扩展跟踪工作总结

`ARToolKit` `ORB-SLAM2` 扩展跟踪

目录

1. 扩展跟踪程序编译及运行

1.1 准备工作

- 硬件环境：Ubuntu16.04
- 编译器：gcc, g++(C++11), Cmake（2.8以上）
- 环境依赖
  由于本程序是由ARToolKit工程与ORB-SLAM2工程结合形成的，所以本程序的库依赖为ARToolKit和ORB-SLAM2库依赖的并集，关于两者库依赖的添加，可参考ARToolKit5-

github以及ORB-SLAM2-github。
如果遇到问题，也可以参考`编译过程中可能遇到的问题.txt`

1.2 程序编译

在Linux终端下，使用`cd`命令进入到`Extended-Tracking`根目录下，输入：

```
chmod +x build.sh
./build.sh
```

即可自动完成整个扩展跟踪程序的编译工作；其编译流程为：
1. 进入`Thirdparty/DBoW2`目录下编译ORB-SLAM2部分所需的DBoW2库。
2. 进入`Thirdparty/g2o`目录下编译ORB-SLAM2部分所需的g2o库。
3. 进入`lib/SRC`目录下根据`makefile`文件编译ARToolKit部分所需的相关库。
4. 根据根目录下的`CmakeLists.txt`文件先编译ORB-SLAM2部分所需的共享库`libORB_SLAM2.so`，随后编译扩展跟踪程序示例。

编译完成后，会在`Thirdparty/DBoW2/lib`下得到DBoW2库，在`Thirdparty/g2o/lib`下得到g2o库，在`lib`下得到ARToolKit相关库和ORB-SLAM2共享库`libORB_SLAM2.so`，在`bin`下得到扩展跟踪程序示例的可执行文件`extended-tracking`。

1.3 程序运行

1. 将`相关参考材料`目录下的`pinball.jpg`图片文件打印到A4纸上作为程序的自然图像模板。
2. 打开命令行终端，输入以下代码并回车：

   ```
   export ARTOOLKIT5_VCONF="-device=LinuxV4L2"
   ```

   该代码是为了指定ARToolKit调用的摄像头驱动类型。
3. 在`Extended-Tracking`根目录下，进入到`bin`目录：

   ```
   cd bin
   ```

   运行程序：

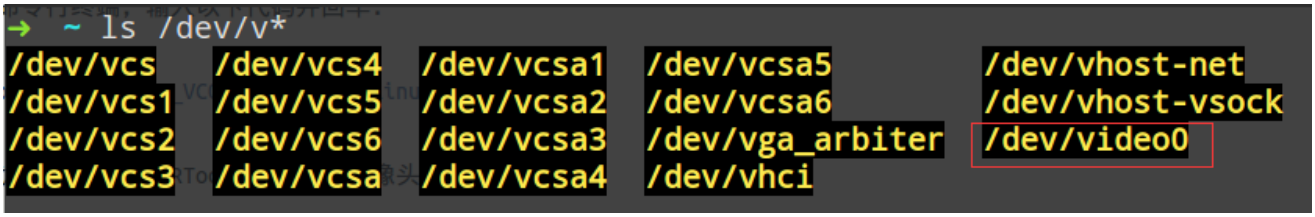
   ```
   ./extended-tracking Vocabulary/ORBvoc.txt TUM1.yaml
   ```

其中`Vocabulary/ORBvoc.txt`参数为ORB字典文件的路径，`TUM1.yaml`参数保存的是相机内参和ORB-SLAM2程序所需的相关设置信息；这两个参数主要供程序的ORB-SLAM2部分初始化使用。

4. 如果想让程序调用USB摄像头，则应使接入的USB摄像头的设备号为**video0**（ARToolKit使用video0设备作为摄像头），可以使用如下命令查看当前的摄像头设备：

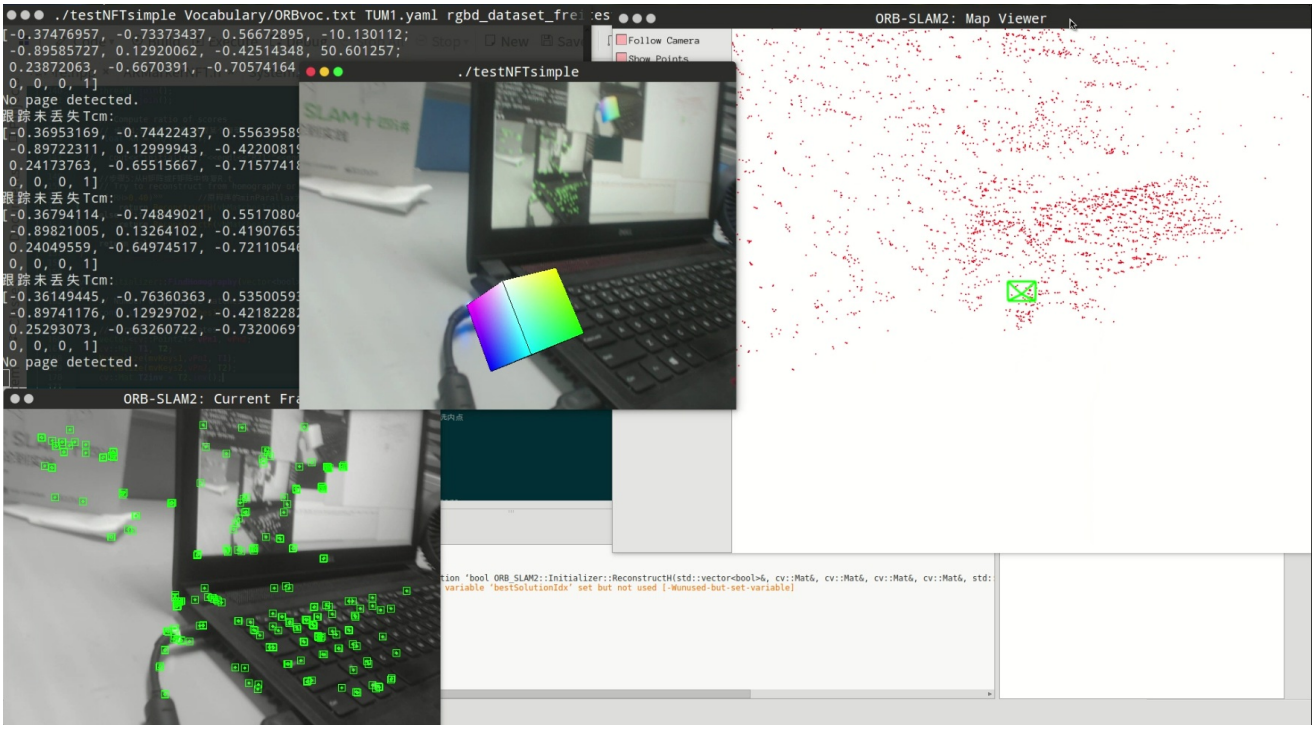
```
ls /dev/v*
```

将在终端显示如下信息：



如果是笔记本电脑，则Linux系统默认将内置的前置摄像头作为**video0**设备，在电脑正常运行情况下接入USB摄像头，此时的USB摄像头会成为**video1**设备。为了将USB摄像头设置成**video0**设备，可以在电脑关机状态下插入USB摄像头再开机，开机后USB摄像头会被自动设置为**video0**设备。

5. 程序运行效果截图如下：



完整演示视频可以参考`相关参考材料`目录下的`扩展跟踪演示.mp4`文件。

1.4 程序清理

在`Extended-Tracking`根目录下，输入：

```
chmod +x clean.sh
./clean.sh
```

即可清理在编译过程中产生的所有中间文件、库文件以及可执行文件。

1.5 其他相关问题

- 如何制作并训练自定义的自然图像模板？
  可参考`相关参考文件/ARToolKit for unity使用整理/ARToolkit for unity使用整理.pdf`文件中的相关内容，在得到`.fset`，`.fset3`，`.iset`文件后，将其放入`Extended-Tracking/bin/DataDFT`替换掉相应格式的文件，随后修改`bin/Data2/markers.dat`文件中的相关信息即可。
- 如何标定当前相机的内参信息？
  可参考`相关参考文件/ARToolKit for unity使用整理/ARToolkit for unity使用整理.pdf`文件中的相关内容，在得到`camera_para.dat`文件后，将其放入`bin/Data2`替换掉同名文件，随后再根据标定结果修改`bin/TUM1.yaml`文件中的内参信息即可。

