程序架构

头文件

- base.h ---结构体定义
- PCOption.h---配置文件定义
- SIFT.h ---三维点特征匹配、BA平差

源文件

- base.cpp
- PCOption.cpp
- SIFT.cpp
- main.cpp

附加包含目录

- D:\Thirdparty\PVADLL(x64)\include\BASEDLL
- D:\Thirdparty\PVADLL(x64)\include\GNSSDLL
- D:\Thirdparty\PVADLL(x64)\include\SINSDLL
- D:\Thirdparty\SIFTGPU\lib

附加依赖项：

- PVADLL.lib
- DevIL.lib
- glew32.lib
- glew32s.lib
- glew64.lib
- glew64s.lib
- glut32.lib
- glut64.lib
- SIFTGPU.lib
- opencv_world341.lib
- DBoW3.lib

核心步骤

- 步骤零：初始化，从配置文件、时间文件、真值文件读取信息
- 步骤一：从序列影像和真值位姿中增量构建特征点的三维地图数据

- 步骤二：将构建出的每个地图点采用不同的像素观测值进行平差

```
bool CPointCloud::runPointCloud(string optfile, CPCOPTION * pPCOpt)
{
    if (pPCOpt) m_PCOpt = *pPCOpt;
    if (m_PCOpt.readOptFile(optfile) == false) return false;
    //1.初始化
    if (Init(&m_PCOpt) == false) return false;

    //2.从序列影像和真值位姿中增量构建特征点的三维地图数据
    for (int j = 0; j < m_vTimestamps.size(); j++)
    {
        m_frameIndex = j;


        //获取所有影像的左右图像匹配情况
        if (RunOneLRSIFT() == false)
            return false;

        RunSeqMatch();
        InitPoint();
        UpdateInfo();
    }

    //3.将构建出的每个地图点采用不同的像素观测值进行平差
    DealAllPoint();

    //4.筛选地图数据库中的地图点并确定其对应的特征描述子们
    //SelectPoint();
    //5.保存剩余地图点文件
    //SaveFile();

    return true;
}
```

SIFTGPU

左右影像特征匹配和滑窗内影像匹配用了Wu Changchang写的开源代码SIFTGPU：

特征匹配是SLAM中非常耗时间的一步，许多人都想把它的时间降至最短，因此目前ORB成了非常受欢迎的特征。而老牌SIFT，则一直给人一种“很严谨很精确，但计算非常慢”的印象。在一个普通的PC上，计算一个640×480的图中的SIFT大约需要几百毫秒左右。如果特征都要算300ms，加上别的ICP什么的，一个SLAM就只能做成两帧左右的速度了，这是很令人失望的。而ORB，FAST之类的特征，由于计算速度较快，在SLAM这种实时性要求较高的场合更受欢迎。

Wu Changchang同学写的GPU版本的SIFT。它能够明显地提升你的程序提取SIFT的速度。同时，它的代码大部分是基于OpenGL的，即使在没有英伟达显卡的机器上也能运行起来。

> 拓展：OpenGL是Khronos Group开发维护的一个规范，它主要为我们定义了用来操作图形和图片的一系列函数的API，需要注意的是OpenGL本身并非API。GPU的硬件开发商则需要提供

> 满足OpenGL规范的实现，这些实现通常被称为“驱动”，它们负责将OpenGL定义的API命令翻译为GPU指令。

```
// SiftGPU模块
#include <SiftGPU.h>

//标准C++
#include <iostream>
#include <vector>

// OpenCV图像
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>

// boost库中计时函数
#include <boost/timer.hpp>

// OpenGL
#include <GL/gl.h>

using namespace std;

int main( int argc, char** argv)
{
    //声明SiftGPU并初始化
    SiftGPU sift;
    char* myargv[4] ={ "-fo", "-1", "-v", "1"};
    sift.ParseParam(4, myargv);

    //检查硬件是否支持SiftGPU
    int support = sift.CreateContextGL();
    if ( support != SiftGPU::SIFTGPU_FULL_SUPPORTED )
    {
        cerr<<"SiftGPU is not supported!"<<endl;
        return 2;
    }

    //测试直接读取一张图像
    cout<<"running sift"<<endl;
    boost::timer timer;
    //在此填入你想测试的图像的路径！不要用我的路径！不要用我的路径！不要用我的路径！
    sift.RunSIFT( "/home/xiang/wallE-slam/data/rgb1.png" );
    cout<<"siftgpu::runsift() cost time="<<timer.elapsed()<<endl;

    // 获取关键点与描述子
    int num = sift.GetFeatureNum();
    cout<<"Feature number="<<num<<endl;
    vector<float> descriptors(128*num);
    vector<SiftGPU::SiftKeypoint> keys(num);
    timer.restart();
    sift.GetFeatureVector(&keys[0], &descriptors[0]);
    cout<<"siftgpu::getFeatureVector() cost time="<<timer.elapsed()<<endl;

    // 先用OpenCV读取一个图像，然后调用SiftGPU提取特征
    cv::Mat img = cv::imread("/home/xiang/wallE-slam/data/rgb1.png", 0);
    int width = img.cols;
```

```
    int height = img.rows;
    timer.restart();
    // 注意我们处理的是灰度图，故照如下设置
    sift.RunSIFT(width, height, img.data, GL_INTENSITY8, GL_UNSIGNED_BYTE);
    cout<<"siftgpu::runSIFT() cost time="<<timer.elapsed()<<endl;

    return 0;
}
```

