q;      R  = s*P;
a *= M;            R  = P + Q;      R  = P/s;
                   R *= Q;          R  = s*P;
                   R += Q;          R *= s;
                   R -= Q;          R /= s;
```

- **Eigen** 矩阵单个元素操作

```
// Vectorized operations on each element independently
// Eigen                  // Matlab
R = P.cwiseProduct(Q);    // R = P .* Q
R = P.array() * s.array();// R = P .* s
R = P.cwiseQuotient(Q);   // R = P ./ Q
R = P.array() / Q.array();// R = P ./ Q
R = P.array() + s.array();// R = P + s
R = P.array() - s.array();// R = P - s
R.array() += s;           // R = R + s
R.array() -= s;           // R = R - s
R.array() < Q.array();    // R < Q
R.array() <= Q.array();   // R <= Q
R.cwiseInverse();         // 1 ./ P
R.array().inverse();      // 1 ./ P
R.array().sin()           // sin(P)
R.array().cos()           // cos(P)
R.array().pow(s)          // P .^ s
R.array().square()        // P .^ 2
R.array().cube()          // P .^ 3
R.cwiseSqrt()             // sqrt(P)
R.array().sqrt()          // sqrt(P)
R.array().exp()           // exp(P)
R.array().log()           // log(P)
R.cwiseMax(P)             // max(R, P)
R.array().max(P.array())  // max(R, P)
R.cwiseMin(P)             // min(R, P)
R.array().min(P.array())  // min(R, P)
R.cwiseAbs()              // abs(P)
R.array().abs()           // abs(P)
R.cwiseAbs2()             // abs(P.^2)
R.array().abs2()          // abs(P.^2)
(R.array() < s).select(P,Q);  // (R < s ? P : Q)
```

- **Eigen** 矩阵化简

```
// Reductions.
int r, c;
// Eigen                  // Matlab
R.minCoeff()              // min(R(:))
R.maxCoeff()              // max(R(:))
s = R.minCoeff(&r, &c)    // [s, i] = min(R(:)); [r, c] = ind2su
b(size(R), i);
s = R.maxCoeff(&r, &c)    // [s, i] = max(R(:)); [r, c] = ind2su
b(size(R), i);
R.sum()                   // sum(R(:))
R.colwise().sum()         // sum(R)
R.rowwise().sum()         // sum(R, 2) or sum(R')'
R.prod()                  // prod(R(:))
R.colwise().prod()        // prod(R)
R.rowwise().prod()        // prod(R, 2) or prod(R')'
R.trace()                 // trace(R)
R.all()                   // all(R(:))
R.colwise().all()         // all(R)
R.rowwise().all()         // all(R, 2)
R.any()                   // any(R(:))
R.colwise().any()         // any(R)
R.rowwise().any()         // any(R, 2)
```

- **Eigen** 矩阵点乘

```
// Dot products, norms, etc.
// Eigen                  // Matlab
x.norm()                  // norm(x).    Note that norm(R) doesn
't work in Eigen.
x.squaredNorm()           // dot(x, x)   Note the equivalence is
 not true for complex
x.dot(y)                  // dot(x, y)
x.cross(y)                // cross(x, y) Requires #include <Eige
n/Geometry>
```

- **Eigen** 矩阵类型转换

```
//// Type conversion
// Eigen                              // Matlab
A.cast<double>();                     // double(A)
A.cast<float>();                      // single(A)
A.cast<int>();                        // int32(A)
A.real();                             // real(A)
A.imag();                             // imag(A)
// if the original type equals destination type, no work is done
```

- **Eigen** 求解线性方程组 **Ax = b**

