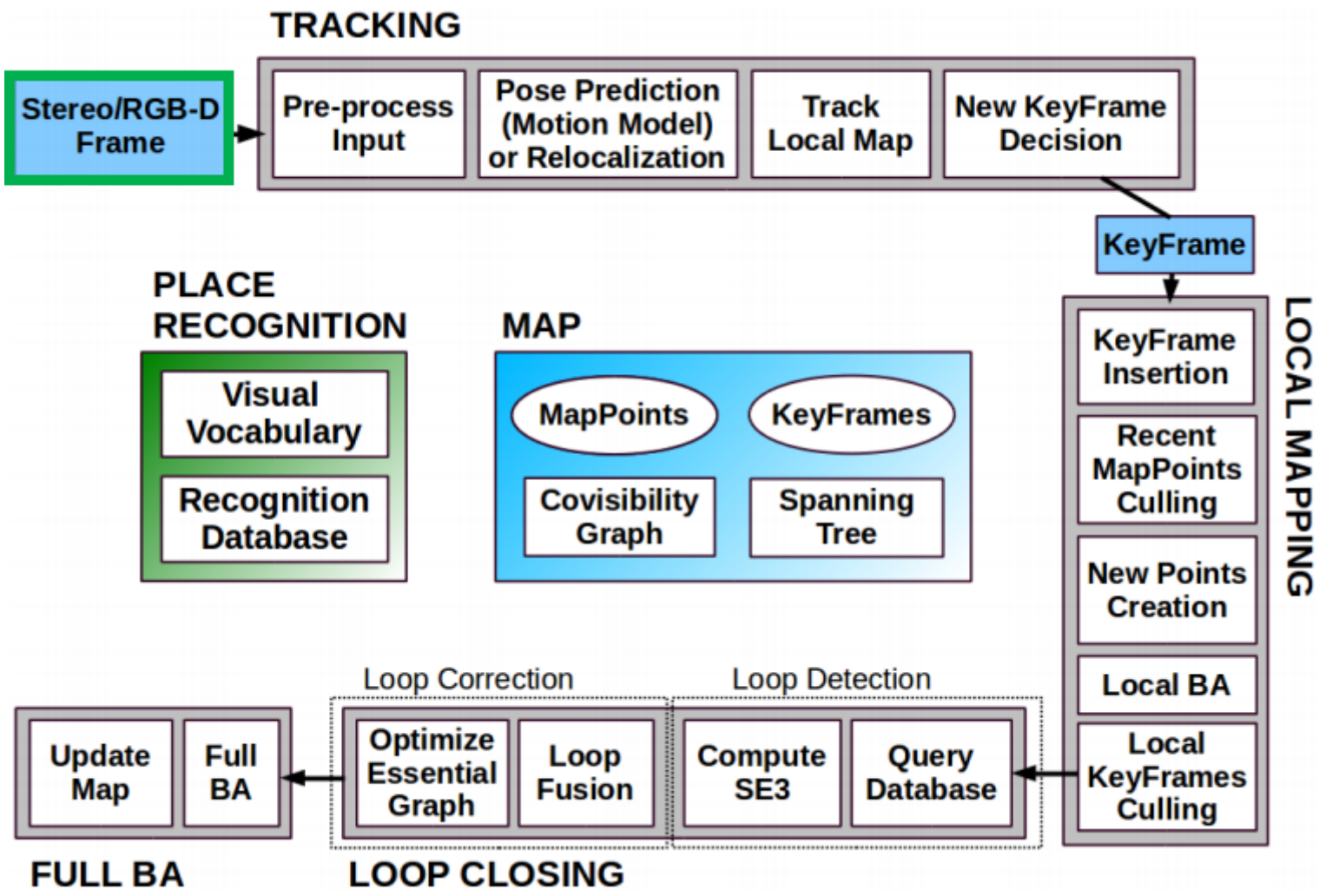


各成员函数/变量

相机相关信息

`Frame`类与相机相关的参数大部分设为`static`类型,整个系统内的所有`Frame`对象共享同一份相机参数.

成员函数/变量	访问控制	意义
`mbInitialComputations`	`public static`	是否需要为`Frame`类的相机参数赋值 初始化为`false`,第一次为相机参数赋值后变为`false`

`float fx`, `float fy`
`float cx`, `float cy`
`float invfx`, `float invfy`
	`public static`	相机内参

`cv::Mat mK`	`public`	相机内参矩阵 设为`static`是否更好?
`float mb`	`public`	相机基线,相机双目间的距离
`float mbf`	`public`	相机基线与焦距的乘积

这些参数首先由`Tracking`对象从配置文件`TUM1.yaml`内读入,再传给`Frame`类的构造函数,第一次调用`Frame`的构造函数时为这些成员变量赋值.

```
Tracking::Tracking(const string &strSettingPath, ...) {

    // 从配置文件中读取相机参数并构造内参矩阵
    cv::FileStorage fSettings(strSettingPath, cv::FileStorage::READ);
    float fx = fSettings["Camera.fx"];
    float fy = fSettings["Camera.fy"];
    float cx = fSettings["Camera.cx"];
    float cy = fSettings["Camera.cy"];

    cv::Mat K = cv::Mat::eye(3, 3, CV_32F);
    K.at<float>(0, 0) = fx;
```

```
    K.at<float>(1, 1) = fy;
    K.at<float>(0, 2) = cx;
    K.at<float>(1, 2) = cy;
    K.copyTo(mK);

    // ...
}

// 每传来一帧图像,就调用一次该函数
cv::Mat Tracking::GrabImageStereo(..., const cv::Mat &imRectLeft, const cv::Mat &imRectRight, const
double &timestamp) {
    mCurrentFrame = Frame(mImGray, mK, mDistCoef, mbf, mThDepth);

    Track();

    // ...
}

// Frame构造函数
Frame::Frame(cv::Mat &K, cv::Mat &distCoef, const float &bf, const float &thDepth)
    : mK(K.clone()), mDistCoef(distCoef.clone()), mbf(bf), mThDepth(thDepth) {

    // ...

    // 第一次调用Frame()构造函数时为所有static变量赋值
    if (mbInitialComputations) {
        fx = K.at<float>(0, 0);
        fy = K.at<float>(1, 1);
        cx = K.at<float>(0, 2);
        cy = K.at<float>(1, 2);
        invfx = 1.0f / fx;
        invfy = 1.0f / fy;

        // ...
        mbInitialComputations = false;      // 赋值完毕后将mbInitialComputations复位
    }

    mb = mbf / fx;
}
```

特征点提取

在`Frame`类构造函数中调用成员变量`mpORBextractorLeft`和`mpORBextractorRight`的`()`运算符进行特征点提取.

