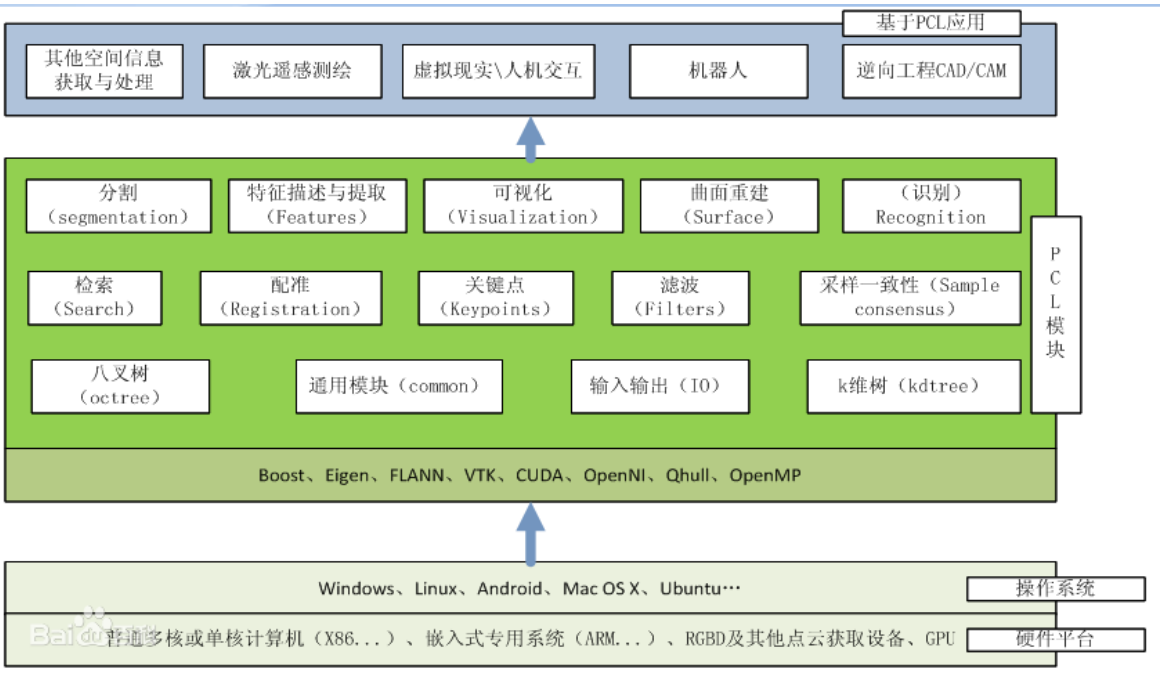
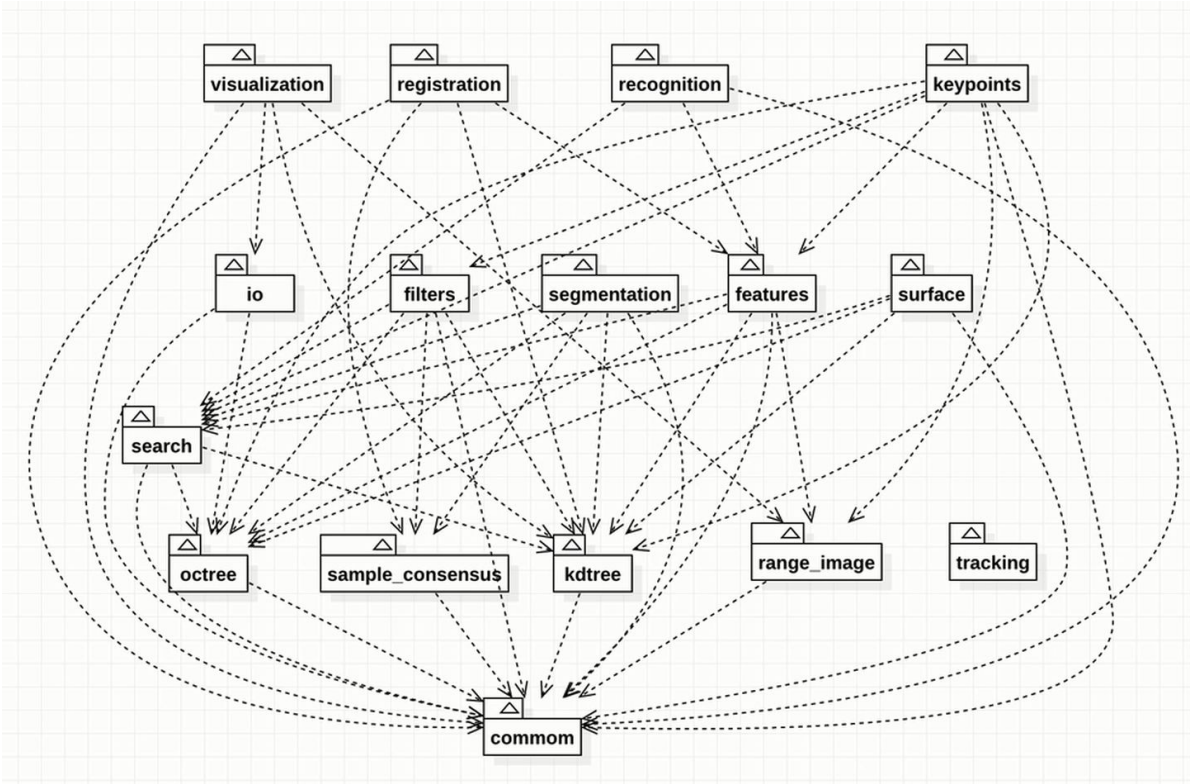
PCL：可以免费进行商业和学术应用

