运行官方Demo

以TUM数据集为例,运行Demo的命令:

`./Examples/RGB-D/rgbd_tum Vocabulary/ORBvoc.txt Examples/RGB-D/TUM1.yaml PATH_TO_SEQUENCE_FOLDER ASSOCIATIONS_FILE`

`rgbd_tum.cc`的源码:

```
int main(int argc, char **argv) {
    // 判断输入参数个数
    if (argc != 5) {
        cerr << endl << "Usage: ./rgbd_tum path_to_vocabulary path_to_settings path_to_sequence path_to_association" << endl;
        return 1;
    }

    // step1. 读取图片及左右目关联信息
    vector<string> vstrImageFilenamesRGB;
    vector<string> vstrImageFilenamesD;
    vector<double> vTimestamps;
    string strAssociationFilename = string(argv[4]);
    LoadImages(strAssociationFilename, vstrImageFilenamesRGB, vstrImageFilenamesD, vTimestamps);

    // step2. 检查图片文件及输入文件的一致性
    int nImages = vstrImageFilenamesRGB.size();
    if (vstrImageFilenamesRGB.empty()) {
        cerr << endl << "No images found in provided path." << endl;
        return 1;
    } else if (vstrImageFilenamesD.size() != vstrImageFilenamesRGB.size()) {
        cerr << endl << "Different number of images for rgb and depth." << endl;
        return 1;
    }

    // step3. 创建SLAM对象,它是一个 ORB_SLAM2::System 类型变量
    ORB_SLAM2::System SLAM(argv[1], argv[2], ORB_SLAM2::System::RGBD, true);

    vector<float> vTimesTrack;
    vTimesTrack.resize(nImages);
    cv::Mat imRGB, imD;
    // step4. 遍历图片,进行SLAM
    for (int ni = 0; ni < nImages; ni++) {
        // step4.1. 读取图片
        imRGB = cv::imread(string(argv[3]) + "/" + vstrImageFilenamesRGB[ni], CV_LOAD_IMAGE_UNCHANGED);
        imD = cv::imread(string(argv[3]) + "/" + vstrImageFilenamesD[ni], CV_LOAD_IMAGE_UNCHANGED);
        double tframe = vTimestamps[ni];
        // step4.2. 进行SLAM
        SLAM.TrackRGBD(imRGB, imD, tframe);
        // step4.3. 加载下一张图片
        double T = 0;
        if (ni < nImages - 1)
            T = vTimestamps[ni + 1] - tframe;
        else if (ni > 0)
            T = tframe - vTimestamps[ni - 1];

        if (ttrack < T)
            usleep((T - ttrack) * 1e6);
    }

    // step5. 停止SLAM
    SLAM.Shutdown();
}
```

运行程序`rgbd_tum`时传入了一个重要的配置文件`TUM1.yaml`,其中保存了**相机参数**和**ORB特征提取参数**:

```
%YAML:1.0

## 相机参数
Camera.fx: 517.306408
```

```
Camera.fy: 516.469215
Camera.cx: 318.643040
Camera.cy: 255.313989

Camera.k1: 0.262383
Camera.k2: -0.953104
Camera.p1: -0.005358
Camera.p2: 0.002628
Camera.k3: 1.163314

Camera.width: 640
Camera.height: 480

Camera.fps: 30.0          # Camera frames per second
Camera.bf: 40.0           # IR projector baseline times fx (aprox.)
Camera.RGB: 1             # Color order of the images (0: BGR, 1: RGB. It is ignored if images are grayscale)
ThDepth: 40.0             # Close/Far threshold. Baseline times.
DepthMapFactor: 5000.0    # Deptmap values factor

## ORB特征提取参数
ORBextractor.nFeatures: 1000         # ORB Extractor: Number of features per image
ORBextractor.scaleFactor: 1.2        # ORB Extractor: Scale factor between levels in the scale pyramid
ORBextractor.nLevels: 8              # ORB Extractor: Number of levels in the scale pyramid
ORBextractor.iniThFAST: 20
ORBextractor.minThFAST: 7
```

阅读代码之前你应该知道的事情

变量命名规则

ORB-SLAM2中的变量遵循一套命名规则:

- 变量名的第一个字母为`m`表示该变量为某类的成员变量.
- 变量名的第一、二个字母表示数据类型:
  - `p`表示指针类型
  - `n`表示`int`类型
  - `b`表示`bool`类型
  - `s`表示`std::set`类型
  - `v`表示`std::vector`类型
  - `l`表示`std::list`类型
  - `KF`表示`KeyFrame`类型

这种将变量类型写进变量名的命名方法叫做**匈牙利命名法**.