成员函数/变量	访问控制	意义
`ORBextractor* mpORBextractorLeft` `ORBextractor* mpORBextractorRight`	`public`	左右目图像的特征点提取器
`cv::Mat mDescriptors` `cv::Mat mDescriptorsRight`	`public`	左右目图像特征点描述子
`std::vector<cv::KeyPoint> mvKeys` `std::vector<cv::KeyPoint> mvKeysRight`	`public`	畸变矫正前的左/右目特征点
`std::vector<cv::KeyPoint> mvKeysUn`	`public`	畸变矫正后的左目特征点
`std::vector<float> mvuRight`	`public`	左目特征点在右目中匹配特征点的横坐标 (左右目匹配特征点的纵坐标相同)
`std::vector<float> mvDepth`	`public`	特征点深度
`float mThDepth`	`public`	判断单目特征点和双目特征点的阈值 深度低于该值得特征点被认为是双目特征点 深度低于该值得特征点被认为是单目特征点

`mvKeys`、`mvKeysUn`、`mvuRight`、`mvDepth`的坐标索引是对应的,也就是说对于第`i`个图像特征点:

- 其畸变矫正前的左目特征点是`mvKeys[i]`.
- 其畸变矫正后的左目特征点是`mvKeysUn[i]`.
- 其在右目图片中对应特征点的横坐标为`mvuRight[i]`,纵坐标与`mvKeys[i]`的纵坐标相同.
- 特征点的深度是`mvDepth[i]`.

对于单目特征点(**单目相机输入的特征点**或**没有找到右目匹配的左目图像特征点**),其`mvuRight`和`mvDepth`均为`-1`.

特征点提取: `ExtractORB()`

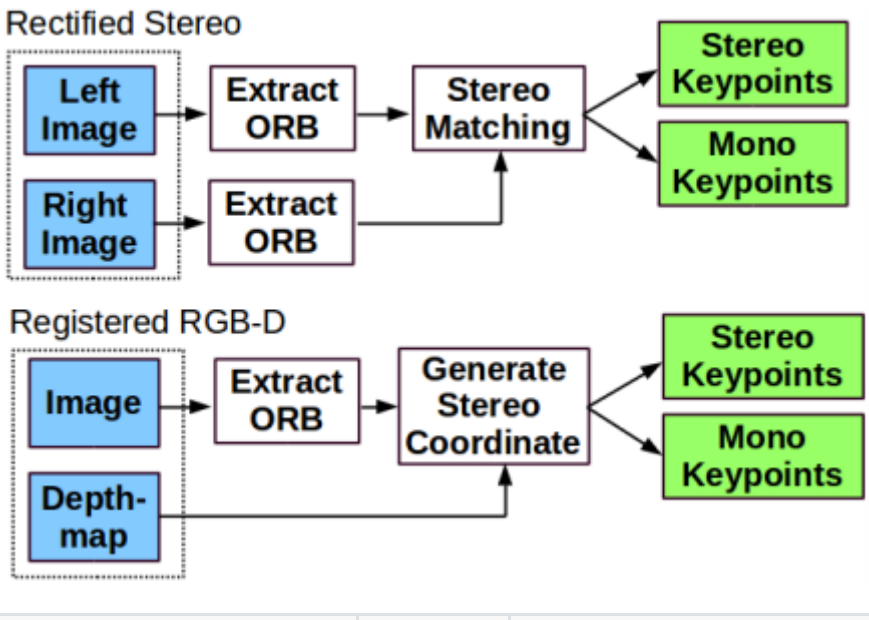
成员函数/变量	访问控制	意义
`void ExtractORB(int flag, const cv::Mat &im)`	`public`	进行`ORB`特征提取

```cpp
void Frame::ExtractORB(int flag, const cv::Mat &im) {
    if (flag == 0)      // flag==0, 表示对左图提取ORB特征点
        (*mpORBextractorLeft)(im, cv::Mat(), mvKeys, mDescriptors);
    else                // flag==1, 表示对右图提取ORB特征点
        (*mpORBextractorRight)(im, cv::Mat(), mvKeysRight, mDescriptorsRight);
}
```

ORB-SLAM2对双目/RGBD特征点的预处理

双目/RGBD相机中可以得到特征点的立体信息,包括**右目特征点信息**(`mvuRight`)、**特征点深度信息**(`mvDepth`)

- 对于双目相机,通过双目特征点匹配关系计算特征点的深度值.
- 对于RGBD相机,根据特征点深度构造虚拟的右目图像特征点.