- 是用于2D / 3D图像和点云处理的大型开源跨平台的C++编程库。 PCL框架实现了大量点云相关的通用算法和高效的数据结构。内容涉及了点云的获取、滤波、分割、配准、检索、特征提取、特征估计，表面重建、识别、模型拟合、追踪、曲面重建、可视化等等，这些算法可用于从嘈杂的数据中过滤出异常值，将3D点云缝合在一起，分割场景的部分区域，提取关键点并计算描述符，根据物体的几何外观识别实际物体，创建表面点云并将其可视化。支持多种操作系统，可以在Windows、Linux、MacOS X、Android、部分嵌入式实时系统上运行。





PCL的起源与发展

- PCL 起初是 ROS(Robot Operating System)下由来自斯坦福大学的年轻博士Radu等人维护和开发的开源项目。主要应用于机器人研究应用领域，随着各个算法模块的积累，于 2011 年独立出来，正式与全球 3D信息获取处理的同行一起，组建了强大的开发维护团队，以多所知名大学、研究所和相关硬件、软件公司为主。

数据结构

- 根据激光测量原理得到的点云，包含三维坐标信息(xyz)和激光反射强度信息（intensity），激光反射强度与仪器的激光发射能量、波长，目标的表面材质、粗糙程度、入射角相关。根据摄影测量原理得到的点云，包括三维坐标（xyz）和颜色信息（rgb）。结合两个原理的多传感器融合技术（多见于手持式三维扫描仪），能够同时得到这三种信息。

基本类型PointCloud

PCL的基本数据类型是`PointCloud`，一个`PointCloud`是一个C++的模板类，它包含了以下字段：

- `width(int)`：指定点云数据集的宽度
  - 对于无组织格式的数据集，width代表了所有点的总数
  - 对于有组织格式的数据集，width代表了一行中的总点数
- `height(int)`：制定点云数据集的高度
  - 对于无组织格式的数据集，值为1
  - 对于有组织格式的数据集，表示总行数
  - 判断数据集是否有组织的方式pcl ::PointCloud<pcl::isOrganized>
  - ```
    if (!cloud.isOrganized ())//数据集有组织则情形很像图片的长和高，这样点与点之间的关系就很明确
    ```
- `points(std::vector<PointT>)`：包含所有PointT类型的点的数据列表
    ```

点的类型：存储在point_types.hpp中

- - PointXYZ
  - PointXYZI
  - PointXYZRGBA
  - PointXYZRGB
  - PointXY
  - InterestPoint
  - Normal
  - PointNormal
  - PointXYZRGBNormal
  - PointXYZINormal
- - PointWithRange
  - PointWithViewpoint
  - MomentInvariants
  - PrincipalRadiiRSD
  - Boundary
  - PrincipalCurvatures
  - BounPFHSignature125dary
  - FPFHSignature33
  - VFHSignature308
  - Narf36
  - BorderDescription
  - IntensityGradient
  - Histogram
  - PointWithScale
  - PointSurfel

如何在自己的项目中使用PCL

在写好的cpp文件目录下写CMakeLists.txt,

```
cmake_minimum_required(VERSION 2.6 FATAL_ERROR)
project(MY_GRAND_PROJECT)
find_package(PCL 1.3 REQUIRED COMPONENTS common io)
include_directories(${PCL_INCLUDE_DIRS})
link_directories(${PCL_LIBRARY_DIRS})
add_definitions(${PCL_DEFINITIONS})#PCL_DEFINITIONS：列出所需的预处理器定义和编译器标志
add_executable(pcd_write_test pcd_write.cpp)
target_link_libraries(pcd_write_test ${PCL_LIBRARIES})
```

点云的矩阵变换（pcl_transform.cpp）

4x4矩阵转换点云：

在线性代数，一个**旋转矩阵**是变换矩阵，用于执行旋转的欧氏空间。例如，使用下面的约定，矩阵

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

将xy平面中的点相对于x轴绕二维笛卡尔坐标系的原点逆时针旋转角度θ。要在具有标准坐标$\mathbf{v} = (x, y)$的平面点上执行旋转，应将其写为列向量，并乘以矩阵R：

$$R\mathbf{v} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}.$$

在这里，我们定义了围绕 Z 轴的 45° (PI/4) 旋转和 X 轴上的平移。这是我们刚刚定义的变换

```
    |  cos(θ) -sin(θ)  0.0 |
R = |  sin(θ)  cos(θ)  0.0 |
    |  0.0     0.0     1.0 |

t = < 2.5, 0.0, 0.0 >
```

有两种方式进行4x4矩阵点云变换：

方式1：

```
Eigen::Matrix4f transform_1 = Eigen::Matrix4f::Identity();
float theta = M_PI/4; // The angle of rotation in radians
  transform_1 (0,0) = std::cos (theta);
  transform_1 (0,1) = -sin(theta);
  transform_1 (1,0) = sin (theta);
  transform_1 (1,1) = std::cos (theta);
  transform_1 (0,3) = 2.5;
```

方式2：

```
  /*  METHOD #2: Using a Affine3f
    This method is easier and less error prone
  */
  Eigen::Affine3f transform_2 = Eigen::Affine3f::Identity();

  // Define a translation of 2.5 meters on the x axis.
  transform_2.translation() << 2.5, 0.0, 0.0;

  // The same rotation matrix as before; theta radians around Z axis
  transform_2.rotate (Eigen::AngleAxisf (theta, Eigen::Vector3f::UnitZ()));