理解多线程

为什么要使用多线程?

1. 加快运算速度:

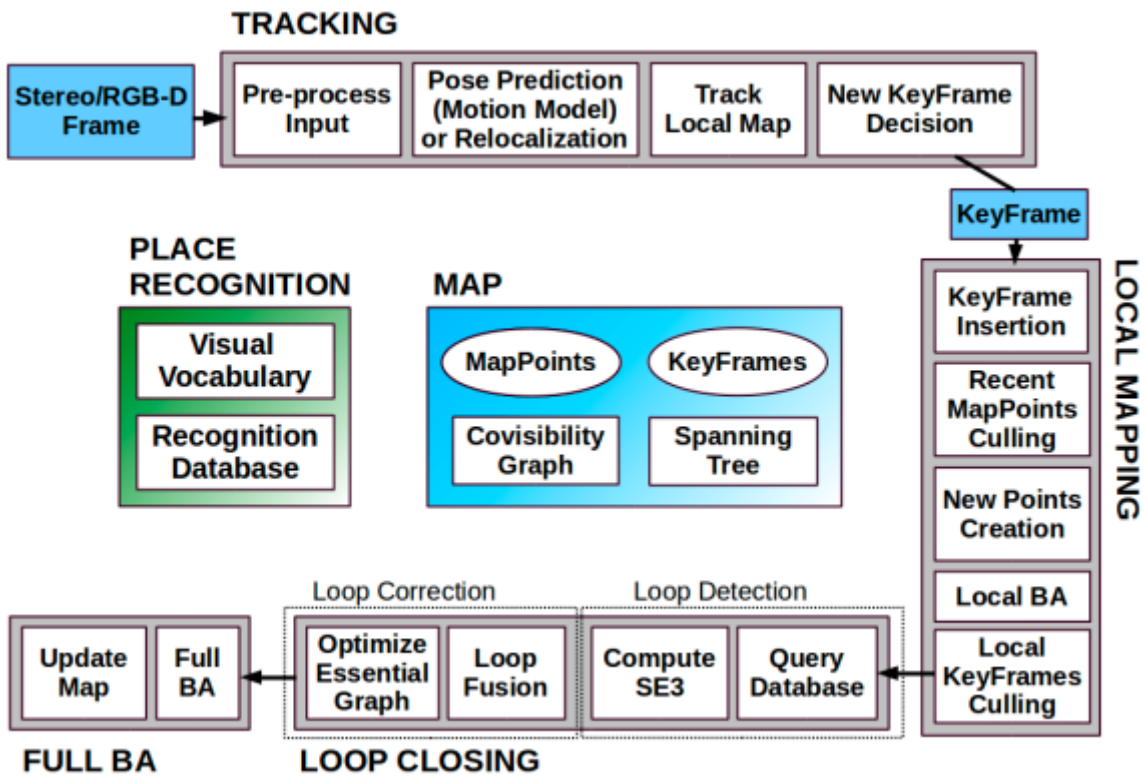
```cpp
bool Initializer::Initialize(const Frame &CurrentFrame) {
    // ...
    thread threadH(&Initializer::FindHomography, this, ref(vbMatchesInliersH), ref(SH), ref(H));
    thread threadF(&Initializer::FindFundamental, this, ref(vbMatchesInliersF), ref(SF), ref(F));
    // ...
}
```

   开两个线程同时计算两个矩阵,在多核处理器上会加快运算速度.

2. 因为系统的随机性,各步骤的运行顺序是不确定的.

   `Tracking`线程不产生关键帧时,`LocalMapping`和`LoopClosing`线程基本上处于空转的状态.

   而`Tracking`线程产生关键帧的频率和时机不是固定的,因此需要3个线程同时运行,`LocalMapping`和`LoopClosing`线程不断循环查询`Tracking`线程是否产生关键帧,产生了的话就处理.




```cpp
// Tracking线程主函数
void Tracking::Track() {
    // 进行跟踪
    // ...

    // 若跟踪成功,根据条件判定是否产生关键帧
    if (NeedNewKeyFrame())
        // 产生关键帧并将关键帧传给LocalMapping线程
        KeyFrame *pKF = new KeyFrame(mCurrentFrame, mpMap, mpKeyFrameDB);
        mpLocalMapper->InsertKeyFrame(pKF);
}

// LocalMapping线程主函数
void LocalMapping::Run() {
    // 死循环
    while (1) {
        // 判断是否接收到关键帧
        if (CheckNewKeyFrames()) {
            // 处理关键帧
            // ...

            // 将关键帧传给LoopClosing线程
            mpLoopCloser->InsertKeyFrame(mpCurrentKeyFrame);
        }

        // 线程暂停3毫秒,3毫秒结束后再从while(1)循环首部运行
        std::this_thread::sleep_for(std::chrono::milliseconds(3));
    }
}

// LoopClosing线程主函数
void LoopClosing::Run() {
    // 死循环
    while (1) {
        // 判断是否接收到关键帧
        if (CheckNewKeyFrames()) {
            // 处理关键帧
            // ...
        }

        // 查看是否有外部线程请求复位当前线程
        ResetIfRequested();

        // 线程暂停5毫秒,5毫秒结束后再从while(1)循环首部运行
        std::this_thread::sleep_for(std::chrono::milliseconds(5));
    }
}
```