Sift接口还是相当简单的。在这程序里，我们一共做了三件事。一是直接对一个图像路径提Sift，二是获取Sift的关键点和描述子。三是对OpenCV读取的一个图像提取Sift。

关键点的定义：

```
    typedef struct SiftKeypoint
    {
        float x, y, s, o; //x, y, scale, orientation.
    }SiftKeypoint;
```

描述子的定义：

```
vector<unsigned char> des;//128维
```

本代码中规定的每一个特征点的sift信息结构体：

```
typedef struct tagSIFTInfo
{

    vector<SiftGPU::SiftKeypoint> Keys;
    vector<float> UR;
    vector<unsigned char> des;
    vector<long int> Point_index;
    int KeyNum;
    tagSIFTInfo()
    {
        ZeroStruct(*this, tagSIFTInfo);

    }
}SIFTInfo;
```

左右影像匹配和滑窗内序列影像匹配

本文中，滑窗宽度为20帧，对每一帧双目影像，首先进行左右图像匹配，获得当前左右匹配到的特征点对，只有当前帧能够匹配到的点才去slidingwindow里找过去是否也能被看到。

为验证当前帧特征点和滑窗历史某特征点是否是真正的对应匹配，引入对极几何约束：

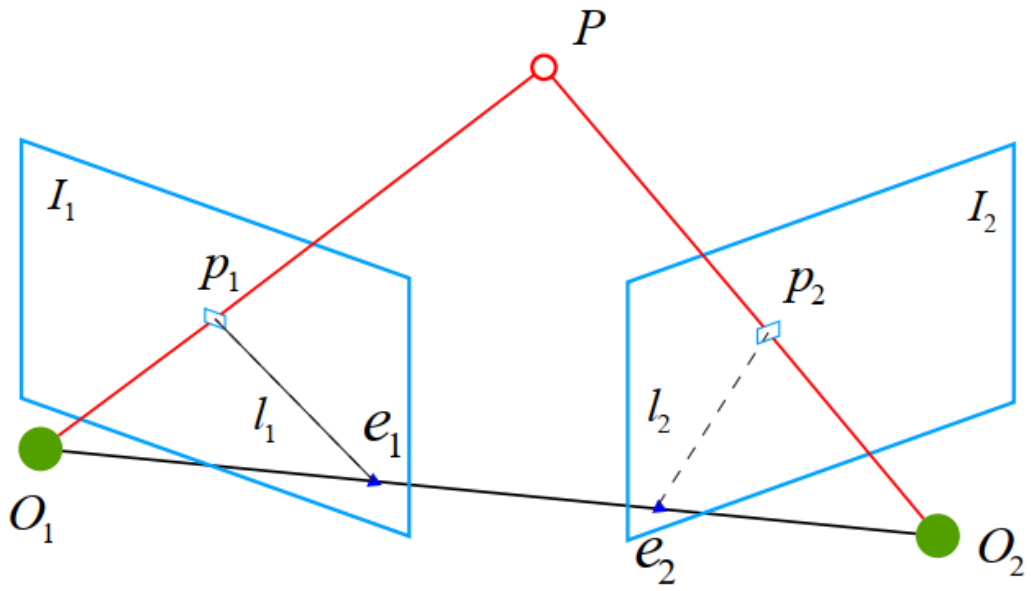

图 7-7　对极几何约束。

$$\boldsymbol{p}_2^T \boldsymbol{K}^{-T} \boldsymbol{t}^\wedge \boldsymbol{R} \boldsymbol{K}^{-1} \boldsymbol{p}_1 = 0. \tag{7.8}$$

这两个式子都称为**对极约束**，它以形式简洁著名。它的几何意义是 O_1, P, O_2 三者共面。对极约束中同时包含了平移和旋转。我们把中间部分记作两个矩阵：基础矩阵（Fundamental Matrix）$\boldsymbol{F}$ 和本质矩阵（Essential Matrix）$\boldsymbol{E}$，可以进一步简化对极约束：

$$\boldsymbol{E} = \boldsymbol{t}^\wedge \boldsymbol{R}, \quad \boldsymbol{F} = \boldsymbol{K}^{-T} \boldsymbol{E} \boldsymbol{K}^{-1}, \quad \boldsymbol{x}_2^T \boldsymbol{E} \boldsymbol{x}_1 = \boldsymbol{p}_2^T \boldsymbol{F} \boldsymbol{p}_1 = 0. \tag{7.9}$$