  // Print the transformation
  printf ("\nMethod #2: using an Affine3f\n");
  std::cout << transform_2.matrix() << std::endl;
```

```
#include <iostream>

#include <pcl/io/pcd_io.h>
#include <pcl/io/ply_io.h>
#include <pcl/point_cloud.h>
#include <pcl/console/parse.h>
#include <pcl/common/transforms.h>
#include <pcl/visualization/pcl_visualizer.h>

// This function displays the help
void
showHelp(char * program_name)
{
  std::cout << std::endl;
  std::cout << "Usage: " << program_name << " cloud_filename.[pcd|ply]" <<
std::endl;
  std::cout << "-h:  Show this help." << std::endl;
}
```

```
// This is the main function
int
main (int argc, char** argv)
{

  // Show help
  if (pcl::console::find_switch (argc, argv, "-h") || pcl::console::find_switch
(argc, argv, "--help")) {
    showHelp (argv[0]);
    return 0;
  }

  // Fetch point cloud filename in arguments | Works with PCD and PLY files
  std::vector<int> filenames;
  bool file_is_pcd = false;

  filenames = pcl::console::parse_file_extension_argument (argc, argv, ".ply");

  if (filenames.size () != 1)  {
    filenames = pcl::console::parse_file_extension_argument (argc, argv,
".pcd");

    if (filenames.size () != 1) {
      showHelp (argv[0]);
      return -1;
    } else {
      file_is_pcd = true;
    }
  }

  // Load file | Works with PCD and PLY files
  pcl::PointCloud<pcl::PointXYZ>::Ptr source_cloud (new
pcl::PointCloud<pcl::PointXYZ> ());

  if (file_is_pcd) {
    if (pcl::io::loadPCDFile (argv[filenames[0]], *source_cloud) < 0)  {
      std::cout << "Error loading point cloud " << argv[filenames[0]] <<
std::endl << std::endl;
      showHelp (argv[0]);
      return -1;
    }
  } else {
    if (pcl::io::loadPLYFile (argv[filenames[0]], *source_cloud) < 0)  {
      std::cout << "Error loading point cloud " << argv[filenames[0]] <<
std::endl << std::endl;
      showHelp (argv[0]);
      return -1;
    }
  }

  /* Reminder: how transformation matrices work :

           |-------> This column is the translation
    | 1 0 0 x |  \
    | 0 1 0 y |   }-> The identity 3x3 matrix (no rotation) on the left
    | 0 0 1 z |  /
    | 0 0 0 1 |    -> We do not use this line (and it has to stay 0,0,0,1)
```

```
    METHOD #1: Using a Matrix4f
    This is the "manual" method, perfect to understand but error prone !
  */
  Eigen::Matrix4f transform_1 = Eigen::Matrix4f::Identity();

  // Define a rotation matrix (see
https://en.wikipedia.org/wiki/Rotation_matrix)
  float theta = M_PI/4; // The angle of rotation in radians
  transform_1 (0,0) = std::cos (theta);
  transform_1 (0,1) = -sin(theta);
  transform_1 (1,0) = sin (theta);
  transform_1 (1,1) = std::cos (theta);
  //    (row, column)

  // Define a translation of 2.5 meters on the x axis.
  transform_1 (0,3) = 2.5;

  // Print the transformation
  printf ("Method #1: using a Matrix4f\n");
  std::cout << transform_1 << std::endl;

  /*  METHOD #2: Using a Affine3f
    This method is easier and less error prone
  */
  Eigen::Affine3f transform_2 = Eigen::Affine3f::Identity();

  // Define a translation of 2.5 meters on the x axis.
  transform_2.translation() << 2.5, 0.0, 0.0;

  // The same rotation matrix as before; theta radians around Z axis
  transform_2.rotate (Eigen::AngleAxisf (theta, Eigen::Vector3f::UnitZ()));

  // Print the transformation
  printf ("\nMethod #2: using an Affine3f\n");
  std::cout << transform_2.matrix() << std::endl;

  // Executing the transformation
  pcl::PointCloud<pcl::PointXYZ>::Ptr transformed_cloud (new
pcl::PointCloud<pcl::PointXYZ> ());
  // You can either apply transform_1 or transform_2; they are the same
  pcl::transformPointCloud (*source_cloud, *transformed_cloud, transform_2);

  // Visualization
  printf(  "\nPoint cloud colors :  white  = original point cloud\n"
      "                        red  = transformed point cloud\n");
  pcl::visualization::PCLVisualizer viewer ("Matrix transformation example");

   // Define R,G,B colors for the point cloud
  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ>
source_cloud_color_handler (source_cloud, 255, 255, 255);
  // We add the point cloud to the viewer and pass the color handler
  viewer.addPointCloud (source_cloud, source_cloud_color_handler,
"original_cloud");