需要注意的是，以上过程所涉及的应用程序可以在**Windows**操作系统下运行`相关参考材料/ARToolKit各版本（官网下载）/ARToolkit5_windows/ARToolKit5/bin`目录下的相关程序来完成。

2. 扩展跟踪程序设计思路及流程分析

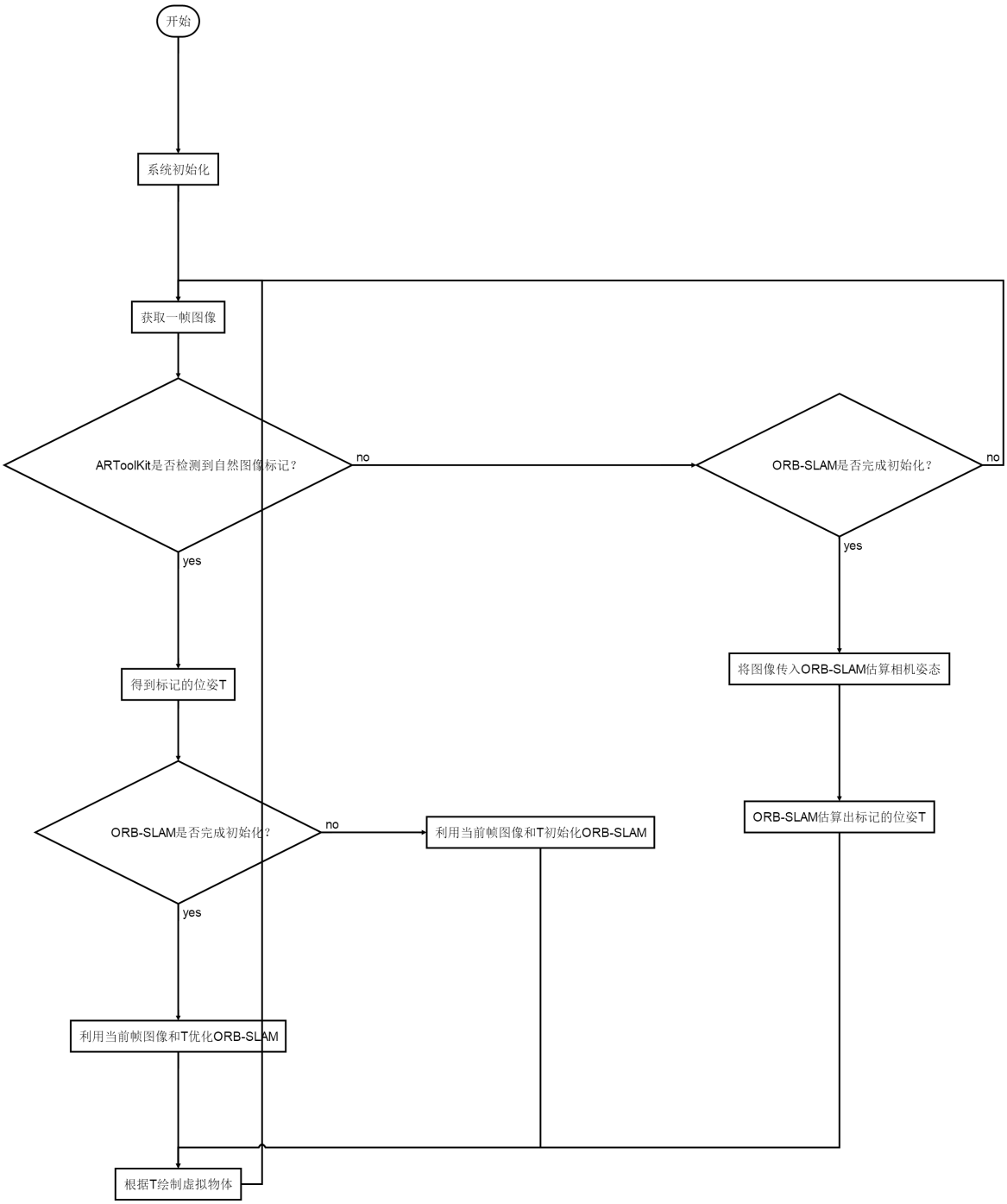
本部分将详细介绍扩展跟踪程序的设计思路；包括在对ARToolKit和ORB-SLAM2进行代码分析前，根据扩展跟踪功能需求所设计出的程序总体流程；随后深入代码层面，分析ARToolKit基于自然图像特征跟踪注册的示例代码以及ORB-SLAM2单目程序的示例代码，了解其程序的处理流程；最后，在此基础上，根据ARToolKit与ORB-SLAM的程序特性对扩展跟踪程序进行详细设计。

2.1 扩展跟踪程序总体设计

由Vuforia的扩展跟踪示例可以得知，扩展跟踪的实现主要结合了两个引擎：

1. 基于自然图像特征跟踪注册的AR引擎
2. 单目视觉SLAM引擎

由于需要使用单目视觉SLAM引擎，所以必然会遇到AR引擎与单目视觉SLAM引擎之间如何进行尺度统一的问题（即单目视觉SLAM引擎的初始化问题）。如今对于AR引擎部分，我们采用的方案是ARToolKit；对于单目视觉SLAM引擎，我们采用的方案是ORB-SLAM2；因此，从扩展跟踪的功能需求出发，我们能够得到扩展跟踪程序总体设计的思路如下：



