工程分析和项目总结

1 一切从Main函数开始

首先从**VelodyneViewer_main.cpp**程序开始分析，这里定义了main函数，是整个工程开始的地方。

```
char lidar_type[64] = "Velodyne-HDL-32E";
...
LoadCfgVeloView(g_CfgVeloView, "./velodyne.ini");
```

程序一开始判断了当前所使用的LiDAR是多少线的，我们实验用的就是32线的LiDAR。随后调用函数进行了全局配置。存储全局配置的对象在**configconStruct.h**文件中定义，包括了各种对于激光雷达测试过程中的参数初始化定义。而这些参数的来源都在**velodyne.ini**中，我们可以看一下大致的内容：

可以看到这里就有一些对于激光最大最小检测距离的设置，我们也是对这里的参数进行调整从而实现我们的filter。配置完全局配置文件之后，往下看到有一大堆的关于线程的操作。过程非常的复杂，概括来说，就是创建出几个线程，并且设置他们的优先级、调度算法等。那么提取出其中几个关键的步骤：

```
pthread_create(&(g_socket_th), NULL, &(SocketThread), NULL);
...
pthread_create(&(g_opengl_th), NULL, &(OpenGLThread), NULL);
```

main函数当前正在运行，可以看做是主线程，而在这里就创建了两个子线程，分别命名为`g_socket_th`和`g_opengl_th`。对于`pthread_create`这个函数来说，第三个参数就是线程函数的起始位置，也就是说线程实际运行的时候将会从这个函数开始执行。
创建完这些线程之后，那么就运用了下面的语句：

```
pthread_join(g_opengl_th, NULL);
pthread_join(g_socket_th, NULL);
```

这个函数的意义是，将子线程加入到队列当中去，而且主线程必须要等这两个线程都结束之后，才能够结束。剩下还有一些代码是说讲线程结束的，不必分析。也就是说，程序就是从这里正式开始运行了。而最为关键的，就是两个线程背后的起始函数，我们接下来的两个版块将分别对这两个线程函数进行分析。

2 SocketThread做了什么

首先来看`SocketThread`这个线程函数，因为它是先被创建出来的。借助编辑器我们跳转到它的定义位置，在**velodyneThread.cpp**中。

函数一开始定义了一个packet的对象和一个Driver的对象，然后就是冗长的一大堆的代码。上网查找资料得知，这些是关于window下Sokcet套接字编程的。回想我们当时测试的时候，电脑和LiDAR是通过网线连接的，那么前面的这些准备工作就是来让本地程序与LiDAR构建起连接。相关的设置包括协议域、类型、协议层等。这里和函数的核心部分无关，不做详细分析。
终于，在经过了漫长的连接建立之后，进入了一个大循环。

那么你一定可以想得到，连接之后，当然就要开始读数据了。所以这个循环就是不断从LiDAR那边读取数据出来的。大致可以分为`recvfrom`、`isNewScan`、`recvPacket`三个步骤:

2.1 recvfrom

```
bytes_received = recvfrom(sockfd,
                          (char*)&pkt, sizeof(pkt),
                          0,
                          (struct sockaddr *)&client, &client_length);
```

第一步还是借助套接字编程所提供的函数，将字节流从服务端读取出来，也就是一个个的数据包。接下来对读取到的数据包进行分析。

2.2 isNewScan

```
velodyneDriver.isNewScan(pkt)
```

第二步执行的是`isNewScan`这个操作，它的作用是判断当前的包是否是扫描新的一周所得到的数据。它的定义在**velodyneDriver.cpp**中。

其判断过程是，先检查包头看是否被篡改，然后看是不是初始值。最关键的是拿当前数据包的第一个block的旋转角来和上一次的旋转角做对比，看看旋转的角度保证已经>=360°了，这样就意味着转完了一圈，足够作为一个新的数据包。
那么一旦判定是新的一圈得到的数据，就就要开始判断下一个应该写在缓冲区的数据位置在哪里。

2.3 recvPacket

```
velodyneDriver.recvPacket(pkt, g_scanBuffer[g_scanBufferWriteIdx]);
```

第三步，在检查完上面的包的顺序之后，此时已经决定了新的包写入的缓冲区的位置，那么就调用这个函数放到velodyneDriver这个对象的成员缓冲区当中去存储。大家应该对这一部分有一点印象，我们第一次写程序的时候就是从这个地方开始要求我们做出一定的修改的，每个packet包括12次fire，每个fire往32个高度发射了32个laser。对满足最大、最小具体范围条件内的数据，通过坐标转换、量纲转换等操作，整理成一个新的包，然后加到缓冲区当中去。32线的LiDAR要求每一圈下来获取到的包具有180个以上。

2.4 conclusion

那么到了这里，整个`SocketThread`的过程就到这里结束了。总之这一部分执行的操作就是和LiDAR构建起连接，不断地获取数据包，然后存储下来。

3.OpenGLThread做了什么

找到这个函数的定义，同样也是在**velodyneThread.cpp**下，定义非常简单：

```
extern int argc_gl;
extern char **argvs_gl;
void* OpenGLThread(void* arg){
    glutInit(&argc_gl, argvs_gl);
    MyGLDispIni();
}
```

只是执行了两个函数，从第一个函数就可以猜想，这些操作是和GLUT相关的，它对GLUT进行初始化，在进行其他的GLUT使用之前必须调用一次，比较死板。那么关键还在下面`MyGLDispIni`函数的执行。

3.1 MyGLDispIni

该函数的定义在**velodyneDraw.cpp**当中，从文件名称可以猜想这个函数是和绘图有关的。

还是借助GLUT中的库函数执行了一系列的操作，包括设置显示模式、窗口大小和名称、改变窗口的响应和键盘操作窗口的响应等。对于`SpecialKey`，我们在实验过程中，通过方向键、pgup、pgdn就可以调整观察角度。
而最关键的就是`myDisplay`，决定了具体显示的方式。

3.2 myDisplay

前面都是关于颜色、旋转、视角、旋转、缩放、平移等操作，关键在于对`show_state`这个参数的选择上，决定了接下来要执行什么样的操作。

而在文件的最上方，就对其进行了初始化:

```
uint8_t show_state = SHOW_ALL_POINTS_BELOW;
```

上面也分析过了，可以对这个参数进行调整，从而改变当前的显示模式，我们选择了只显示平面以上的所有点。继续向下探寻，就是`drawAllPoints`这个函数。

3.3 drawAllPoints

首先根据传进来的参数，设置了`circle_start`和`circle_end`两个参数，后者设置的是一个fire下有多少个laser即32，前者这是设置对每一个fire从哪一个开始输出。如果设置的是高于地面的话，那么就是从23才开始，有效过滤掉很多的点。

最后终于到了产生图像的时候，每次的输出都是将同一水平高度的所有点输出，再输出下一高度。初始设置下，第一象限的点为绿色，第二象限的点为红色，第三象限为浅蓝色。第四象限为深蓝色。

4 总结

到这里我们大致理解了整个运行框架。总之就是main函数创建了两个线程。SocketThread负责读取从LiDAR读取并处理数据。OpenGLThread负责将处理好的数据绘制成图。