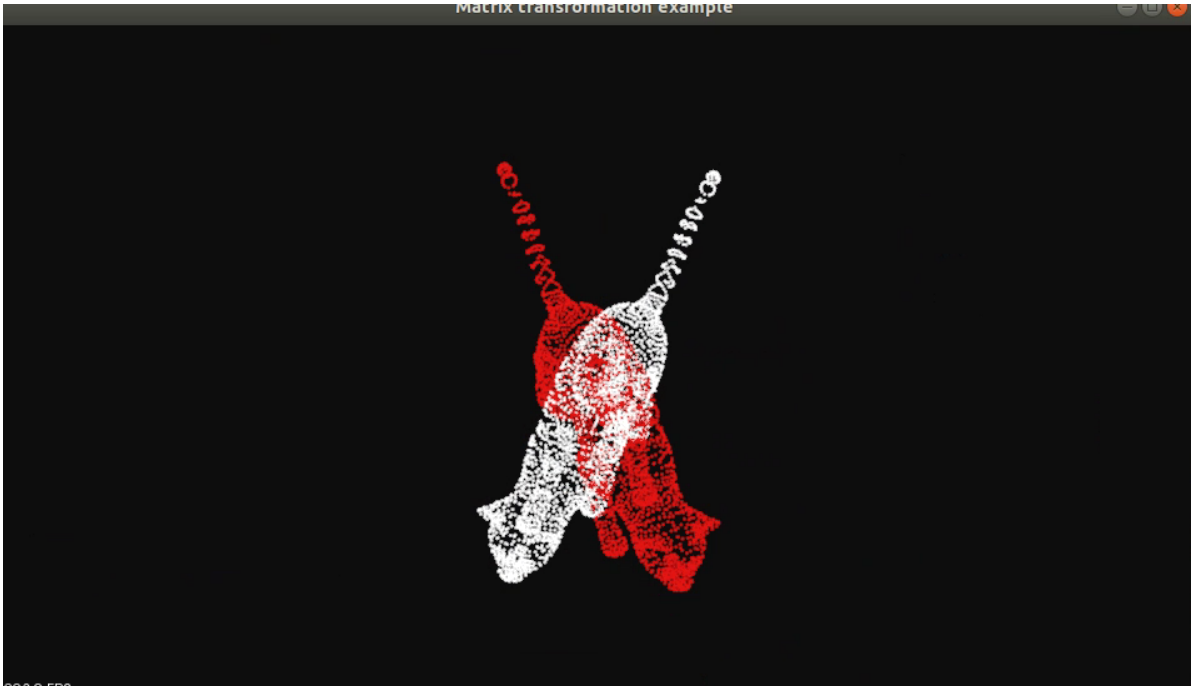
  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ>
transformed_cloud_color_handler (transformed_cloud, 230, 20, 20); // Red
```

```
  viewer.addPointCloud (transformed_cloud, transformed_cloud_color_handler,
"transformed_cloud");

  viewer.addCoordinateSystem (1.0, "cloud", 0);
  viewer.setBackgroundColor(0.05, 0.05, 0.05, 0); // Setting background to a
dark grey
  viewer.setPointCloudRenderingProperties
(pcl::visualization::PCL_VISUALIZER_POINT_SIZE, 2, "original_cloud");
  viewer.setPointCloudRenderingProperties
(pcl::visualization::PCL_VISUALIZER_POINT_SIZE, 2, "transformed_cloud");
  //viewer.setPosition(800, 400); // Setting visualiser window position

  while (!viewer.wasStopped ()) { // Display the visualiser until 'q' key is
pressed
    viewer.spinOnce ();
  }

  return 0;
}
```



可视化工具

pcl_viewer工具：

- **基本使用**

  进入：`pcl_viewer xxxxx.pcd`

  帮助：在界面中输入h，可以在控制台看到帮助信息

  退出：界面中输入q

  放大缩小：鼠标滚轮 或 Alt + [+/-]

  平移：Shift+鼠标拖拽

  旋转：Ctrl+鼠标拖拽

- **其他技巧**

  修改点颜色：数字1,2,3,4,5....9，重复按1可切换不同颜色方便观察

  放大缩小点：放大Ctrl+Shift+加号，缩小 Ctrl+减号

  保存截图：`j`

  显示颜色尺寸：`u`

  显示比例尺寸：`g`

  在控制列出所有几何和颜色信息：`l`

- **鼠标选点打印坐标**

  选点模式进入：`pcl_viewer -use_point_picking bunny.pcd`

  选择指定点：shift+鼠标左键

CloudViewer

基本上代码都在cloud_viewer.h中实现

```
void showCloud(const ColorCloud::ConstPtr &cloud,const std::string&
cloudname="cloud");//多个showCloud的重载函数（分别用于show不同类型的点）
void runOnVisualizationThread(vizCallable x,const std::string & key="callable");
而vizCallable是：
    typedef
boost::function1<void,pcl::visualization::PCLVisualizer&>VizCallable;