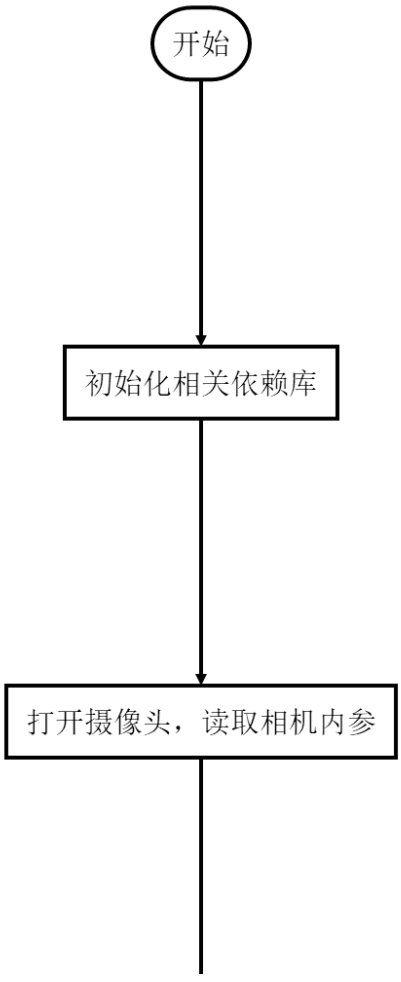
以上流程图大致描述了如下的处理流程：

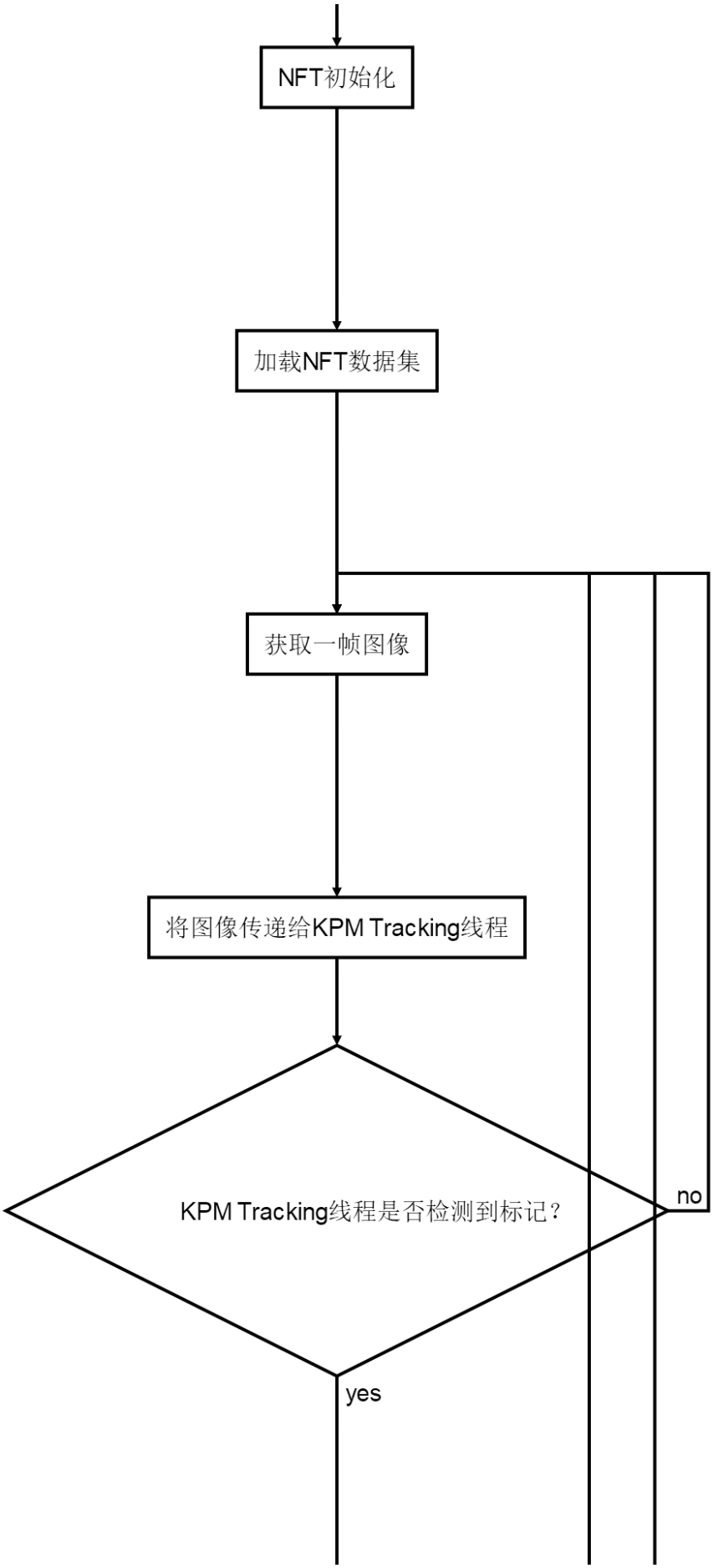
1. 首先，程序进行系统初始化，这里主要是为ARToolKit和ORB-SLAM系统做相关初始化设置。
2. 系统初始化完成后，打开相机读取并依次处理每一帧图像。
3. 获取到的图像帧先进入ARToolKit系统，此时ARToolKit可以根据当前图像帧的内容进行检测并查看是否存在事先导入的自然图像标记；若检测到标记，则计算出标记的位姿；否则，查看ORB-SLAM系统是否初始化完成；若完成，则将图像帧递交给ORB-SLAM系统计算；否则说明ORB-SLAM还未进行初始化，不能接收图像帧，此时需要进行下一帧的读取。

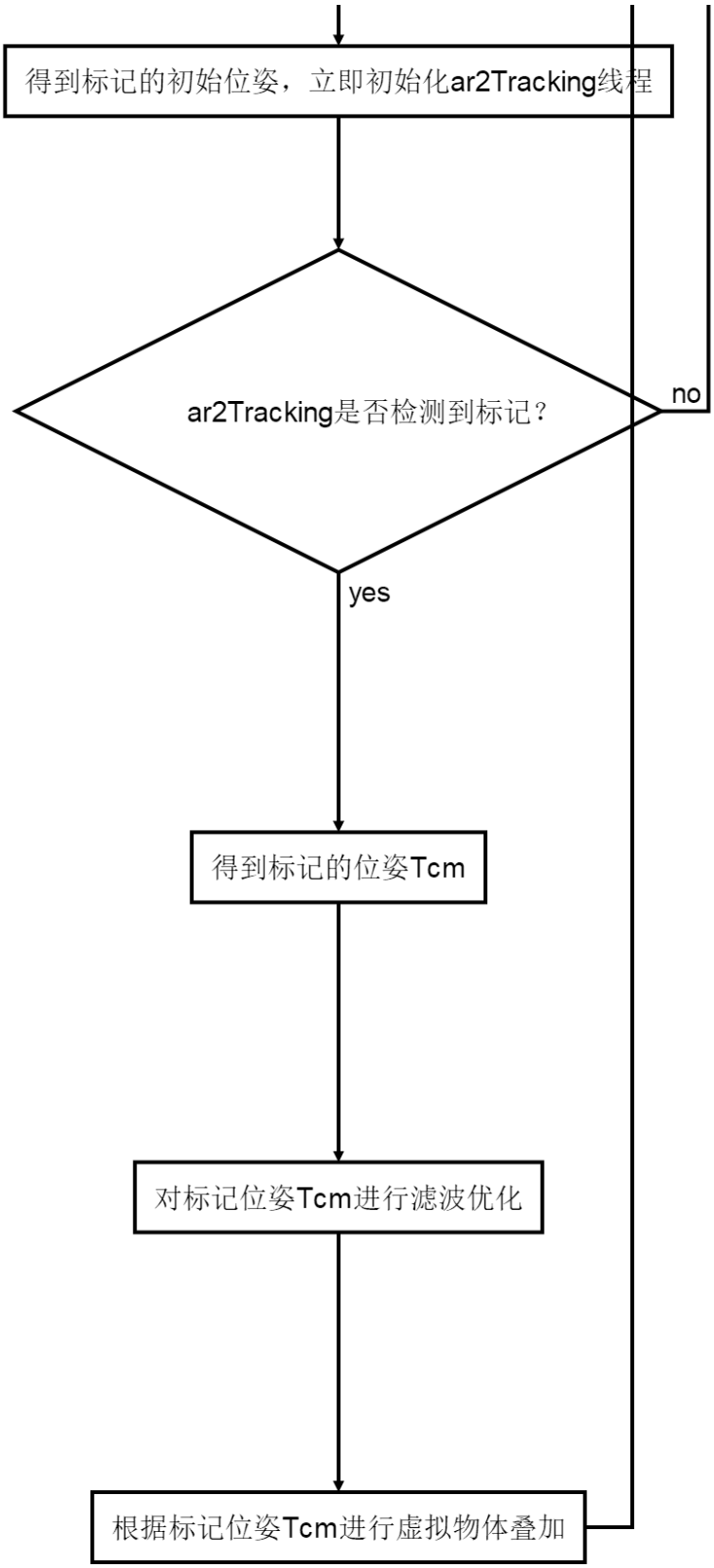
4. 通过ARToolKit得到标记的位姿后，此时判断是否需要初始化ORB-SLAM系统，若ORB-SLAM系统还未初始化，则将标记位姿和图像帧传入ORB-SLAM进行初始化；若ORB-SLAM系统初始化完毕，此时则利用当前帧图像和T优化ORB-SLAM的计算结果。
5. 无论是通过ARToolKit还是ORB-SLAM系统得到标记位姿后，将其递交给渲染模块进行三维虚拟物体的渲染叠加。
6. 虚拟物体叠加完成后，重新获取新的图像帧，重复执行前述跟踪流程。