初始化点的坐标

存储所有的地图点：

```
map<int,PointXYZ> m_AllPoint;//地图点编号从0开始

typedef struct tagUV
{
    int Frame_id;
    int Des_id;
    double UV[3];//UV[0]=uL UV[1]=vL UV[2]=uL-uR
    int ValidFlag;
    double BackProjectUVError[3];
    double MeanBP_UVError;
    tagUV()
    {
        ZeroStruct(*this, tagUV);

    }
}UV;
```

```
typedef struct tagPointXYZ
{
    long int Point_id;
    double XYZ[3];
    double VarXYZ[9];
    int Recent_FI;//要更新，看到这个点的最新一帧
    int n_Frame;  //共视帧数
    vector<UV> uvlist;

    tagPointXYZ()
    {
        ZeroStruct(*this, tagPointXYZ);

    }
}PointXYZ;
```

把左右匹配、滑窗匹配后的特征点，在地图点云库中找到对应的序号，如果第一次出现，初始化当前点的序号为m_PointIndex，如果历史时刻出现过，在点云库中找到对应的点序号，更新该点的uvlist和共视帧数。

uvlist的含义：Frame_id代表第几帧有该特征点，去Frame_id.sift文件中查找描述子description；Des_id定位描述子的在文件中的位置，一个被多帧共视的特征子会对应多个描述子；UV[3]代表在这一帧影像上，特征子的像素坐标和左右横坐标之差；validflag用于后续的抗差估计，当一个特征子重投影误差太大，删除该特征子的像素观测值，ValidFlag=0表示该点不可用；

三维点坐标联合优化

取共视帧数大于等于3且小于50的路标点进行光束法平差（Bundle Adjustment）。

```
CurPoint = m_AllPoint[Point_id];
    while (CurPoint.n_Frame>1)
    {
        //1.做一次BA
        //每一次BA开始前，状态置为1；
        m_BAState = 1;
        RunOneBA();
        //2.计算像素的反投影误差 判断反投影误差是否超过阈值
        CalBackProjectError();
        if (m_BAState == 1)
        {
            printf("%d th point:BA success\n",Point_id);
            break;
        }
        else
        {
            //3.超过残差设定阈值，找到残差最大的对应观测值，并剔除
            int MeasID = 0;
            double MaxMeanBacProjectError = 0.0;
            for (int j = 0; j < CurPoint.uvlist.size(); j++)
            {
                if (CurPoint.uvlist[j].ValidFlag > 0 && CurPoint.uvlist[j].MeanBP_UVError
> MaxMeanBacProjectError)
```

```
                {
                    MeasID = j;
                }
            }
            CurPoint.uvlist[MeasID].ValidFlag = 0;
            CurPoint.n_Frame = CurPoint.n_Frame - 1;
            //CurPoint的初值变了，保持初值不变
            CurPoint.XYZ[0] = m_AllPoint[Point_id].XYZ[0];
            CurPoint.XYZ[1] = m_AllPoint[Point_id].XYZ[1];
            CurPoint.XYZ[2] = m_AllPoint[Point_id].XYZ[2];
        }
    }
```

三维点联合优化的目的就是最小化重投影误差，观测值是UV[3],R、T由GNSS/INS事后解算得到，待估参数是三维点的世界坐标。引入抗差策略，可以有效提高定位精度。

点云数据清洗

对视觉点云地图进行数据清洗，可以剔除冗余的、质量不佳的、甚至坐标错误的点云，使得IBL后方交会更准确。

指标	说明
共视帧数	同名路标点被共视的帧数，共视帧数越多，方程的多余观测越多
三维点位置标准差	三维点坐标的协方差矩阵表征了点云的内符合精度
残差RMS	残差RMS大小表征了各个观测值之间自洽程度，抗差迭代后残差较小
深度值	视觉传感器探测距离有限，双目前方交会，深度值越大，交会角越小
视线夹角	相机光心和路标点的连线方向和前视方向的夹角，视线夹角越大，交会角越小
几何距离	深度值变大和视线夹角变大，几何距离随之增加，交会几何条件变差

输出文件

- xxxxxx.sift：

存储每一帧提取的特征点对应的描述子。对每一帧存储：当前帧有多少个描述子，对每一个描述子记录：描述子的序号，特征子在地图中的点序号，特征子的像素坐标，128维的描述子。

当前帧特征子/描述子总个数
描述子的序号

当前帧特征子/描述子总个数
特征子的序号
特征子的像素坐标
描述子128位数据
下一个描述子的信息

- uvlist.txt

输出数据清洗后的地图点云特征：包括点的序号、点的世界坐标、共视帧数、XYZ标准差、几何距离的数学统计、侧向距离的数学统计、深度距离的数学统计、视线夹角的数学统计

- Point.bin

当前地图点
三维坐标XYZ
坐标解算的协方差
地图点的共视帧数
第0帧的描述子序号
…
第n帧的描述子序号
下一个地图点

- Descriptor.bin

index	**data**
描述子序号0	描述子数据
描述子序号1	描述子数据

Point.bin和Descriptor.bin就是输出的点云地图文件，存储了每一个地图点的三维坐标和所有共视帧得到的描述子