void runOnVisualizationThreadOnce(vizCallable x);
```

1.简单显示点云

使用如下代码就可以简单可视化点云文件

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

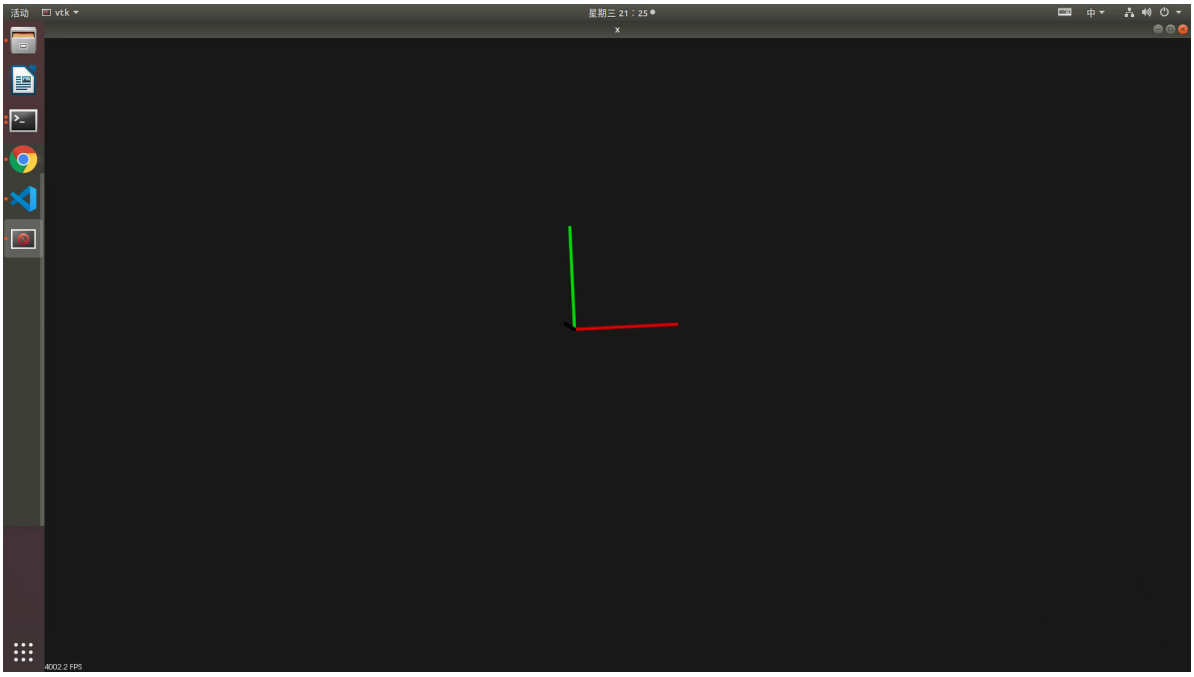
int main(int argc, char **argv) {

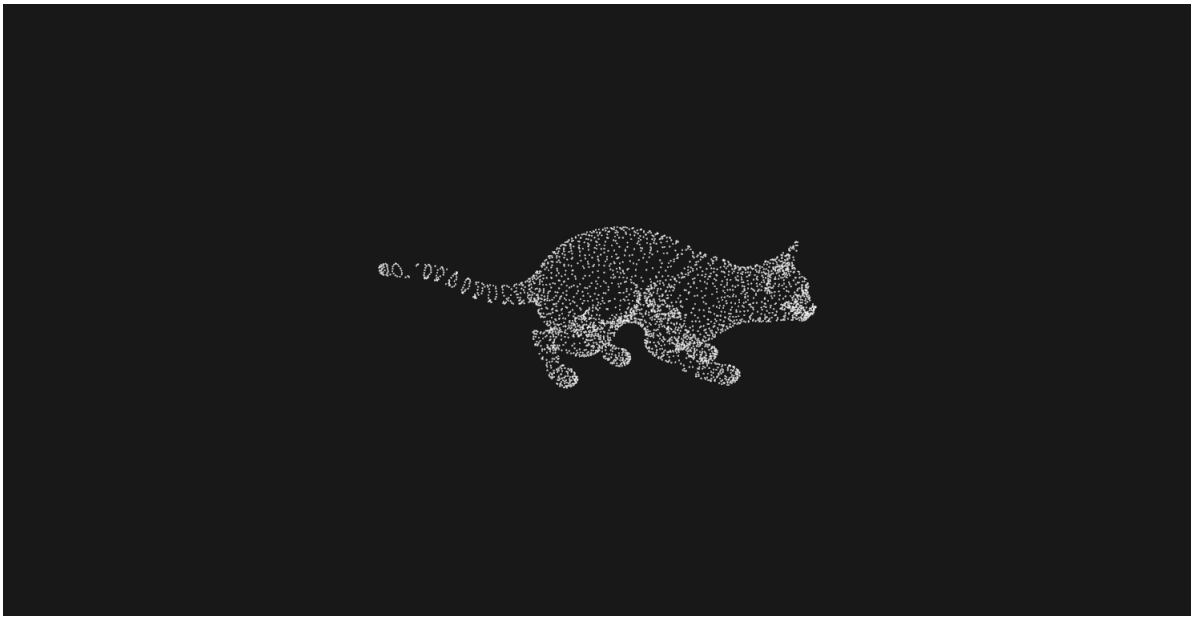
    // 创建PointCloud的智能指针
    pcl::PointCloud<pcl::PointXYZRGBA>::Ptr cloud(new
pcl::PointCloud<pcl::PointXYZRGBA>);
```

```
    // 加载pcd文件到cloud
    pcl::io::loadPCDFile("./data/pcl_logo.pcd", *cloud);
    pcl::visualization::CloudViewer viewer("Cloud Viewer");
    //这里会一直阻塞直到点云被渲染
    viewer.showCloud(cloud);

    // 循环判断是否退出
    while (!viewer.wasStopped()) {
        // 你可以在这里对点云做很多处理
    }
    return 0;
}
```

但是，在该实验中，因为画出了坐标轴，坐标轴比较短，显得没有点云，但是实际上是有的，点云太大了，需要将画面缩小一点看





2.在点云中添加一些文字或者绘图、改变背景颜色等

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

int user_data;

void
viewerOneOff(pcl::visualization::PCLVisualizer &viewer) {
    // 设置背景色为粉红色
    viewer.setBackgroundColor(1.0, 0.5, 1.0);
    pcl::PointXYZ o;
    o.x = 1.0;
    o.y = 0;
    o.z = 0;
    //    添加一个圆心为o，半径为0.25m的球体
    viewer.addSphere(o, 0.25, "sphere", 0);
    std::cout << "i only run once" << std::endl;
}

void
viewerPsycho(pcl::visualization::PCLVisualizer &viewer) {
    static unsigned count = 0;
    std::stringstream ss;
    ss << "Once per viewer loop: " << count++;
    // 每次刷新时，移除text，添加新的text
    viewer.removeShape("text", 0);
    viewer.addText(ss.str(), 200, 300, "text", 0);