2.2 基于自然特征跟踪注册的ARToolKit示例流程分析

关于基于自然特征跟踪注册的程序示例，ARToolKit有`nftsimple`和`nftbook`，可在`相关参考材料/ARToolKit各版本（官网下载）/ARToolKit5-bin-5.3.2r1-Linux-x86_64/examples`目录下找到其源代码，本人在扩展跟踪程序开发时，参考了`nftsimple`示例，并在其上融合了ORB-SLAM进行开发。因此本部分将分析`nftsimple`程序示例，其程序流程大致如下：




NFT初始化
加载NFT数据集
获取一帧图像
将图像传递给KPM Tracking线程
KPM Tracking线程是否检测到标记？
no
yes



流程描述如下：

1. 首先，ARToolKit初始化相关的依赖库，其中主要是与OpenGL库的初始化有关，因为ARToolKit使用OpenGL进行虚拟物体的绘制与显示。

2. 随后，ARToolKit读取相机参数文件camera_para.dat（该文件位于`bin/Data2`目录下），将相机的内参信息保存到相关结构体中。

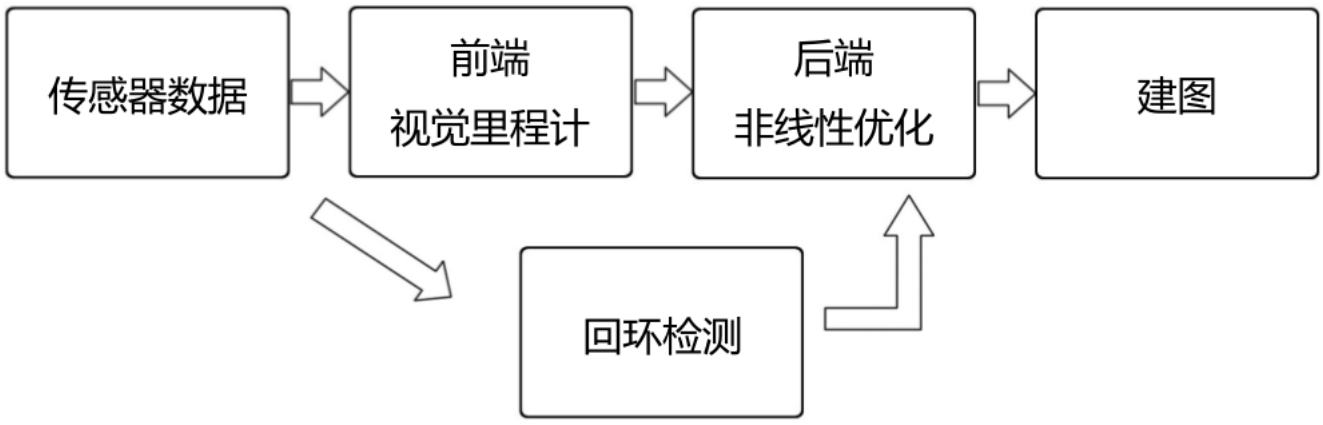
3. ARToolKit进行NFT（自然特征追踪）初始化工作;ARToolKit在对自然图像标记的追踪过程中，将主要开启两个线程：KPM Tracking和AR2Tracking，该步的作用主要是为了分配并初始化KPM Tracking和AR2Tracking需要用到的结构体kpmHandle和ar2Handle。
4. NFT初始化完成后，ARToolKit开始加载nft数据集，该数据集通过后缀名为`.fset`和`.fset3`的文件读入，这些文件用于向ARToolKit指示需要追踪的自然图像标记的特征信息；数据集加载完毕后，ARToolKit将立即开启KPM Tracking和AR2Tracking线程并等待系统的处理信号。
5. 当所有的初始化工作完成后，ARToolKit进入主循环（循环调用`mainloop`函数）开始追踪自然图像标记；首先ARToolKit从摄像头处获取一帧图像，将其传递给KPM Tracking线程，并向其发出开始工作信号。
6. KPM Tracking线程收到传来的图像帧和开始信号后，开始对当前帧图像提取特征点并计算其描述子，通过与加载的数据集信息进行比对来判断当前帧内是否含有自然图像标记，如果有，则计算出**自然图像标记坐标系到当前相机坐标系**的初始变换矩阵，即标记的位姿；如果没有，则获取下一帧图像并重复之前的工作。
7. 当KPM Tracking线程获取到自然图像标记的初始位姿后，将立即使用该位姿初始化AR2Tracking，随后将当前帧图像和初始化位姿一起传递给AR2Tracking，使其执行后续的标记跟踪工作。
8. 若AR2Tracking线程在跟踪过程中发现标记丢失，则将退出该线程并重新获取新的图像帧，并等到下一次KPM Tracking线程跟踪标记成功以唤醒AR2Tracking线程；否则，AR2Tracking跟踪当前图像帧成功并返回准确的标记位姿。
9. 得到标记位姿后，ARToolKit将会对该位姿进行滤波优化，去除抖动；并最终将优化后的位姿传递给绘图程序进行虚拟物体的叠加，以将其放到视野中的正确位置上。

`nftsimple`程序执行效果如下图：

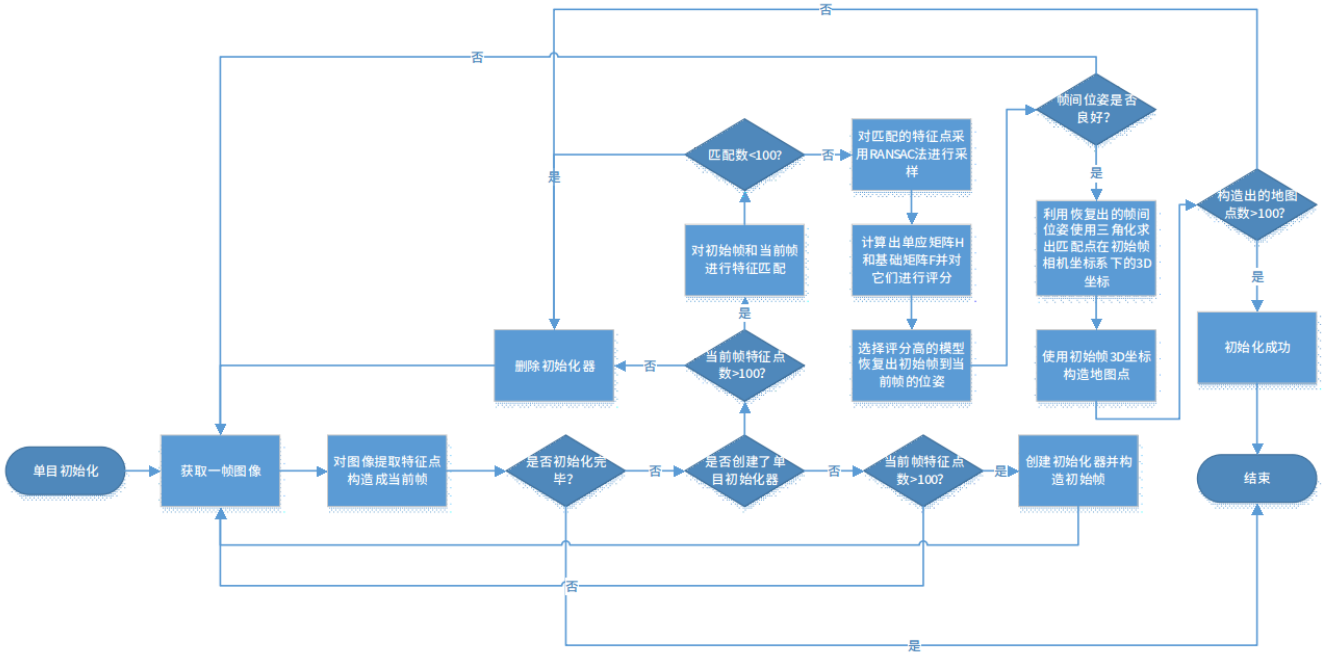
2.3 ORB-SLAM单目自动初始化流程分析

在扩展跟踪程序中，视觉SLAM所扮演的角色是能够只依赖相机读取一帧帧周围环境的图像通过特征提取与匹配等计算机视觉处理技术就能准确判断出当前相机的位姿，并将计算得到的位姿提交给ARToolkit引擎进行虚拟物体的渲染和叠加。