多线程中的锁

为防止多个线程同时操作同一变量造成混乱,引入锁机制:

将成员函数本身设为私有变量(`private`或`protected`),并在操作它们的公有函数内加锁.

```cpp
class KeyFrame {
protected:
    KeyFrame* mpParent;

public:
    void KeyFrame::ChangeParent(KeyFrame *pKF) {
        unique_lock<mutex> lockCon(mMutexConnections);      // 加锁
        mpParent = pKF;
```

```
        pKF->AddChild(this);
    }

    KeyFrame *KeyFrame::GetParent() {
        unique_lock<mutex> lockCon(mMutexConnections);      // 加锁
        return mpParent;
    }
}
```

一把锁在某个时刻只有一个线程能够拿到,如果程序执行到某个需要锁的位置,但是锁被别的线程拿着不释放的话,当前线程就会暂停下来;直到其它线程释放了这个锁,当前线程才能拿走锁并继续向下执行.

- 什么时候加锁和释放锁?

  `unique_lock<mutex> lockCon(mMutexConnections);`这句话就是加锁,锁的有效性仅限于大括号`{}`之内,也就是说,程序运行出大括号之后就释放锁了.因此可以看到有一些代码中加上了看似莫名其妙的大括号.

```
void KeyFrame::EraseConnection(KeyFrame *pKF) {
    // 第一部分加锁
    {
        unique_lock<mutex> lock(mMutexConnections);
        if (mConnectedKeyFrameWeights.count(pKF)) {
            mConnectedKeyFrameWeights.erase(pKF);
            bUpdate = true;
        }
    }// 程序运行到这里就释放锁,后面的操作不需要抢到锁就能执行

    UpdateBestCovisibles();
}
```

SLAM主类`System`

`System`类是ORB-SLAM2系统的主类,先分析其主要的成员函数和成员变量:

成员变量/函数	访问控制	意义
`eSensor mSensor`	`private`	传感器类型 `MONOCULAR`,`STEREO`,`RGBD`
`ORBVocabulary* mpVocabulary`	`private`	ORB字典,保存ORB描述子聚类结果
`KeyFrameDatabase* mpKeyFrameDatabase`	`private`	关键帧数据库,保存ORB描述子倒排索引
`Map* mpMap`	`private`	地图
`Tracking* mpTracker`	`private`	追踪器
`LocalMapping* mpLocalMapper` `std::thread* mptLocalMapping`	`private` `private`	局部建图器 局部建图线程
`LoopClosing* mpLoopCloser` `std::thread* mptLoopClosing`	`private` `private`	回环检测器 回环检测线程

`Viewer* mpViewer`
`FrameDrawer* mpFrameDrawer`
`MapDrawer* mpMapDrawer`
`std::thread* mptViewer`
	`private`
	`private`
	`private`
	`private`
		查看器
		帧绘制器
		地图绘制器
		查看器线程

`System(const string &strVocFile, string &strSettingsFile, const eSensor sensor, const bool bUseViewer=true)`	`public`	构造函数

`cv::Mat TrackStereo(const cv::Mat &imLeft, const cv::Mat &imRight, const double &timestamp)`
`cv::Mat TrackRGBD(const cv::Mat &im, const cv::Mat &depthmap, const double &timestamp)`
`cv::Mat TrackMonocular(const cv::Mat &im, const double &timestamp)`
`int mTrackingState`
`std::mutex mMutexState`
	`public`
	`public`
	`public`
	`private`
	`private`
		跟踪双目相机,返回相机位姿
		跟踪RGBD相机,返回相机位姿
		跟踪单目相机,返回相机位姿
		追踪状态
		追踪状态锁

`bool mbActivateLocalizationMode`
`bool mbDeactivateLocalizationMode`
`std::mutex mMutexMode`
`void ActivateLocalizationMode()`
`void DeactivateLocalizationMode()`
	`private`
	`private`
	`private`
	`public`
	`public`
		开启/关闭纯定位模式