    //FIXME: possible race condition here:
    user_data++;
}

int
main() {
    pcl::PointCloud<pcl::PointXYZRGBA>::Ptr cloud(new
pcl::PointCloud<pcl::PointXYZRGBA>);
    pcl::io::loadPCDFile("./data/pcl_logo.pcd", *cloud);
    pcl::visualization::CloudViewer viewer("Cloud Viewer");

    //这里会一直阻塞直到点云被渲染
    viewer.showCloud(cloud);

    // 只会调用一次 (非必须)
    viewer.runOnVisualizationThreadOnce (viewerOneOff);
    // 每次可视化迭代都会调用一次（频繁调用） (非必须)
    viewer.runOnVisualizationThread (viewerPsycho);
    while (!viewer.wasStopped()) {
        user_data++;
    }
    return 0;
}
```

Visualizer

```
创建一个PCLVisualizer：
boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer(new
pcl::visualization::PCLVisualizer("3D viewer"));
viewer->setBackgroundColor(0.05,0.05,0.05,0);
```

点云拓扑结构

通过3D相机（雷达、激光扫描、立体相机）获取到的点云，一般数据量较大，分布不均匀，数据主要表征了目标物表面的大量点的集合，这些离散的点如果希望实现基于邻域关系的**快速查找比对功能**，就必须对这些离散的点之间建立拓扑关系。常见的空间索引一般是自上而下逐级划分空间的各种索引结构，包括BSP树，k-d tree、KDB tree、R tree、CELL tree、八叉树等。有了这些关系，我们就可以实现点云的降采样，计算特征向量，点云匹配，点云拆分等功能。

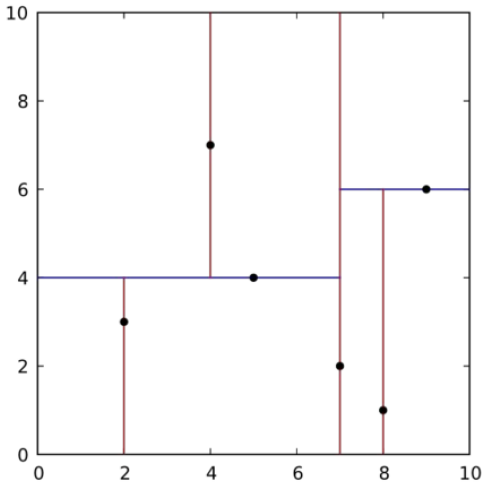
k-d tree（ k-dimensional tree)

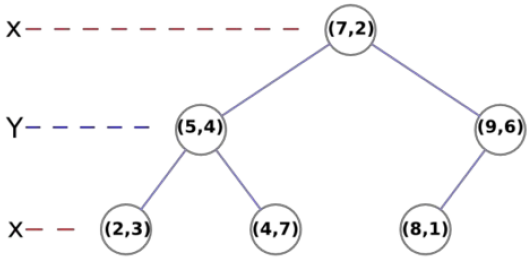
是计算机科学中用于在k维空间中一些点建立关系的数据结构。它是一个包含特定约束的二叉搜索树。k-d tree对于**范围搜索和最近邻居搜索**非常有用。我们通常只处理三维空间的点云，因此我们所有的k-d树都是三维空间的。

k-d树的每个级别都使用垂直于相应轴的超平面沿特定维度拆分所有子级。在树的根部，所有子项都将根据第一维进行拆分（即，如果第一维坐标小于根，则它将位于左子树中，如果大于根，则显然位于右边的子树）。树中向下的每个级别都在下一个维度上划分，其他所有元素都用尽后，将返回到第一个维度。他们构建k-d树的最有效方法是使用一种分区方法，例如快速排序所用的一种方法，将中值点放置在根上，所有具有较小一维值的东西都放置在根部，而右侧则更大。然后，在左右两个子树上都重复此过程，直到要分区的最后一棵树仅由一个元素组成。

树的分割方式：

2D分割

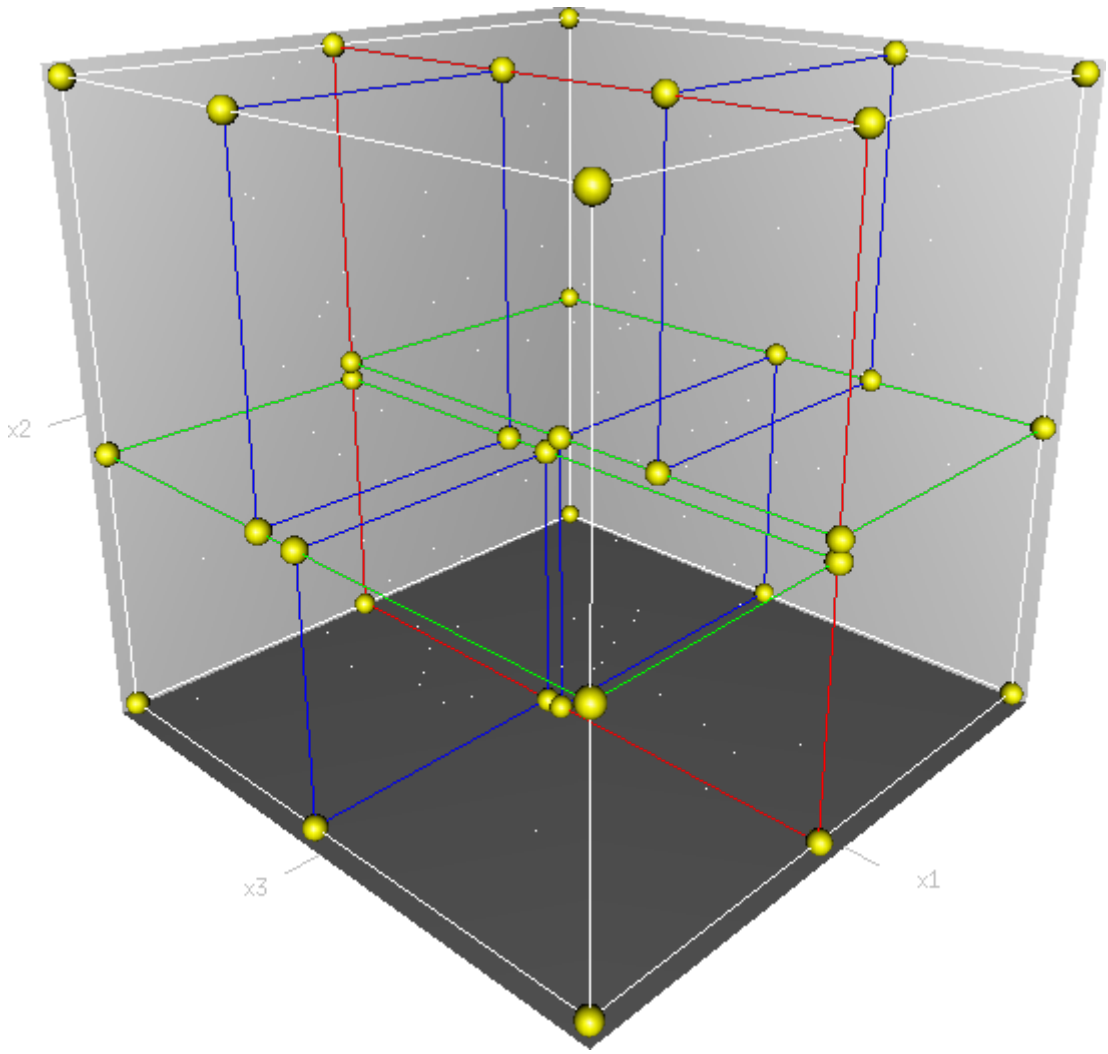




所有点为{（2,3），（5,4），（9,6），（4,7），（8,1），（7,2）}

分割时选方差较大的那个维度进行分割

3D分割



k-d tree的查询

最近邻查询

1-NN

假如只查最近的一个节点，维护一个搜索点到目前找到最近叶子节点的最小值shortestDistance，初始值为DOUBLE_MAX;计算搜索点到 当前节点的外界矩形的最小距离min_shorest_dis,如果这个距离超过了shortestDistance，那么这个节点没有必要搜索，因为min_shorest_dis代表的就是当前节点中所有点到目标节点的最小值的下界。