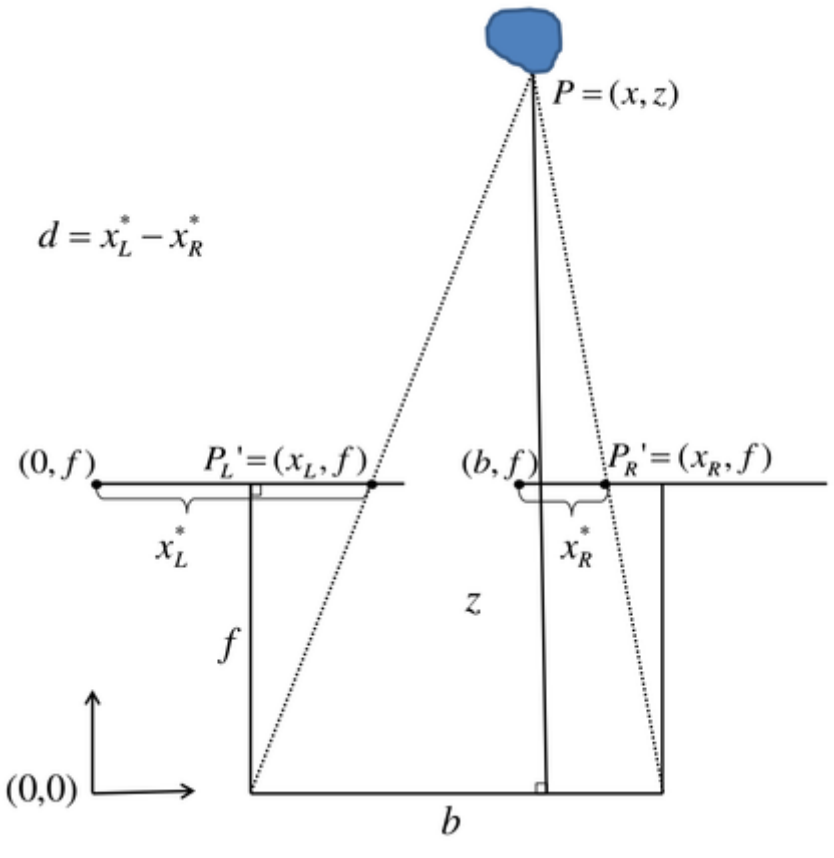


成员函数/变量	访问控制	意义
`void ComputeStereoMatches()`	`public`	双目图像特征点匹配,用于双目相机输入图像预处理
`void ComputeStereoFromRGBD(const cv::Mat &imDepth)`	`public`	根据深度信息构造虚拟右目图像,用于RGBD相机输入图像预处理
`cv::Mat UnprojectStereo(const int &i)`	`public`	根据深度信息将第`i`个特征点反投影成`MapPoint`

通过这种预处理,在后面SLAM系统的其他部分中**不再区分**双目特征点和RGBD特征点,它们以双目特征点的形式被处理.(仅通过判断`mvuRight[idx]`判断某特征点是否有深度).

```cpp
int ORBmatcher::SearchByProjection(Frame &F, const vector<MapPoint *> &vpMapPoints, const float th) {
    // ...
    for (size_t idx : vIndices) {
        if (F.mvuRight[idx] > 0) {      // 通过判断 mvuRight[idx] 判断该特征点是否有深度
            // 针对有深度的特征点特殊处理
        } else {
            // 针对单目特征点的特殊处理
        }
    }
    // ...
}
```

双目视差公式



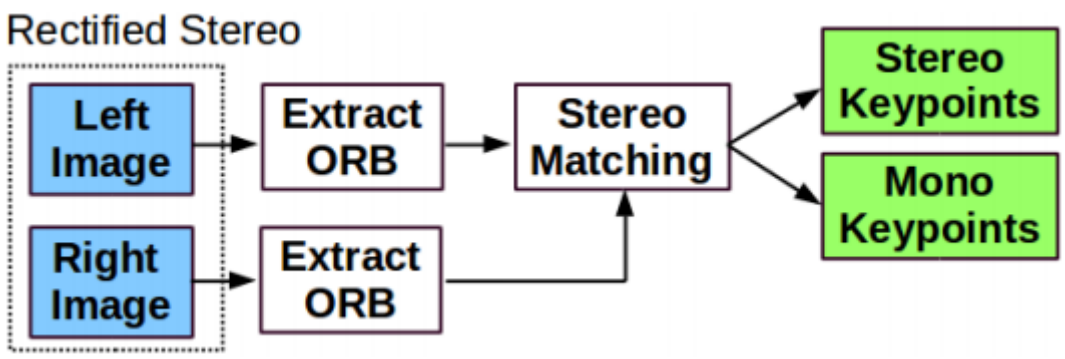
观测距离z,基线b,焦距f,视差d,根据三角形相似性:

$$\frac{z}{b}=\frac{z-f}{b-d}$$

得到:

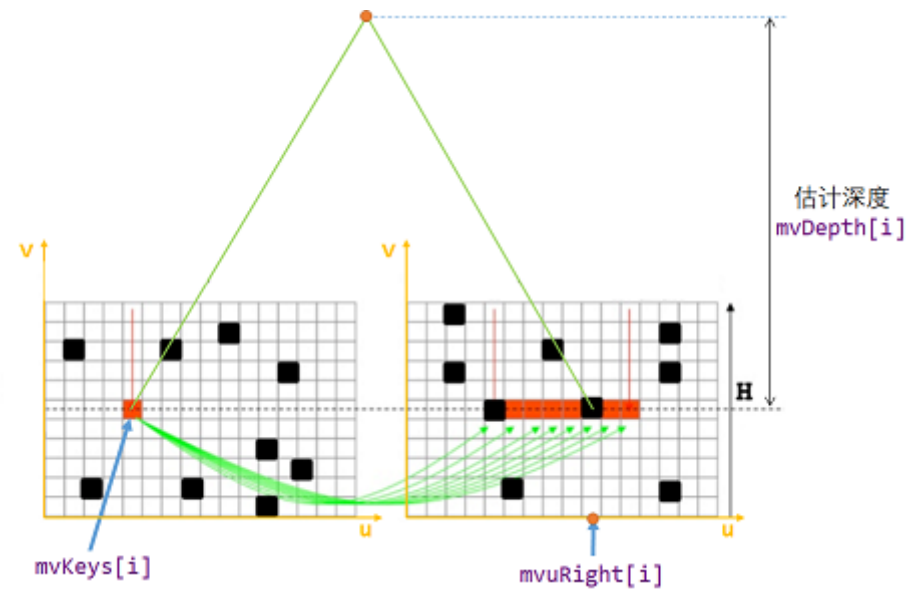
$$d=\frac{b\cdot f}{z}$$

双目特征点的处理:双目图像特征点匹配: `ComputeStereoMatches()`



双目相机分别提取到左右目特征点后对特征点进行双目匹配,并通过双目视差估计特征点深度.双目特征点匹配步骤:

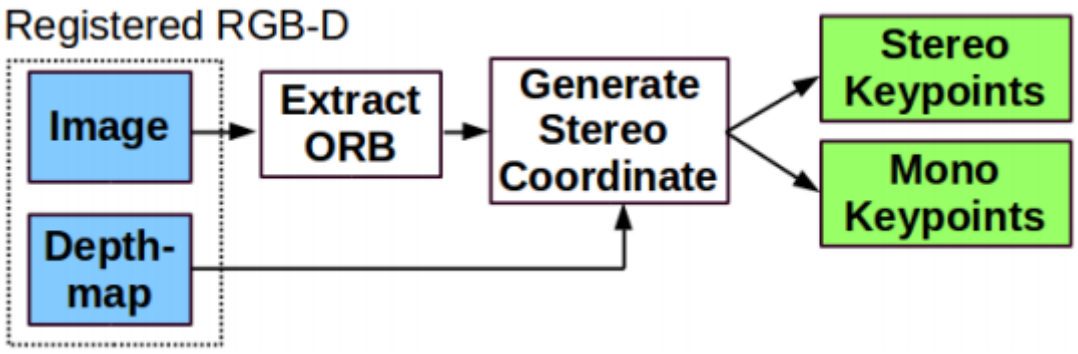
1. 粗匹配: 根据特征点**描述子距离**和**金字塔层级**判断匹配.粗匹配关系是按行寻找的,对于左目图像中每个特征点,在右目图像对应行上寻找匹配特征点.
2. 精匹配: 根据特征点**周围窗口内容相似度**判断匹配.
3. 亚像素插值: 将**特征点相似度**与**匹配坐标**之间拟合成二次曲线,寻找最佳匹配位置(得到的是一个小数).
4. 记录**右目匹配** `mvuRight` 和**深度** `mvDepth` 信息.
5. 离群点筛选: 以**平均相似度的2.1倍**为标准,筛选离群点.




```cpp
void Frame::ComputeStereoMatches() {

    mvuRight = vector<float>(N, -1.0f);
    mvDepth = vector<float>(N, -1.0f);

    // step0. 右目图像特征点逐行统计: 将右目图像中每个特征点注册到附近几行上
    vector<vector<size_t> > vRowIndices(nRows, vector<size_t>());    // 图像每行的1右目特征点索引
    for (int iR = 0; iR < mvKeysRight.size(); iR++) {
        const cv::KeyPoint &kp = mvKeysRight[iR];
        const float &kpY = kp.pt.y;
        const int maxr = ceil(kpY + 2.0f * mvScaleFactors[mvKeysRight[iR].octave]);
        const int minr = floor(kpY - 2.0f * mvScaleFactors[mvKeysRight[iR].octave]);
        for (int yi = minr; yi <= maxr; yi++)
            vRowIndices[yi].push_back(iR);
    }

    // step1. + 2. 粗匹配+精匹配
    const float minZ = mb, minD = 0, maxD = mbf / minZ;      // 根据视差公式计算两个特征点匹配搜索的范围
    const int thOrbDist = (ORBmatcher::TH_HIGH + ORBmatcher::TH_LOW) / 2;

    vector<pair<int, int> > vDistIdx;       // 保存特征点匹配

    for (int iL = 0; iL < N; iL++) {

        const cv::KeyPoint &kpL = mvKeys[iL];
        const int &levelL = kpL.octave;
        const float &vL = kpL.pt.y, &uL = kpL.pt.x;

        const vector<size_t> &vCandidates = vRowIndices[vL];
        if (vCandidates.empty()) continue;

        // step1. 粗匹配,根据特征点描述子和金字塔层级进行粗匹配
        int bestDist = ORBmatcher::TH_HIGH;
        size_t bestIdxR = 0;
        const cv::Mat &dL = mDescriptors.row(iL);
        for (size_t iC = 0; iC < vCandidates.size(); iC++) {

```

```
            const size_t iR = vCandidates[iC];
            const cv::KeyPoint &kpR = mvKeysRight[iR];
            if (kpR.octave < levelL - 1 || kpR.octave > levelL + 1)
                continue;
            const float &uR = kpR.pt.x;

            if (uR >= minU && uR <= maxU) {
                const cv::Mat &dR = mDescriptorsRight.row(iR);
                const int dist = ORBmatcher::DescriptorDistance(dL, dR);
                if (dist < bestDist) {
                    bestDist = dist;
                    bestIdxR = iR;
                }
            }
        }

        // step2. 精匹配: 滑动窗口匹配,根据匹配点周围5✕5窗口寻找精确匹配
        if (bestDist < thOrbDist) {
            const float uR0 = mvKeysRight[bestIdxR].pt.x;
            const float scaleFactor = mvInvScaleFactors[kpL.octave];
            const float scaleduL = round(kpL.pt.x * scaleFactor);
            const float scaledvL = round(kpL.pt.y * scaleFactor);
            const float scaleduR0 = round(uR0 * scaleFactor);
            const int w = 5;
            cv::Mat IL = mpORBextractorLeft->mvImagePyramid[kpL.octave].rowRange(scaledvL - w, scaledvL
+ w + 1).colRange(scaleduL - w, scaleduL + w + 1);
            IL.convertTo(IL, CV_32F);
            IL = IL - IL.at<float>(w, w) * cv::Mat::ones(IL.rows, IL.cols, CV_32F);
            int bestDist = INT_MAX;
            int bestincR = 0;
            const int L = 5;
            vector<float> vDists;
            vDists.resize(2 * L + 1);
            const float iniu = scaleduR0 + L - w;
            const float endu = scaleduR0 + L + w + 1;
            for (int incR = -L; incR <= +L; incR++) {
                cv::Mat IR = mpORBextractorRight->mvImagePyramid[kpL.octave].rowRange(scaledvL - w,
scaledvL + w + 1).colRange(scaleduR0 + incR - w, scaleduR0 + incR + w + 1);
                IR.convertTo(IR, CV_32F);
                IR = IR - IR.at<float>(w, w) * cv::Mat::ones(IR.rows, IR.cols, CV_32F);
                float dist = cv::norm(IL, IR, cv::NORM_L1);
                if (dist < bestDist) {
                    bestDist = dist;
                    bestincR = incR;
                }
                vDists[L + incR] = dist;
            }

            // step3. 亚像素插值: 将特征点匹配距离拟合成二次曲线,寻找二次曲线最低点(是一个小数)作为最优匹配点坐标
            const float dist1 = vDists[L + bestincR - 1];
            const float dist2 = vDists[L + bestincR];
            const float dist3 = vDists[L + bestincR + 1];
            const float deltaR = (dist1 - dist3) / (2.0f * (dist1 + dist3 - 2.0f * dist2));

            // step4. 记录特征点的右目和深度信息
            float bestuR = mvScaleFactors[kpL.octave] * ((float) scaleduR0 + (float) bestincR + deltaR);
            float disparity = (uL - bestuR);
            if (disparity >= minD && disparity < maxD) {
                mvDepth[iL] = mbf / disparity;
                mvuRight[iL] = bestuR;
                vDistIdx.push_back(pair<int, int>(bestDist, iL));
            }
        }
    }

    // step5. 删除离群点: 匹配距离大于平均匹配距离2.1倍的视为误匹配
    sort(vDistIdx.begin(), vDistIdx.end());
    const float median = vDistIdx[vDistIdx.size() / 2].first;
    const float thDist = 1.5f * 1.4f * median;
    for (int i = vDistIdx.size() - 1; i >= 0; i--) {
        if (vDistIdx[i].first < thDist)
            break;
        else {
            mvuRight[vDistIdx[i].second] = -1;
            mvDepth[vDistIdx[i].second] = -1;
        }
    }
}
```