ORB-SLAM程序的设计来源于经典的视觉SLAM框架，如下图:



整个视觉SLAM流程包括以下步骤：
1. 传感器信息读取。在视觉SLAM种主要为相机图像信息的读取和预处理。如果是在机器人种，还可能有码盘、惯性传感器等信息的读取和同步。
2. 视觉里程计（Visual Odometry）。视觉里程计的任务是估算相邻图像间相机的运动，以及局部地图的样子。视觉里程计又称前端。
3. 后端优化。后端接收不同时刻视觉里程计测量的相机位姿，以及回环检测的信息，对它们进行优化，得到全局一致的轨迹和地图。由于接在视觉里程计之后，又称后端。
4. 回环检测（Loop Closing）。回环检测判断机器人是否到达过先前的位置。如果检测到回环，它会把信息提供给后端进行处理。
5. 建图（Mapping）。它根据估计的轨迹，建立与任务要求对应的地图。
从整个视觉SLAM流程中可以得知，视觉SLAM功能的核心是视觉里程计，
它可以计算得出相机当前的位姿信息；而其它步骤都是围绕着视觉里程计展开，如建图是为了更好地使视觉里程计求出结果；后端优化、回环检测是为了对视觉里程计求出的位姿结果进行优化，降低其噪音使其尽可能准确。
可见，视觉SLAM程序的流程设计具有标准性和一般性的特点；然而对于单目视觉SLAM而言，情况略有不同；由于单目摄像机在拍摄图像过程中省略了环境的深度信息，所以直接导致单目视觉SLAM的应用具有尺度不确定性；为了解决这个问题，一般的单目SLAM系统会进行自动初始化从而规定一个特定的尺度。这对于只使用单独的视觉SLAM系统完成特定任务的程序而言没有任何问题，然而对于需要利用单目SLAM系统与其它单目系统（如本项目中使用的ARToolKit引擎）配合工作的程序来说，为了使得程序能够正常运行并获得理想的效果，必须统一两者的尺度；一般的处理方式是使用其它单目视觉系统的尺度来规定SLAM系统的尺度，从而达到尺度统一的结果。
因此，在扩展跟踪程序中，理所当然的做法是使用ARToolkit系统采用的尺度来规定ORB-SLAM单目系统的尺度，而关闭其自动初始化确定尺度的过程，以达到两者在尺度上的统一。故在此之前，深入分析ORB-SLAM单目系统自动初始化工作流程变得十分关键；只有对ORB-SLAM单目初始化过程具有清晰透彻的认识，才能使得利用ARToolKit初始化ORB-SLAM流程顺利完成。
本人对ORB-SLAM单目自动初始化过程进行代码分析总结出其关键工作流程如下：
（本人分析的源代码位
于`相关参考资料/ORB-SLAM2原工程（github下载）/ORB_SLAM2/Examples/Monocular/mono_tum.cc`）



流程描述如下：

1. 首先，ORB-SLAM系统获取到一帧图像后，先将其转换为灰度图，然后对该帧图像提取ORB特征点并计算其描述子，随后将得到的信息连同图像本身一起用于构造当前帧对象mCurrentFrame，这表明ORB-SLAM系统已成功获取了当前图像。

2. 得到当前帧对象后，ORB-SLAM将判断系统当前的状态信息，如果当前系统已经初始化完成，则不进入初始化流程而进入追踪流程；如果系统还未进行初始化，则系统即将进行单目初始化处理。

3. 进入初始化流程后，首先判断是否已创建初始化器，这实际上是判断系统是否已经确定了用于初始化的初始帧；如果没有确定初始帧，则判断当前帧提取到的图像特征点数量是否大于100，如果满足，则将当前帧确定为初始帧；如果不满足，则放弃当前帧重新筛选符合条件的初始帧。

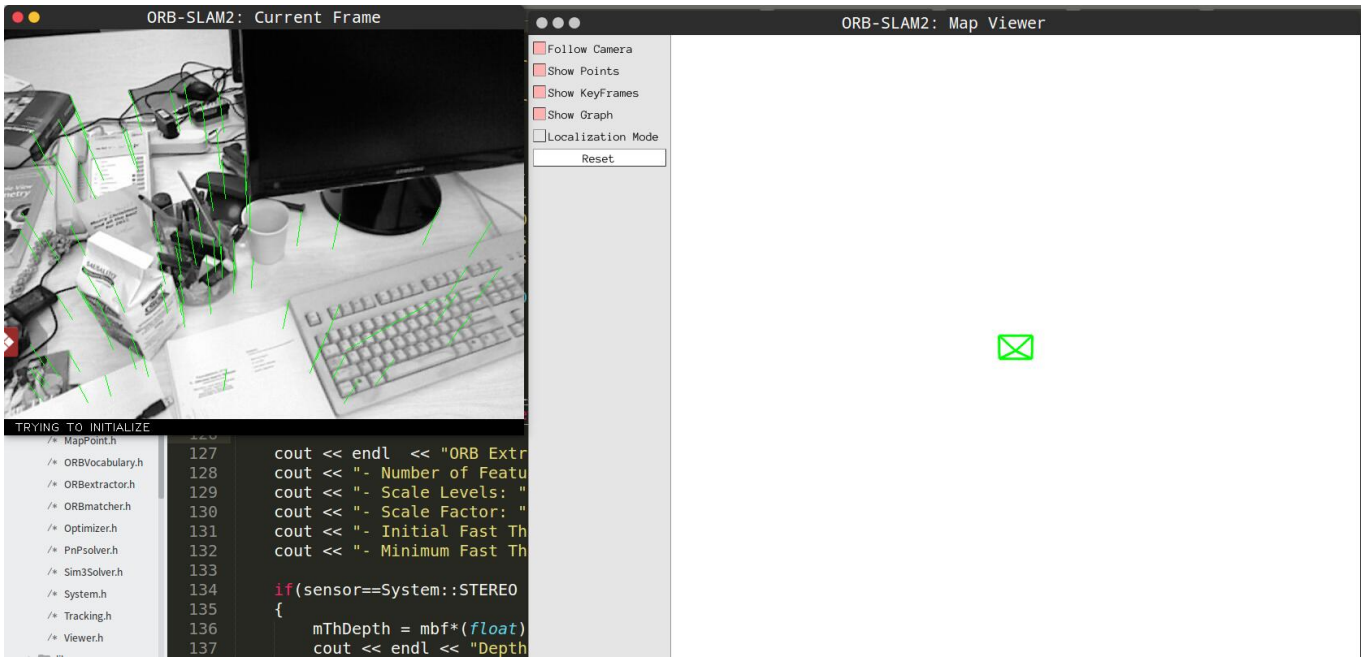
4. 初始帧确定完毕后，系统进行第二帧的选取工作；系统希望选取的第二帧特征点数量也大于100（连续两帧特征点数量充足），如果不满足，则删除初始化器以通知系统重新确定初始帧；这是为了确保初始帧和第二帧能有足够的特征点进入后续的特征匹配。

5. 选取到连续且特征点数量充足的初始帧和第二帧图像后，对它们进行特征匹配，以确定在两帧图像上同时出现的特征点，这样的点称为匹配点；如果得到的匹配点数量大于100，则说明匹配良好；否则，删除初始化器，重新确定初始帧和第二帧图像；系统之所以选择在匹配点数量不足的情况下重新选取初始帧，是为了防止系统继续做无意义的第二帧挑选工作；因为此时两帧之间相机的位移已足够大，以至于当前选取的第二帧与初始帧之间匹配到的特征点数量已无法满足要求；若继续下去，挑选到的第二帧继续与初始帧之间进行匹配获得的匹配点数量将越来越少。因此在这种情况下应重新确定初始帧来“重置”相机的位移。