```
// Solve Ax = b. Result stored in x. Matlab: x = A \ b.
x = A.ldlt().solve(b));  // A sym. p.s.d.    #include <Eigen/Cho
lesky>
x = A.llt() .solve(b));  // A sym. p.d.      #include <Eigen/Cho
lesky>
x = A.lu()  .solve(b));  // Stable and fast. #include <Eigen/LU>
x = A.qr()  .solve(b));  // No pivoting.     #include <Eigen/QR>
x = A.svd() .solve(b));  // Stable, slowest. #include <Eigen/SVD
>
// .ldlt() -> .matrixL() and .matrixD()
// .llt()  -> .matrixL()
// .lu()   -> .matrixL() and .matrixU()
// .qr()   -> .matrixQ() and .matrixR()
// .svd()  -> .matrixU(), .singularValues(), and .matrixV()
```

- **Eigen** 矩阵特征值

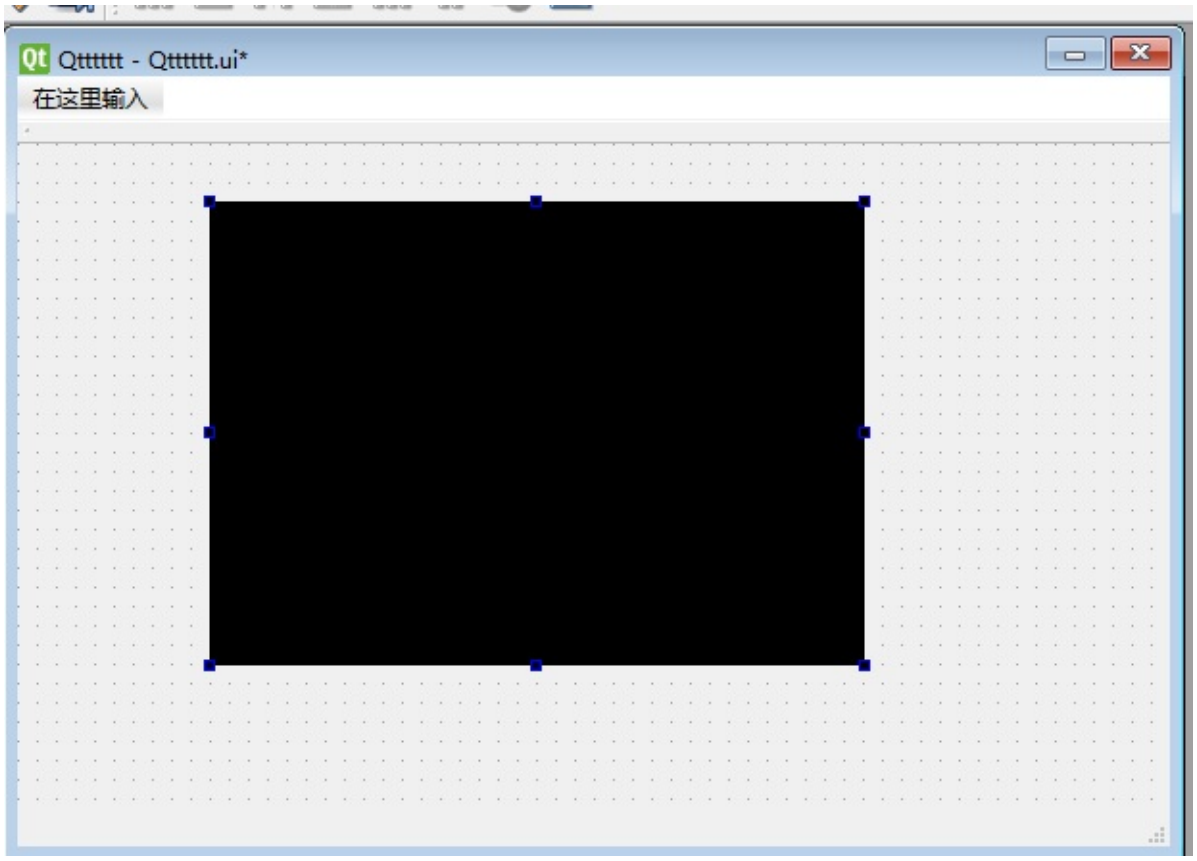
```
// Eigenvalue problems
// Eigen                            // Matlab
A.eigenvalues();                    // eig(A);
EigenSolver<Matrix3d> eig(A);       // [vec val] = eig(A)
eig.eigenvalues();                  // diag(val)
eig.eigenvectors();                 // vec
// For self-adjoint matrices use SelfAdjointEigenSolver<>
```

新建QT界面项目

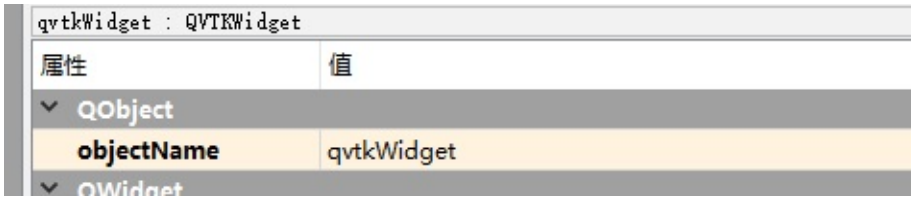
此处给出利用QT和VTK生成的插件，在QT界面中显示点云的例子。

- 1、新建QT GUI项目，原生QT或者VS插件建立都可以。(此处我的项目名字叫Qtttttt) 注意：VS添加“附加包含目录”、“附加库目录”、“附加依赖项”。QT则添加INCLUDEPATH、LIBS。
- 2、用Qt Designer打开.ui文件，左侧工具栏可以看到QVTK插件。

拖入GUI中，调整合适大小。



属性中可以看到默认名字为qvtkWidget



- 3、修改代码

修改Qtttttt.h文件，添加如下内容