RBGD特征点的处理: 根据深度信息构造虚拟右目图像: `ComputeStereoFromRGBD()`



对于RGB特征点,根据深度信息构造虚拟右目图像

```cpp
void Frame::ComputeStereoFromRGBD(const cv::Mat &imDepth) {
    // 初始化 右目 和 深度 信息
    mvuRight = vector<float>(N, -1);
    mvDepth = vector<float>(N, -1);

    for (int i = 0; i < N; i++) {
        const cv::KeyPoint &kp = mvKeys[i];
        const cv::KeyPoint &kpU = mvKeysUn[i];

        // 从未畸变矫正的深度图中获取深度信息,从校正过后的左图中获取特征点位置信息,构造虚拟右目
        const float d = imDepth.at<float>(kp.pt.y, kp.pt.x);
        if (d > 0) {
            mvDepth[i] = d;
            mvuRight[i] = kpU.pt.x - mbf / d;
        }
    }
}
```

畸变矫正: `UndistortKeyPoints()`

成员函数/变量	访问控制	意义
`cv::Mat mDistCoef`	`public`	相机的畸变矫正参数
`std::vector<cv::KeyPoint> mvKeys` `std::vector<cv::KeyPoint> mvKeysRight`	`public`	畸变矫正前的左/右目特征点
`std::vector<cv::KeyPoint> mvKeysUn`	`public`	畸变矫正后的左目特征点
`void UndistortKeyPoints()`	`private`	对所有特征点进行畸变矫正

`float mnMinX`
`float mnMaxX`
`float mnMinY`
`float mnMaxY`
	`public`	畸变矫正后的图像边界

`void ComputeImageBounds(const cv::Mat &imLeft)`	`private`	计算畸变矫正后的图像边界

实际上,畸变矫正只对单目和RGBD相机输入图像有效,双目相机的畸变矫正参数均为`0`,因为双目相机数据集在发布之前预先做了**双目矫正**.

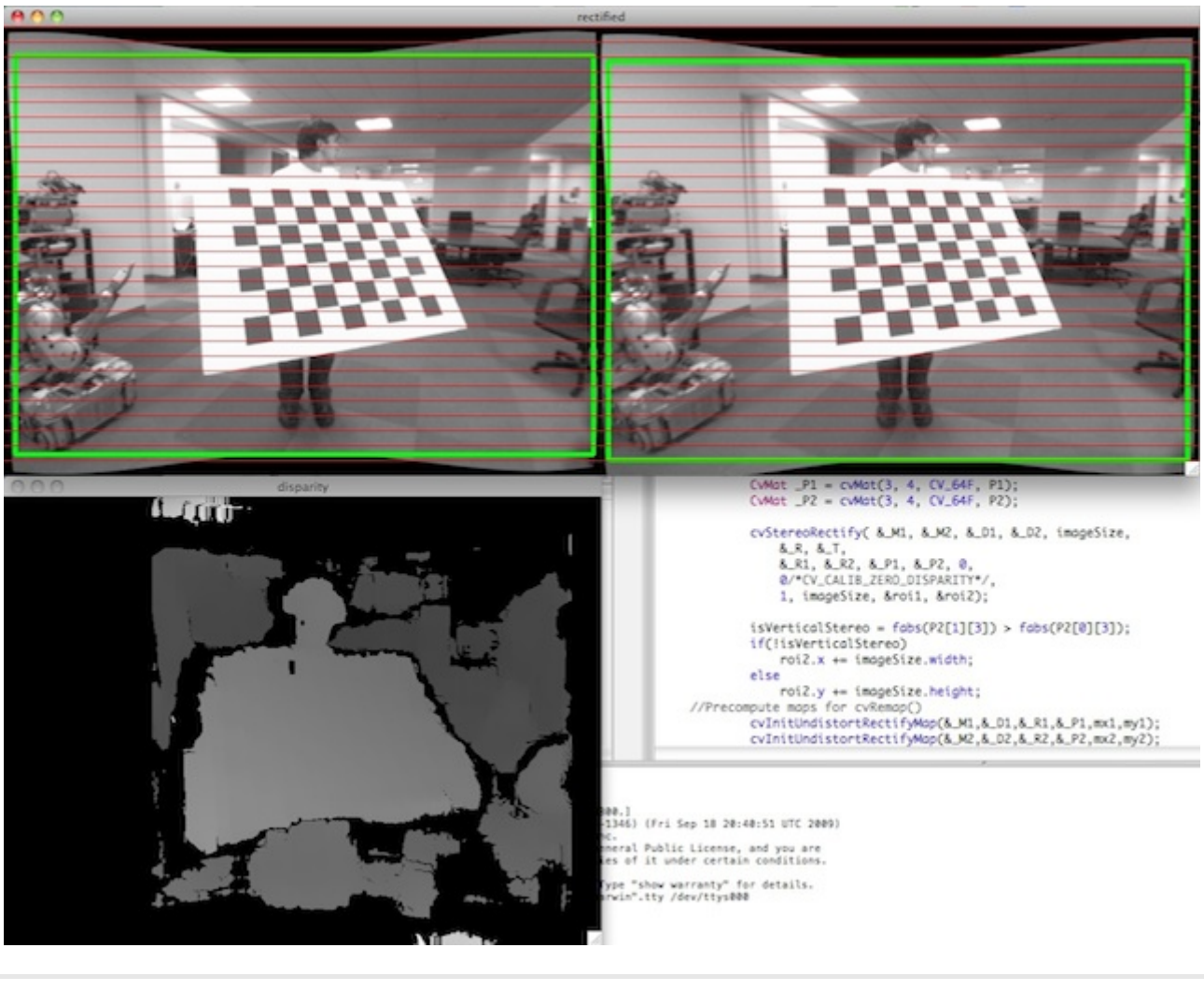
- RGBD相机输入配置文件`TUM1.yaml`

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

#....
```

- 双目相机输入配置文件`EuRoC.yaml`

```yaml
Camera.k1: 0.0
Camera.k2: 0.0
Camera.p1: 0.0
Camera.p2: 0.0

# ...
```

双目矫正效果如下,双目矫正将两个相机的成像平面矫正到同一平面上.双目矫正之后两个相机的极线相互平行,极点在无穷远处,这也是我们在函数`ComputeStereoMatches()`中做极线搜索的理论基础.



`UndistortKeyPoints()`函数和`ComputeImageBounds()`内调用了`cv::undistortPoints()`函数对特征点进行畸变矫正