6. 获取到足够数量的匹配点后，系统采用RANSAC（随机采样一致性）的方法来随机挑选出8对匹配的特征点，将其构成一组，共挑选出200组用于后续的基础矩阵F和单应矩阵H的求解（即八点法）。

7. 得到200组随机挑选的8对匹配点后，系统将开启两个线程分别计算初始帧到第二帧之间通过对极几何约束确定的基础矩阵F和单应矩阵H，并对它们的求解效果进行评分，目的是为了评估相机更符合哪种模型的运动，以供后续恢复出最为准确的帧间位姿。
8. 得到基础矩阵F和单应矩阵H后，系统通过评分模型（经验模型）来选取其中的一个矩阵，并选择从它恢复出初始帧到第二帧的位姿（t向量经过归一化，但此时并没有决定单目SLAM的尺度，真正决定尺度是在最后一步）。
9. 恢复出帧间位姿后，系统会检验恢复出的结果是否良好，如果良好，则利用该帧间位姿结合三角测量的知识恢复出匹配的特征点在采集初始帧时相机坐标系下的三维坐标；如果结果较差，则保留初始帧，重新筛选第二帧并重复以上步骤。
10. 得到初始帧三维坐标后，系统利用这些三维点来构造成Mappoint(地图点)以初始化地图，并**将初始帧相机坐标系设定为系统的世界坐标系。**
11. 初始化地图完成后，系统最后判断内部的地图点数量，若地图点数量小于100，则删除初始化器重新确定初始帧；若地图点数量充足，则系统的初始化成功，此时设置整个SLAM的尺度以及系统的相应状态，并退出初始化流程。
12. ORB-SLAM单目初始化完成后，系统随即进入追踪状态，以后每当向ORB-SLAM系统传递一个当前帧图像，就可以从ORB-SLAM系统返回当前相机的位姿`Tcw`（**世界坐标系到当前帧相机坐标系的变换矩阵**）

ORB-SLAM2系统单目初始化过程如下图所示：
1. ORB-SLAM2单目初始化进行中



2. ORB-SLAM2单目初始化完成

2.4 扩展跟踪程序详细设计

由基于自然特征跟踪注册的ARToolKit引擎示例流程分析可知，当ARToolKit检测到自然图像标记时，可以得到标记当前的位姿，需要注意的是，此处标记的位姿是指**标记(marker)三维坐标系到相机(camera)三维坐标系的变换矩阵**,即`Tcm`。

得到`Tcm`后，需要将其作为ORB-SLAM单目初始化的初始信息；然而在ORB-SLAM单目自动初始化流程分析中得知，ORB-SLAM系统的初始化需要初始帧到第二帧的帧间位姿`T21`，即**拍摄初始帧时相机三维坐标系到拍摄第二帧时相机三维坐标系的变换矩阵**；因此无法使用直接初始化ORB-SLAM系统；然而，若**假设自然图像标记在场景中不移动**的情况，可以利用自然图像标记作为中间物间接得到帧间位姿`T21`；具体操作为：假设在ORB-SLAM确定初始帧时，ARToolKit得到**自然图像标记三维坐标系到当前相机坐标系**的变换矩阵为`Tc1m`；同理，ORB-SLAM确定第二帧时，ARToolKit得到标记三维坐标系到当前相机坐标系的变换矩阵为`Tc2m`；为了得到初始帧到第二帧的帧间位姿`Tc2c1`(`T21`)，则有：

$$T_{21} = T_{c_2c_1} = T_{c_2m} * T_{mc_1} = T_{c_2m} * T_{c_1m}^{-1}$$

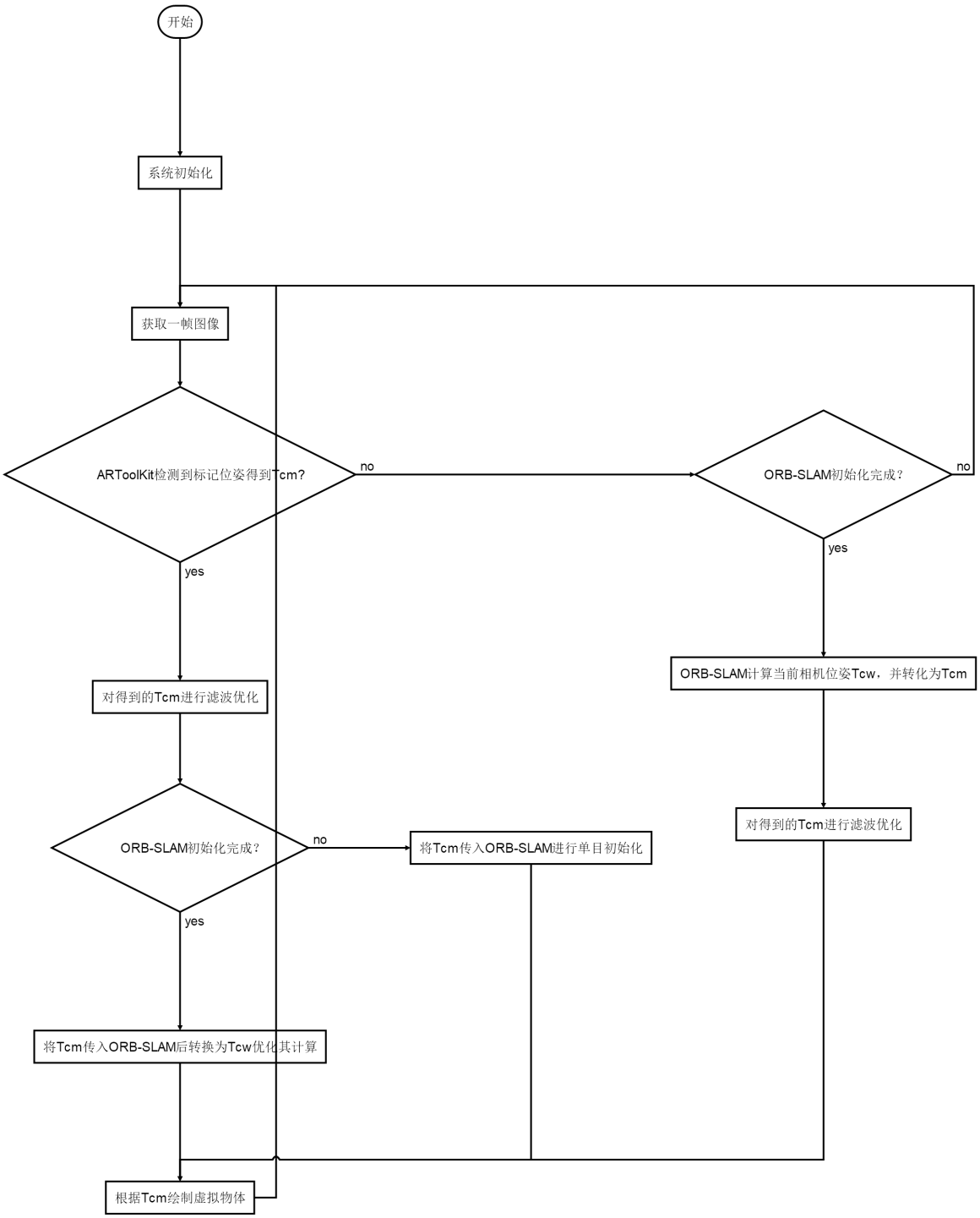
当ORB-SLAM初始化完成后跟踪状态时，每当ARToolKit向其传递一帧图片，需要实时向ARToolKit反馈估计的标记位姿`Tcm`;然而ORB-SLAM计算得到的结果是相机当前的位姿，即世界三维坐标系到当前相机三维坐标系的变换矩阵`Tcw`；为了得到`Tcm`，需要注意的是，之前提到ORB-SLAM系统将确定初始帧时相机的三维坐标系设置为世界坐标系，故`Tcw`实质是`Tcc1`,因此为了得到`Tcm`，有：