```
#pragma once

#include <QtWidgets/QMainWindow>
#include "ui_Qtttttt.h"

// --------------- 添加 ------------
#include <vtkAutoInit.h>
VTK_MODULE_INIT(vtkRenderingOpenGL2);
VTK_MODULE_INIT(vtkInteractionStyle);

#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/visualization/pcl_visualizer.h>
// ----------------------------------

class Qtttttt : public QMainWindow
{
    Q_OBJECT

public:
    Qtttttt(QWidget *parent = Q_NULLPTR);

    // -------------- 添加----------------
    //点云数据存储
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud;
    boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer;

    //初始化vtk部件
    void initialVtkWidget();

    private slots:
    //创建打开槽
    void onOpen();
    // -------------------------------=

private:
    Ui::QtttttClass ui;
};
```

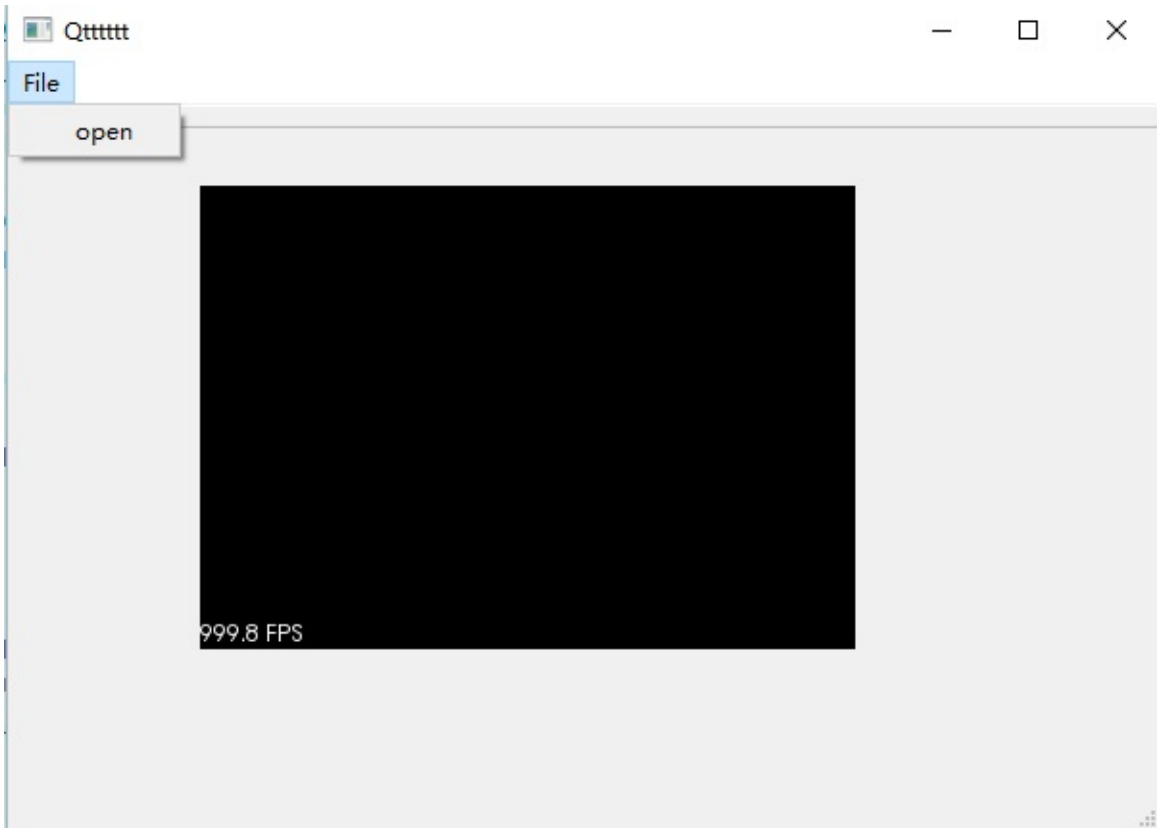
修改Qttttt.p文件，添加如下内容

```
#include "Qttttt.h"
// ------------------ 添加-------------
#include <QMenu>
#include <QFileDialog>
#include <iostream>
#include <vtkRenderWindow.h>
// -------------------------------------