```cpp
void Frame::UndistortKeyPoints() {
    // step1. 若输入图像是双目图像,则已做好了双目矫正,其畸变参数为0
    if (mDistCoef.at<float>(0) == 0.0) {
        mvKeysUn = mvKeys;
        return;
    }

    // 将特征点坐标转为undistortPoints()函数要求的格式
    cv::Mat mat(N, 2, CV_32F);
    for (int i = 0; i < N; i++) {
        mat.at<float>(i, 0) = mvKeys[i].pt.x;
        mat.at<float>(i, 1) = mvKeys[i].pt.y;
    }
    mat = mat.reshape(2);
    // 进行畸变矫正
    cv::undistortPoints(mat, mat, mK, mDistCoef, cv::Mat(), mK);

    // 记录校正后的特征点
    mat = mat.reshape(1);
    mvKeysUn.resize(N);
    for (int i = 0; i < N; i++) {
        cv::KeyPoint kp = mvKeys[i];
        mvKeysUn[i].pt.x = mat.at<float>(i, 0);
        mvKeysUn[i].pt.y = mat.at<float>(i, 1);
    }
}

// 通过计算图片顶点畸变矫正后的坐标来计算畸变矫正后的图片有效范围
void Frame::ComputeImageBounds(const cv::Mat &imLeft) {
    if (mDistCoef.at<float>(0) != 0.0) {
        // 4个顶点坐标
        cv::Mat mat(4, 2, CV_32F);
        mat.at<float>(0, 0) = 0.0;         //左上
        mat.at<float>(0, 1) = 0.0;
        mat.at<float>(1, 0) = imLeft.cols; //右上
        mat.at<float>(1, 1) = 0.0;
        mat.at<float>(2, 0) = 0.0;         //左下
        mat.at<float>(2, 1) = imLeft.rows;
        mat.at<float>(3, 0) = imLeft.cols; //右下
        mat.at<float>(3, 1) = imLeft.rows;
        // 畸变矫正
        mat = mat.reshape(2);
        cv::undistortPoints(mat, mat, mK, mDistCoef, cv::Mat(), mK);
        mat = mat.reshape(1);
        // 记录图片有效范围
        mnMinX = min(mat.at<float>(0, 0), mat.at<float>(2, 0));     //左上和左下横坐标最小的
        mnMaxX = max(mat.at<float>(1, 0), mat.at<float>(3, 0));     //右上和右下横坐标最大的
        mnMinY = min(mat.at<float>(0, 1), mat.at<float>(1, 1));     //左上和右上纵坐标最小的
        mnMaxY = max(mat.at<float>(2, 1), mat.at<float>(3, 1));     //左下和右下纵坐标最小的
    } else {
        mnMinX = 0.0f;
        mnMaxX = imLeft.cols;
        mnMinY = 0.0f;
```

```
        mnMaxY = imLeft.rows;
    }
}
```

特征点分配: `AssignFeaturesToGrid()`

在对特征点进行预处理后,将特征点分配到`48`行`64`列的网格中以加速匹配

成员函数/变量	访问控制	意义
`FRAME_GRID_ROWS=48` `FRAME_GRID_COLS=64`	`#DEFINE`宏	网格行数/列数
`float mfGridElementWidthInv` `float mfGridElementHeightInv`	`public static` `public static`	每个网格的宽度/高度
`std::vector<std::size_t> mGrid[FRAME_GRID_COLS][FRAME_GRID_ROWS]`	`public`	每个网格内特征点编号列表
`void AssignFeaturesToGrid()`	`private`	将特征点分配到网格中
`vector<size_t> GetFeaturesInArea(float &x, float &y, float &r, int minLevel, int maxLevel)`	`public`	获取半径为`r`的圆域内的特征点编号列表

成员变量`std::vector<std::size_t> mGrid[FRAME_GRID_COLS][FRAME_GRID_ROWS]`是一个二维数组,数组中每个元素是对应网格的所有特征点索引列表.

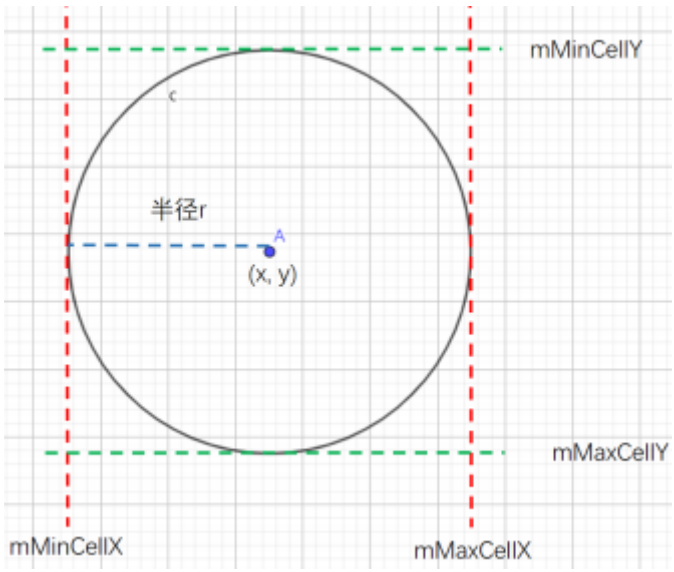
`static`成员变量`mfGridElementWidthInv`、`mfGridElementHeightInv`表示网格宽度/高度,它们在第一次调用`Frame`构造函数时被计算赋值.

```
// Frame构造函数
Frame::Frame(cv::Mat &K, cv::Mat &distCoef, const float &bf, const float &thDepth)
    : mK(K.clone()), mDistCoef(distCoef.clone()), mbf(bf), mThDepth(thDepth) {

    // ...