```
#include <pcl/point_cloud.h>
#include <pcl/kdtree/kdtree_flann.h>

#include <iostream>
#include <vector>
#include <ctime>
//#include <pcl/search/kdtree.h>
//#include <pcl/search/impl/search.hpp>
#include <pcl/visualization/cloud_viewer.h>

int
main(int argc, char **argv) {
    // 用系统时间初始化随机种子
    srand(time(NULL));

    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new
pcl::PointCloud<pcl::PointXYZ>);

    // 生成点云数据1000个
    cloud->width = 1000;
    cloud->height = 1;  // 1 表示点云为无序点云
    cloud->points.resize(cloud->width * cloud->height);

    // 给点云填充数据 0 - 1023
    for (size_t i = 0; i < cloud->points.size(); ++i) {
        cloud->points[i].x = 1024.0f * rand() / (RAND_MAX + 1.0f);
        cloud->points[i].y = 1024.0f * rand() / (RAND_MAX + 1.0f);
        cloud->points[i].z = 1024.0f * rand() / (RAND_MAX + 1.0f);
    }

    // 创建KdTree的实现类KdTreeFLANN (Fast Library for Approximate Nearest
Neighbor)
    pcl::KdTreeFLANN<pcl::PointXYZ> kdtree;
    // pcl::search::KdTree<pcl::PointXYZ> kdtree;
    // 设置搜索空间，把cloud作为输入
    kdtree.setInputCloud(cloud);

    // 初始化一个随机的点，作为查询点
    pcl::PointXYZ searchPoint;
    searchPoint.x = 1024.0f * rand() / (RAND_MAX + 1.0f);
    searchPoint.y = 1024.0f * rand() / (RAND_MAX + 1.0f);
    searchPoint.z = 1024.0f * rand() / (RAND_MAX + 1.0f);

    // K nearest neighbor search
    // 方式一：搜索K个最近邻居
```

```
    // 创建K和两个向量来保存搜索到的数据
    // K = 10 表示搜索10个临近点
    // pointIdxNKNSearch         保存搜索到的临近点的索引
    // pointNKNSquaredDistance   保存对应临近点的距离的平方
    int K = 10;
    std::vector<int> pointIdxNKNSearch(K);
    std::vector<float> pointNKNSquaredDistance(K);

    std::cout << "K nearest neighbor search at (" << searchPoint.x
              << " " << searchPoint.y
              << " " << searchPoint.z
              << ") with K=" << K << std::endl;

    if (kdtree.nearestKSearch(searchPoint, K, pointIdxNKNSearch,
pointNKNSquaredDistance) > 0) {
        for (size_t i = 0; i < pointIdxNKNSearch.size(); ++i)
            std::cout << "    " << cloud->points[pointIdxNKNSearch[i]].x
                      << " " << cloud->points[pointIdxNKNSearch[i]].y
                      << " " << cloud->points[pointIdxNKNSearch[i]].z
                      << " (距离平方: " << pointNKNSquaredDistance[i] << ")" <<
std::endl;
    }