`bool mbReset`
`std::mutex mMutexReset`
`void Reset()`
	`private`
	`private`
	`public`
		系统复位

`void Shutdown()`	`public`	系统关闭

`void SaveTrajectoryTUM(const string &filename)`
`void SaveKeyFrameTrajectoryTUM(const string &filename)`
`void SaveTrajectoryKITTI(const string &filename)`
	`public`
	`public`
	`public`
		以TUM/KITTI格式保存相机运动轨迹和关键帧位姿

构造函数

`System(const string &strVocFile, string &strSettingsFile, const eSensor sensor, const bool bUseViewer=true)`:构造函数

```
System::System(const string &strVocFile, const string &strSettingsFile, const eSensor sensor, const bool bUseViewer) :
        mSensor(sensor), mpViewer(static_cast<Viewer *>(NULL)), mbReset(false), mbActivateLocalizationMode(false), mbDeactivateLocalizationMode(false) {

    // step1. 初始化各成员变量
    // step1.1. 读取配置文件信息
    cv::FileStorage fsSettings(strSettingsFile.c_str(), cv::FileStorage::READ);
    // step1.2. 创建ORB词袋
    mpVocabulary = new ORBVocabulary();
    // step1.3. 创建关键帧数据库,主要保存ORB描述子倒排索引(即根据描述子查找拥有该描述子的关键帧)
    mpKeyFrameDatabase = new KeyFrameDatabase(*mpVocabulary);
    // step1.4. 创建地图
    mpMap = new Map();

    // step2. 创建3大线程: Tracking、LocalMapping和LoopClosing
    // step2.1. 主线程就是Tracking线程,只需创建Tracking对象即可
    mpTracker = new Tracking(this, mpVocabulary, mpFrameDrawer, mpMapDrawer, mpMap, mpKeyFrameDatabase, strSettingsFile, mSensor);
    // step2.2. 创建LocalMapping线程及mpLocalMapper
    mpLocalMapper = new LocalMapping(mpMap, mSensor==MONOCULAR);
    mptLocalMapping = new thread(&ORB_SLAM2::LocalMapping::Run, mpLocalMapper);
    // step2.3. 创建LoopClosing线程及mpLoopCloser
```

```
    mpLoopCloser = new LoopClosing(mpMap, mpKeyFrameDatabase, mpVocabulary, mSensor!=MONOCULAR);
    mptLoopClosing = new thread(&ORB_SLAM2::LoopClosing::Run, mpLoopCloser);

    // step3. 设置线程间通信
    mpTracker->SetLocalMapper(mpLocalMapper);
    mpTracker->SetLoopClosing(mpLoopCloser);
    mpLocalMapper->SetTracker(mpTracker);
    mpLocalMapper->SetLoopCloser(mpLoopCloser);
    mpLoopCloser->SetTracker(mpTracker);
    mpLoopCloser->SetLocalMapper(mpLocalMapper);
}
```

> `LocalMapping`和`LoopClosing`线程在`System`类中有对应的`std::thread`线程成员变量,为什么`Tracking`线程没有对应的`std::thread`成员变量?
>
> 因为`Tracking`线程就是主线程,而`LocalMapping`和`LoopClosing`线程是其子线程,主线程通过持有两个子线程的指针(`mptLocalMapping`和`mptLoopClosing`)控制子线程.
>
> (ps: 虽然在编程实现上三大主要线程构成父子关系,但逻辑上我们认为这三者是并发的,不存在谁控制谁的问题).

跟踪函数

`System`对象所在的主线程就是跟踪线程,针对不同的传感器类型有3个用于跟踪的函数,其内部实现就是调用成员变量`mpTracker`的`GrabImageMonocular`(`GrabImageStereo`或`GrabImageRGBD`)方法.

传感器类型	用于跟踪的成员函数
`MONOCULAR`	`cv::Mat TrackRGBD(const cv::Mat &im, const cv::Mat &depthmap, const double &timestamp)`
`STEREO`	`cv::Mat TrackStereo(const cv::Mat &imLeft, const cv::Mat &imRight, const double &timestamp)`
`RGBD`	`cv::Mat TrackMonocular(const cv::Mat &im, const double &timestamp)`

```
cv::Mat System::TrackMonocular(const cv::Mat &im, const double &timestamp) {
    cv::Mat Tcw = mpTracker->GrabImageMonocular(im, timestamp);
    unique_lock<mutex> lock(mMutexState);
    mTrackingState = mpTracker->mState;
    mTrackedMapPoints = mpTracker->mCurrentFrame.mvpMapPoints;
    mTrackedKeyPointsUn = mpTracker->mCurrentFrame.mvKeysUn;
    return Tcw;
}
```