$$T_{cm} = T_{cc_1} * T_{c_1m} = T_{cw} * T_{c_1m}$$

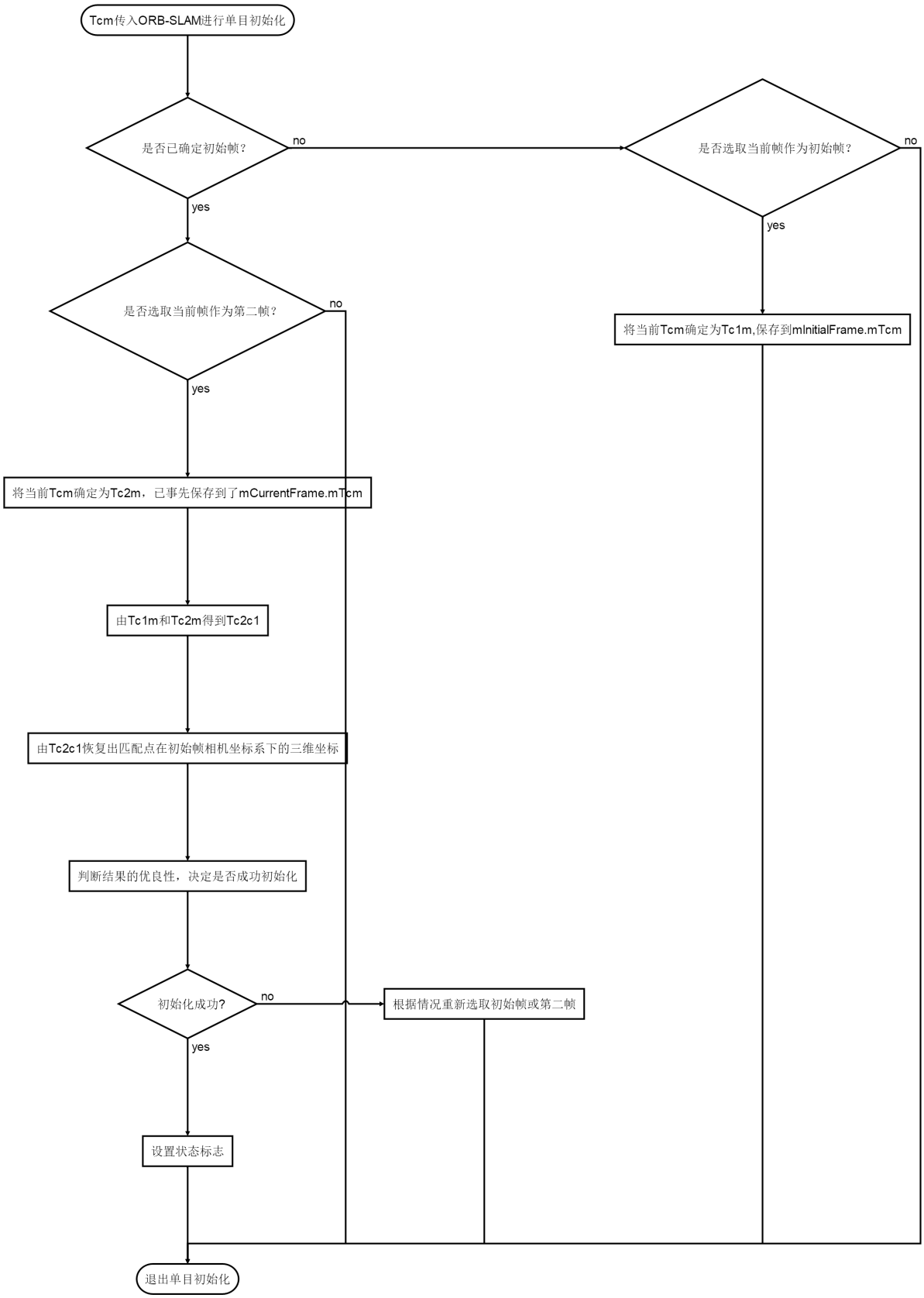
当ORB-SLAM初始化完成进入追踪状态后，ORB-SLAM会不断从ARToolKit处获取当前帧图像以便后台计算相机的位姿`Tcw`，当ARToolKit能检测通过检测到标记获得其位姿时`Tcm`时，由于该值是准确的，所以此时应该将其连同当前帧图像一起传入ORB-SLAM系统来优化ORB-SLAM计算的结果，即需要从`Tcm`转变到`Tcw`；针对该过程，有：

$$T_{cw} = T_{cc_1} = T_{cm} * T_{mc_1} = T_{cm} * T_{c_1m}^{-1}$$

综上，可以得到扩展跟踪流程的详细设计流程为：



其中，关于ORB-SLAM进行单目初始化的流程如下：


Tcm传入ORB-SLAM进行单目初始化
是否已确定初始帧？
no
yes
是否选取当前帧作为初始帧？
no
yes
将当前Tcm确定为Tc1m,保存到mInitialFrame.mTcm
是否选取当前帧作为第二帧？
no
yes
将当前Tcm确定为Tc2m，已事先保存到了mCurrentFrame.mTcm
由Tc1m和Tc2m得到Tc2c1
由Tc2c1恢复出匹配点在初始帧相机坐标系下的三维坐标
判断结果的优良性，决定是否成功初始化
初始化成功?
no
根据情况重新选取初始帧或第二帧
yes
设置状态标志
退出单目初始化

3. 扩展跟踪程序编码实现

3.1 ARToolKit与ORB-SLAM2编译整合

由于ARToolKit与ORB-SLAM2是两套不同的系统，即独立的工程。它们每个工程的内部都拥有自己的头文件、源文件、库文件以及用于指导编译工作的编译文件。为了将两个系统编译整合到一起，就需要对代码进行融合，以及创建新的编译文件来指导编译。在ARToolKit中，编译文件是通过`Configure`文件生成的`makefile`文件；而在ORB-SLAM2中，编译文件是位于根目录下的`CmakeLists.txt`文件；为了将两者结合起来，本人以ARToolKit工程为基础，在其上添加ORB-SLAM2工程的相关文件：
1. 将ORB-SLAM2工程的相关头文件添加到ARToolKit工程中的`include`目录下
2. 将ORB-SLAM2工程的相关源文件添加到ARToolKit工程中的`src`目录下
3. 将ORB-SLAM2工程编译`libORB_SLAM2.so`库所需的第三方依赖库移至ARToolKit工程中的`Thirdparty`目录下
4. 以ARToolKit工程的`nftsimple`示例作为扩展跟踪示例的源文件，在其上添加ORB-SLAM2工程中`mono_tum`示例的源代码

在ARToolKit中，`nftsimple`示例的编译工作大致可分为两部分：
1. 将`lib/SRC`下的库源文件编译成ARToolKit的各种基础库以供示例程序调用
2. 编译`nftsimple`示例

在ORB-SLAM2中，`mono_tum`示例的编译工作也大致相同：
1. 将`src`目录下的源文件编译成库`libORB_SLAM2.so`以供示例程序调用
2. 编译`mono_tum`示例

本人创建了新的编译文件（位于扩展跟踪根目录下的`CmakeLists.txt`），采用Cmake工具对扩展跟踪示例程序进行编译，在该文件内，指定的编译工作为：
1. 统一ARToolKit与ORB-SLAM2的编译参数（两者部分参数有冲突）
2. 编译ORB-SLAM2系统所需的`libORB_SLAM2.so`
3. 编译扩展跟踪示例`extended-tracking.cc`
需要注意的是，由于没有修改ARToolKit的库文件，所以无需对其进行编译重构；因此在编译扩展跟踪示例前，需要单独编译ARToolKit部分所需的库文件，具体操作为在`lib/SRC`下输入：

```
1.  make -j4
```