    // Neighbors within radius search
    // 方式二：通过指定半径搜索
    std::vector<int> pointIdxRadiusSearch;
    std::vector<float> pointRadiusSquaredDistance;

    // 创建一个随机[0,256)的半径值
    float radius = 256.0f * rand() / (RAND_MAX + 1.0f);

    std::cout << "Neighbors within radius search at (" << searchPoint.x
              << " " << searchPoint.y
              << " " << searchPoint.z
              << ") with radius=" << radius << std::endl;


    if (kdtree.radiusSearch(searchPoint, radius, pointIdxRadiusSearch,
pointRadiusSquaredDistance) > 0) {
        for (size_t i = 0; i < pointIdxRadiusSearch.size(); ++i)
            std::cout << "    " << cloud->points[pointIdxRadiusSearch[i]].x
                      << " " << cloud->points[pointIdxRadiusSearch[i]].y
                      << " " << cloud->points[pointIdxRadiusSearch[i]].z
                      << " (距离平方:: " << pointRadiusSquaredDistance[i] << ")"
<< std::endl;
    }

    pcl::visualization::PCLVisualizer viewer("PCL Viewer");
    viewer.setBackgroundColor(0.0, 0.0, 0.5);
    viewer.addPointCloud<pcl::PointXYZ>(cloud, "cloud");

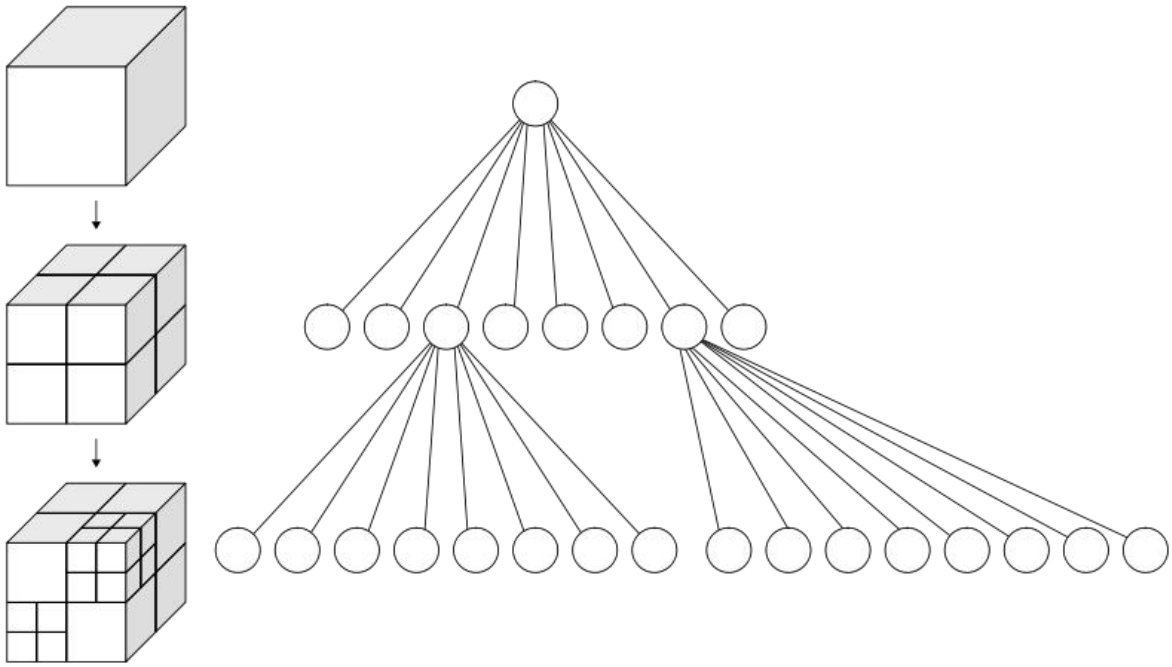
    pcl::PointXYZ originPoint(0.0, 0.0, 0.0);
    // 添加从原点到搜索点的线段
    viewer.addLine(originPoint, searchPoint);
    // 添加一个以搜索点为圆心，搜索半径为半径的球体
    viewer.addSphere(searchPoint, radius, "sphere", 0);
    // 添加一个放到200倍后的坐标系
```

```
    viewer.addCoordinateSystem(200);

    while (!viewer.wasStopped()) {
        viewer.spinOnce();
    }

    return 0;
}
```

八叉树



八叉树（Octree）的定义是：若不为空树的话，树中任一节点的子节点恰好只会有八个，或零个，也就是子节点不会有0与8以外的数目。那么，这要用来做什么？想象一个立方体，我们最少可以切成多少个相同等分的小立方体？答案就是8个。再想象我们有一个房间，房间里某个角落藏着一枚金币，我们想很快的把金币找出来，聪明的你会怎么做？我们可以把房间当成一个立方体，先切成八个小立方体，然后排除掉没有放任何东西的小立方体，再把有可能藏金币的小立方体继续切八等份....如此下去，平均在log8(n)log8(n)（n表示房间内的所有物体数）的时间内就可找到金币。因此，八叉树就是用在3D空间中的场景管理，可以很快地知道物体在3D场景中的位置，或侦测与其它物体是否有碰撞以及是否在可视范围内。

P