Qttttt::Qttttt(QWidget *parent)
    : QMainWindow(parent)
{
    ui.setupUi(this);


    // ------------- 添加 ----------------------
    QMenu *littleTools;
    QAction *openAction;

    littleTools = new QMenu("File");
    ui.menuBar->addMenu(littleTools);
    openAction = new QAction("open");
    littleTools->addAction(openAction);

    //初始化vtk
    initialVtkWidget();
    //连接信号和槽
    connect(openAction, SIGNAL(triggered()), this, SLOT(onOpen()
));
    // --------------------------------------------
}

// ----------------- 添加-----------------
void Qttttt::initialVtkWidget()
{
    cloud.reset(new pcl::PointCloud<pcl::PointXYZ>);
    viewer.reset(new pcl::visualization::PCLVisualizer("viewer",
 false));
    viewer->addPointCloud(cloud, "cloud");
```

```
    ui.qvtkWidget->SetRenderWindow(viewer->getRenderWindow());
    viewer->setupInteractor(ui.qvtkWidget->GetInteractor(), ui.q
vtkWidget->GetRenderWindow());
    ui.qvtkWidget->update();
}

//读取文本型和二进制型点云数据
void Qtttttt::onOpen()
{
    //只能打开PCD文件
    QString fileName = QFileDialog::getOpenFileName(this,
        tr("Open PointCloud"), ".",
        tr("Open PCD files(*.pcd)"));

    if (!fileName.isEmpty())
    {
        std::string file_name = fileName.toStdString();
        //sensor_msgs::PointCloud2 cloud2;
        pcl::PCLPointCloud2 cloud2;
        //pcl::PointCloud<Eigen::MatrixXf> cloud2;
        Eigen::Vector4f origin;
        Eigen::Quaternionf orientation;
        int pcd_version;
        int data_type;
        unsigned int data_idx;
        int offset = 0;
        pcl::PCDReader rd;
        rd.readHeader(file_name, cloud2, origin, orientation, pc
d_version, data_type, data_idx);

        if (data_type == 0)
        {
            pcl::io::loadPCDFile(fileName.toStdString(), *cloud)
;
        }
        else if (data_type == 2)
        {
            pcl::PCDReader reader;
            reader.read<pcl::PointXYZ>(fileName.toStdString(), *
```

```
cloud);
        }

        viewer->updatePointCloud(cloud, "cloud");
        viewer->resetCamera();
        ui.qvtkWidget->update();
    }
}
//-------------------------------
```

- 4、main.cpp 不用修改，直接可以运行。程序如下：



点击open弹出文件选择框，可以将选择的pcd文件显示出来，由于路径没做处理，中文路径可能会出问题。

- 注意：

1、运行vtk出现错误： 错误：Error:no override found for ‘vtkRenderWindow’. 解决：在第一次使用vtk的头文件最前添加下面代码。

```
#include <vtkAutoInit.h>
VTK_MODULE_INIT(vtkRenderingOpenGL2); // 此处由于我编译qtvtk时选择的
OpenGL2
VTK_MODULE_INIT(vtkInteractionStyle);
```

2、加入pcl出现错误：错误： `error C2440: “static_cast”: 无法从“vtkObjectBase *const ”转换为“vtkRenderWindow ”` 解决： `在Qtttttt.cpp文件里添加#include <vtkRenderWindow.h>`

- 参考文章

http://blog.csdn.net/wokaowokaowokao12345/article/details/51314439

http://blog.csdn.net/wokaowokaowokao12345/article/details/51096887

http://blog.csdn.net/wokaowokaowokao12345/article/details/51078495

常用模型库

- https://www.cc.gatech.edu/~turk/bunny/bunny.html
- http://www.cc.gatech.edu/projects/large_models/
- http://graphics.stanford.edu/data/3Dscanrep/
- http://www.cc.gatech.edu/~turk/zipper/zipper.html
- http://staffhome.ecm.uwa.edu.au/~00053650/recognition.html
- http://gfx.cs.princeton.edu/proj/sugcon/models/
- http://rgbd-dataset.cs.washington.edu/dataset/
- http://pointclouds.org/media/

- api文档：http://docs.pointclouds.org/trunk/index.html/
- 英文教程：http://pointclouds.org/documentation/tutorials/
- 英文论坛：http://www.pcl-users.org/
- 中文论坛：http://www.pclcn.org/bbs/forum.php
- 各种描述子：http://robotica.unileon.es/index.php/PCL/OpenNI_tutorial_4:_3D_object_recognition_(descriptors)/

- http://lgg.epfl.ch/index.php

- http://graphics.cs.aueb.gr/graphics/index.html