需要清理ARToolKit库文件时，同样在该目录下输入：

```
make clean
```

即可。

3.2 扩展跟踪示例代码编写

1. 将ARToolKit工程中的`nftsimple.c`更名为`extended-tracking.cc`
2. 在`extended-tracking.cc`中添加ORB-SLAM2部分所需头文件：

```

#include <iostream>


#include <algorithm>


#include <fstream>


#include <chrono>


#include <opencv2/core/core.hpp>


#include <System.h>


using namespace std;
```

3. 定义与ORB-SLAM2相关的全局变量：

```
ORB_SLAM2::System *ptr_SLAM;
ORB_SLAM2::System::my_eTrackingState *ptr_State;     //用于标记SLAM系统的状态
```

4. 在主函数(`main()`)中添加初始化ORB-SLAM2系统的代码：

```
if(argc != 3)
{
  cerr << endl << "Usage: ./mono_tum path_to_vocabulary
```

```
path_to_settings" << endl;
  return 1;
}

//初始化所有线程
//参数：字典文件、内参文件（SLAM是个system变量）
ORB_SLAM2::System
SLAM(argv[1],argv[2],ORB_SLAM2::System::MONOCULAR,&ptr_State,true);
ptr_SLAM = &SLAM;
```

5. 在ARToolKit回调函数`mainloop()`中，当ARToolKit通过ar2tracking线程成功得到标记位姿并经过滤波优化后，添加如下代码：

```
ARfloat trans_temp[4][4];
for(int m=0;m<3;m++)
  for(int n=0;n<4;n++)
    trans_temp[m][n]=markersNFT[i].trans[m][n];

trans_temp[3][0]=trans_temp[3][1]=trans_temp[3][2]=0;
trans_temp[3][3]=1;
cv::Mat Tcm(4,4,CV_32F,(ARfloat *)trans_temp);  //得到模板坐标系到相机坐标系的变换矩阵

cv::Mat im(480,640,CV_8UC3,gARTImage);  //构造mat类型图像

if(im.empty())
{
  cerr << endl << "Failed to load image from ARToolkit."<<endl;
  return;
}

//获取当前的时间戳
std::chrono::system_clock::time_point t_epoch;
std::chrono::duration<double,std::ratio<1,1000000>> duration_micr_sec =
std::chrono::system_clock::now() - t_epoch;
double tframe = duration_micr_sec.count() / 1000000.0;

(*ptr_SLAM).TrackMonocular(im,tframe,Tcm,1); //将图片和时间戳以及位姿传入SLAM系统
```

6. 在ARToolKit跟踪标记的位姿失败时，添加如下代码：

```
//若ORB-SLAM初始化成功
if((*ptr_State)==ORB_SLAM2::System::OK ||
(*ptr_State)==ORB_SLAM2::System::LOST)
{
```

```
    cv::Mat im(480,640,CV_8UC3,gARTImage);     //构造mat类型图像
    if(im.empty())
    {
      cerr << endl << "Failed to load image from ARToolkit."<<endl;
      return;
    }

    //获取当前的时间戳
    std::chrono::system_clock::time_point t_epoch;
    std::chrono::duration<double,std::ratio<1,1000000>> duration_micr_sec
= std::chrono::system_clock::now() - t_epoch;
    double tframe = duration_micr_sec.count() / 1000000.0;

    //将图片和时间戳传入SLAM系统,得到当前模板坐标系到相机坐标系的变换矩阵
    cv::Mat Tcm((*ptr_SLAM).TrackMonocular(im,tframe,0));

    if((*ptr_State)==ORB_SLAM2::System::OK)    //位姿有效(SLAM跟踪未丢失)
    {
      cout<<"跟踪未丢失Tcm:"<<endl<<Tcm<<endl;
      for(int m=0;m<3;m++)
        for(int n=0;n<4;n++)
          markersNFT[i].trans[m][n]=(ARdouble)Tcm.at<float>(m,n);

      markersNFT[i].valid = TRUE; //更改本次的跟踪状态为成功

      //对位姿结果进行滤波优化
      if (markersNFT[i].ftmi) {
          if (arFilterTransMat(markersNFT[i].ftmi, markersNFT[i].trans, 0)
< 0) {
            ARLOGe("arFilterTransMat error with marker %d.\n", i);
          }
      }
}
```

7. 在ORB-SLAM2端，当ARToolKit检测到标记传入Tcm时，调用的函数其原型为：

```
cv::Mat TrackMonocular(const cv::Mat &im, const double &timestamp, const
cv::Mat &Tcm, bool tag);
```

以及

```
cv::Mat GrabImageMonocular(const cv::Mat &im, const double &timestamp, c
onst cv::Mat &Tcm, bool tag);
```

其中的`tag`用于向ORB-SLAM2系统指示当前ARToolKit系统是否识别到标记，0为未识别，1为

识别；在`GrabImageMonocular()`中，将`tag`和`Tcm`保存到当前帧对象中：

```
mCurrentFrame.mTcm = Tcm.clone();        //传入当前帧结构体
mCurrentFrame.mbtag = tag;
```

此处传入的`Tcm`有两个作用：

1. 若ORB-SLAM系统还未进行初始化，则使用其进行初始化
2. 若ORB-SLAM系统已完成初始化，则传入该值对ORB-SLAM当前相机的位姿进行设置以“纠正”其计算结果

8. 当ARToolKit无法检测到标记时，调用的函数其原型为：

```
cv::Mat TrackMonocular(const cv::Mat &im, const double &timestamp, bool tag);
```

以及

```
cv::Mat Tracking::GrabImageMonocular(const cv::Mat &im, const double &timestamp,bool tag);
```

在`GrabImageMonocular()`中，只设置了`tag`：

```
mCurrentFrame.mbtag=tag;
```

但此时返回了当前模板坐标系到相机坐标系的变换矩阵`Tcm`：

```
return mCurrentFrame.mTcw*mInitialFrame.mTcm;
```

9. 在初始化函数`MonocularInitialization()`中，当确定ORB-SLAM2所需的初始帧时，有：

```
//步骤1：得到用于初始化的第一帧，初始化需要两帧
mInitialFrame = Frame(mCurrentFrame);

//初始帧确定时，设置当前状态下的Tmc到初始帧
mInitialFrame.mTcm = mCurrentFrame.mTcm.clone();
```

10. 第二帧得到确定后，为了计算出初始帧到第二帧的帧间位姿，有：

```
//求解T21——即第一帧相机坐标系到第二帧相机坐标系的变换矩阵
cv::Mat T21 = mCurrentFrame.mTcm * mInitialFrame.mTcm.inv();
```

11. 计算出帧间位姿`T21`后，将`T21`分解为`Rcw`和`tcw`两部分，随后传入`Initialize()`中恢复出匹配点在初始帧相机坐标系下的三维坐标：

```
cv::Mat Rcw(T21.rowRange(0,3).colRange(0,3));    //第一帧到第二帧的R矩阵
cv::Mat tcw(T21.rowRange(0,3).col(3));       //第一帧到第二帧的T矩阵

mpInitializer->Initialize(mCurrentFrame, mvIniMatches, Rcw, tcw, mvIniP3D, vbTriangulated);
```

12. ORB-SLAM2完成初始化进入追踪状态后，为了由`Tcm`计算出`Tcw`，并以此来设置ORB-SLAM2的当前相机位姿，有：

```
//ARToolKit追踪成功，利用ARToolKit得到的位姿修正ORB-SLAM
if(mCurrentFrame.mbtag)
{
  mCurrentFrame.mTcw= mCurrentFrame.mTcm * mInitialFrame.mTcm.inv();
}
else
{
  mCurrentFrame.SetPose(mVelocity*mLastFrame.mTcw);
}
```

与之相关的函数为：`TrackWithMotionModel()`以及`TrackReferenceKeyFrame()`.