    ComputeImageBounds(imGray);     // 计算去畸变后图像的边界
    // 计算特征点网格宽度/高度
    mfGridElementWidthInv = static_cast<float>(FRAME_GRID_COLS) / static_cast<float>(mnMaxX - mnMinX);
    mfGridElementHeightInv = static_cast<float>(FRAME_GRID_ROWS) / static_cast<float>(mnMaxY - mnMinY);

    // 第一次调用Frame()构造函数时为所有static变量赋值
    if (mbInitialComputations) {
        fx = K.at<float>(0, 0);
        fy = K.at<float>(1, 1);
        cx = K.at<float>(0, 2);
        cy = K.at<float>(1, 2);
        invfx = 1.0f / fx;
        invfy = 1.0f / fy;

        mbInitialComputations = false;      // 赋值完毕后将mbInitialComputations复位
    }

    mb = mbf / fx;
}
```

函数`AssignFeaturesToGrid()`将特征点分配到网格中

```
void Frame::AssignFeaturesToGrid() {
    for (int i = 0; i < N; i++) {
        // 遍历特征点,将每个特征点索引加入到对应网格中
        const cv::KeyPoint &kp = mvKeysUn[i];
        int nGridPosX, nGridPosY;
        if (PosInGrid(kp, nGridPosX, nGridPosY))
            mGrid[nGridPosX][nGridPosY].push_back(i);
    }
}
```

函数`vector<size_t> GetFeaturesInArea(float &x, float &y, float &r, int minLevel, int maxLevel)`获取点`(y,x)`周围半径为`r`的圆域内所有特征点编号.



构造函数: `Frame()`

`Frame()`构造函数依次进行上面介绍的步骤:

```
// 双目相机Frame构造函数
Frame::Frame(const cv::Mat &imLeft, const cv::Mat &imRight, const double &timeStamp, ORBextractor *extractorLeft, ORBextractor *extractorRight, ORBVocabulary *voc, cv::Mat &K, cv::Mat &distCoef, const float &bf, const float &thDepth)
    : mpORBvocabulary(voc), mpORBextractorLeft(extractorLeft), mpORBextractorRight(extractorRight), mTimeStamp(timeStamp), mK(K.clone()), mDistCoef(distCoef.clone()), mbf(bf), mThDepth(thDepth), mpReferenceKF(static_cast<KeyFrame *>(NULL)) {



    // step0. 帧ID自增
    mnId = nNextId++;

    // step1. 计算金字塔参数
    mnScaleLevels = mpORBextractorLeft->GetLevels();
    mfScaleFactor = mpORBextractorLeft->GetScaleFactor();
    mfLogScaleFactor = log(mfScaleFactor);
    mvScaleFactors = mpORBextractorLeft->GetScaleFactors();
    mvInvScaleFactors = mpORBextractorLeft->GetInverseScaleFactors();
    mvLevelSigma2 = mpORBextractorLeft->GetScaleSigmaSquares();
    mvInvLevelSigma2 = mpORBextractorLeft->GetInverseScaleSigmaSquares();

    // step2. 提取双目图像特征点
    thread threadLeft(&Frame::ExtractORB, this, 0, imLeft);
    thread threadRight(&Frame::ExtractORB, this, 1, imRight);
    threadLeft.join();
    threadRight.join();

    N = mvKeys.size();
    if (mvKeys.empty())
        return;

    // step3. 畸变矫正,实际上UndistortKeyPoints()不对双目图像进行矫正
    UndistortKeyPoints();

    // step4. 双目图像特征点匹配
    ComputeStereoMatches();


    // step5. 第一次调用构造函数时计算static变量
    if (mbInitialComputations) {
        ComputeImageBounds(imLeft);
        mfGridElementWidthInv = static_cast<float>(FRAME_GRID_COLS) / static_cast<float>(mnMaxX - mnMinX);
        mfGridElementHeightInv = static_cast<float>(FRAME_GRID_ROWS) / static_cast<float>(mnMaxY - mnMinY);
        fx = K.at<float>(0, 0);
        fy = K.at<float>(1, 1);
        cx = K.at<float>(0, 2);
        cy = K.at<float>(1, 2);
        invfx = 1.0f / fx;
        invfy = 1.0f / fy;

        // 计算完成,标志复位
        mbInitialComputations = false;
    }

    mvpMapPoints = vector<MapPoint *>(N, static_cast<MapPoint *>(NULL));    // 初始化本帧的地图点
    mvbOutlier = vector<bool>(N, false);    // 标记当前帧的地图点不是外点
    mb = mbf / fx;      // 计算双目基线长度

    // step6. 将特征点分配到网格中
    AssignFeaturesToGrid();
}
```

```cpp
// RGBD相机Frame构造函数
Frame::Frame(const cv::Mat &imGray, const cv::Mat &imDepth, const double &timeStamp, ORBextractor *extractor, ORBVocabulary *voc, cv::Mat &K, cv::Mat &distCoef, const float &bf, const float &thDepth)
    : mpORBvocabulary(voc), mpORBextractorLeft(extractor), mpORBextractorRight(static_cast<ORBextractor*>(NULL)), mTimeStamp(timeStamp), mK(K.clone()), mDistCoef(distCoef.clone()), mbf(bf), mThDepth(thDepth)
{
    // step0. 帧ID自增
    mnId = nNextId++;

    // step1. 计算金字塔参数
    mnScaleLevels = mpORBextractorLeft->GetLevels();
    mfScaleFactor = mpORBextractorLeft->GetScaleFactor();
    mfLogScaleFactor = log(mfScaleFactor);
    mvScaleFactors = mpORBextractorLeft->GetScaleFactors();
    mvInvScaleFactors = mpORBextractorLeft->GetInverseScaleFactors();
    mvLevelSigma2 = mpORBextractorLeft->GetScaleSigmaSquares();
    mvInvLevelSigma2 = mpORBextractorLeft->GetInverseScaleSigmaSquares();

    // step2. 提取左目图像特征点
    ExtractORB(0, imGray);

    N = mvKeys.size();
    if (mvKeys.empty())
        return;

    // step3. 畸变矫正
    UndistortKeyPoints();

    // step4. 根据深度信息构造虚拟右目图像
    ComputeStereoFromRGBD(imDepth);

    mvpMapPoints = vector<MapPoint *>(N, static_cast<MapPoint *>(NULL));
    mvbOutlier = vector<bool>(N, false);

    // step5. 第一次调用构造函数时计算static变量
    if (mbInitialComputations) {
        ComputeImageBounds(imLeft);
        mfGridElementWidthInv = static_cast<float>(FRAME_GRID_COLS) / static_cast<float>(mnMaxX - mnMinX);
        mfGridElementHeightInv = static_cast<float>(FRAME_GRID_ROWS) / static_cast<float>(mnMaxY - mnMinY);
        fx = K.at<float>(0, 0);
        fy = K.at<float>(1, 1);
        cx = K.at<float>(0, 2);
        cy = K.at<float>(1, 2);
        invfx = 1.0f / fx;
        invfy = 1.0f / fy;

        // 计算完成,标志复位
        mbInitialComputations = false;
    }

    mvpMapPoints = vector<MapPoint *>(N, static_cast<MapPoint *>(NULL));    // 初始化本帧的地图点
    mvbOutlier = vector<bool>(N, false);    // 标记当前帧的地图点不是外点
    mb = mbf / fx;      // 计算双目基线长度


    // step6. 将特征点分配到网格中
    AssignFeaturesToGrid();
}
```

`Frame`类的用途

`Tracking`类有两个`Frame`类型的成员变量

成员函数/变量	访问控制	意义
`Frame mCurrentFrame`	`public`	当前正在处理的帧
`Frame mLastFrame`	`protected`	上一帧

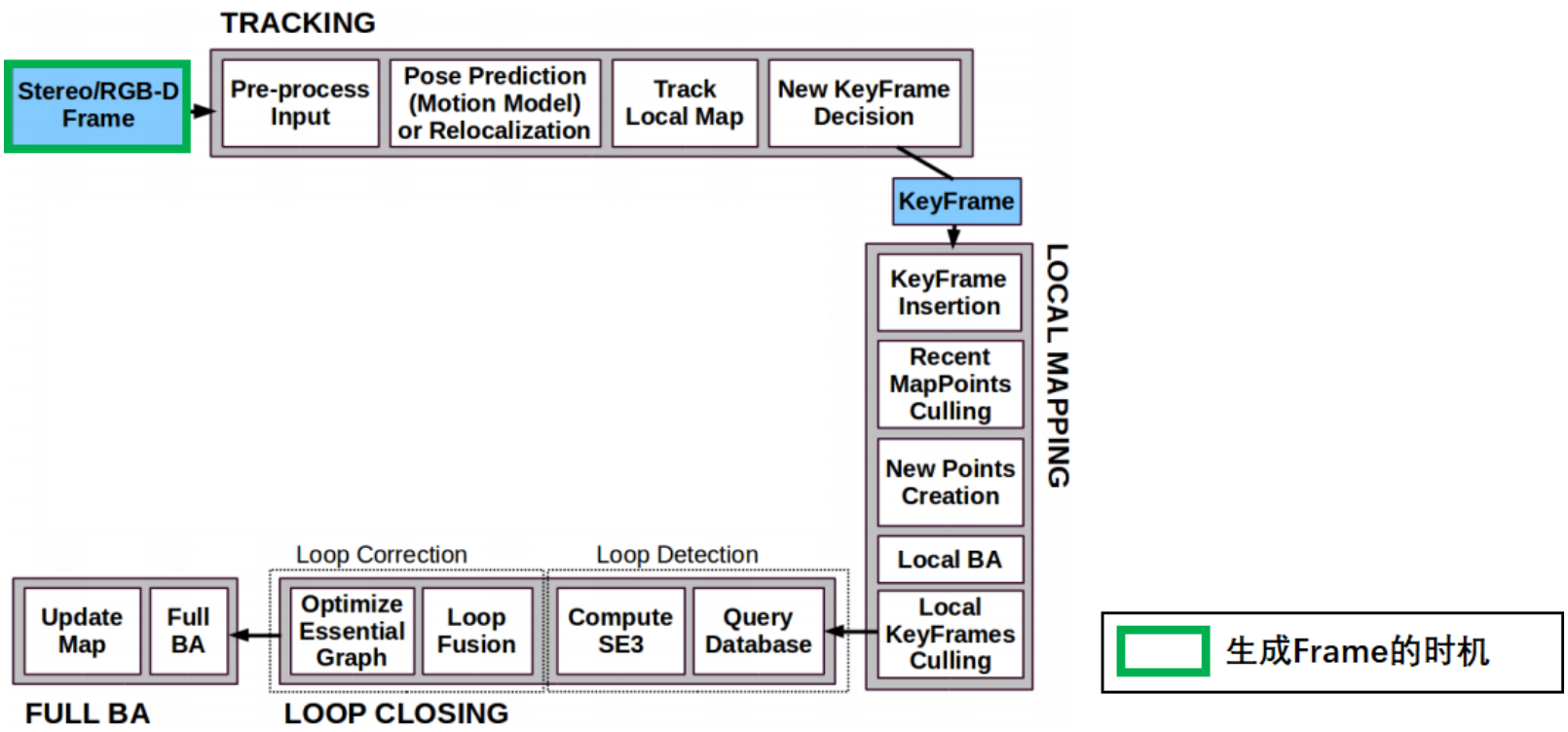
`Tracking`线程每收到一帧图像,就调用函数`Tracking::GrabImageMonocular()`、`Tracking::GrabImageStereo()`或`Tracking::GrabImageRGBD()`创建一个`Frame`对象,赋值给`mCurrentFrame`.

```cpp
// 每传来一帧图像,就调用一次这个函数
cv::Mat Tracking::GrabImageMonocular(const cv::Mat &im, const double &timestamp) {
    mImGray = im;
    // 图像通道转换
    if (mImGray.channels() == 3) {
        if (mbRGB)
            cvtColor(mImGray, mImGray, CV_RGB2GRAY);
        else
            cvtColor(mImGray, mImGray, CV_BGR2GRAY);
```

```
    } else if (mImGray.channels() == 4) {
        if (mbRGB)
            cvtColor(mImGray, mImGray, CV_RGBA2GRAY);
        else
            cvtColor(mImGray, mImGray, CV_BGRA2GRAY);
    }

    // 构造Frame
    if (mState == NOT_INITIALIZED || mState == NO_IMAGES_YET) //没有成功初始化的前一个状态就是NO_IMAGES_YET
        mCurrentFrame = Frame(mImGray, timestamp, mpIniORBextractor, mpORBVocabulary, mK, mDistCoef, mbf, mThDepth);
    else
        mCurrentFrame = Frame(mImGray, timestamp, mpORBextractorLeft, mpORBVocabulary, mK, mDistCoef, mbf, mThDepth);

    // 跟踪
    Track();

    // 返回当前帧的位姿
    return mCurrentFrame.mTcw.clone();
}
```

`Track()`函数跟踪结束后,会将`mCurrentFrame`赋值给`mLastFrame`

```
void Tracking::Track() {

    // 进行跟踪
    // ...

    // 将当前帧记录为上一帧
    mLastFrame = Frame(mCurrentFrame);

    // ...
}
```



除了少数被选为`KeyFrame`的帧以外,大部分`Frame`对象的作用仅在于`Tracking`线程内追踪当前帧位姿,不会对`LocalMapping`线程和`LoopClosing`线程产生任何影响,在`mLastFrame`和`mCurrentFrame`更新之后就被系统销毁了.