Universidad Miguel Hernández de Elche

MÁSTER UNIVERSITARIO EN

ROBÓTICA

"ORB-SLAM2 vs ARCore, Comparativa y extensión"

Trabajo de Fin de Máster

2017-2018

Autor: Javier Morcillo Marín

Tutor: Óscar Reinoso García

Índice

1. Agradecimientos

Por descontado, los primeros que han de aparecer en esta sección ya que han hecho posible que yo llegue a este punto, es mi familia, que me ha apoyado en todo momento para este empeño. En concreto, gracias a mi mujer y su esfuerzo atendiendo las tareas que tocaba mientras yo dedicaba tiempo al Máster. Gracias a mi suegra, por cuidar a mi hija todas las tardes que yo no podía estar con ella debido a las clases, y gracias a mi hija por devolverme la ilusión en épocas no demasiado buenas.

Una vez dicho esto, he de agradecer a todos los profesores que han impartido el máster, su paciencia por la gran cantidad de preguntas que debieron responder por mi parte, en especial a mi tutor de TFM, por haber sabido guiarme y redirigirme en la adversidad de mis descubrimientos.

Por último, he de agradecer al equipo de ORB-SLAM2 el haber publicado su trabajo, ya que de esa manera me han dado a conocer un poco más los entresijos necesarios para llevar a cabo un SLAM visual.

2. Introducción/Objetivo

En el ámbito de la robótica existen varios hitos en los cuales hay líneas de desarrollo. Uno predilecto de la robótica es el SLAM.

El SLAM (Simultaneous Localization And Map), es la capacidad que tiene un sistema de localizarse geométricamente dentro de un entorno mientras que al mismo tiempo genera un mapa de dicho entorno, también geométrico.

Existen numerosos tipos de sensores con los cuales, un sistema al que podemos llamar robot, puede extraer información del entorno que le rodea.

Este trabajo se centrará en SLAM visual en tiempo real, es decir, en conseguir localizar un dispositivo dentro de un mapa 3D (y a su vez generar dicho mapa), el cual se genera en tiempo real utilizando únicamente (o casi únicamente) información visual.

Como se verá en la sección de estado del arte, existen numerosos trabajos acerca de este campo, pero en este documento se abordará el estudio únicamente de 2 librerías multiplataforma para hacer SLAM, sus nombres son ARCore y ORB-SLAM2.

Además, este trabajo intentará trascender las librerías estudiadas intentando idear o incorporar características nuevas que las pudiesen hacer más atractivas/potentes.

Parte de este trabajo, se redactará a modo de cuaderno de bitácora, pues es una de la intenciones el otorgar al lector la travesía de descubrimientos obtenidos.

Antes de comentar el estudio, con respecto al código del software que se usará en cada caso, se ha de comentar algún aspecto de las licencias más usadas:

- Public domain: En este tipo de licencias no hay ninguna restricción. Los desarrolladores pueden usar este código que es libre y abierto (se tiene acceso al código fuente) a su antojo, incluso pueden modificarlos.
- GPLv3: En esta licencia se ofrece el código. La versión 3 relaja un poco las características del software bajo el que se firman. Eres libre de usar y modificar todo el software de manera privada, pero si publicas algo dependiente de GPLv3 te sometes a sus restricciones.
- MIT: Esta licencia tiene muy pocas restricciones, permite usar el código fuente en software propietario.

2.1. Justificación

Este trabajo, como ya se ha comentado, está basado en el estudio de dos librerías concretas, ORB-SLAM2 y ARCore.

El motivo por el cual se han elegido dichas librerías es que ambas son bastantes representativas en su dominio de aplicación.

ARCore de Google es una librería moderna que se usa sólamente en dispositivos móviles y es multiplataforma. Es gratuita por lo que cualquier desarrollador la puede usar. Al ser la programación de dispositivos móviles potentes bastante reciente, no hay demasiados referentes en los que elegir, pero si van apareciendo, seguramente que dichas nuevas publicaciones se compararán con ARCore en sus benchmarks.

Con respecto a ORB-SLAM2, la justificación es aún mayor pues es un referente en los sistemas de SLAM totalmente aceptado por toda la comunidad científica. Para ver hasta qué punto es ORB-SLAM2 un referente, en los sistemas punteros, siempre se publican videos comparando sus resultados con ORB-SLAM2. Esto mismo ocurre con muchos sistemas que se expondrán en el Estado del Arte. En https://openslam-org.github.io/ aparece ORB-SLAM como una de las librerías de referencia, por lo que queda patente su importancia.

2.2. Consideraciones previas

En algunas secciones, a modo de ejemplo, aparece código fuente propio y de las librerías que se usan. En la mayoría de ocasiones este código fuente se ha resumido un poco para reflejar de la manera más concreta posible el concepto en cuestión que se quiere dar a conocer.

En otros casos, hay que tener en cuenta que no se llega a explicar con profundidad algún hito, esto se debe a que hacerlo sería demasiado extenso y se escaparía un poco del objetivo de este trabajo.

3. Estado del arte

En el momento de redactar este trabajo, existen numerosas publicaciones acerca de librerías, algoritmos y sistemas que hacen posible el realizar SLAM monocular.

Existen dos grandes grupos de sistemas para hacer SLAM (para un mapa geométrico): los que usan puntos característicos y los que no. Los que no usan puntos característicos, en ocasiones, sí que usan algún modo de detección interna de píxeles (algún detector o algún descriptor), pero no usan directamente los puntos característicos tal y como se conocen, al menos de manera explícita.

En esta sección se expondrán los sistemas y librerías que se suelen usar para SLAM.

3.1. Técnicas

3.1.1. Puntos característicos

El uso de puntos característicos es una idea relativamente reciente en sistemas de conocimiento basados en imágenes.

La idea general es recorrer una imagen y buscar zonas dentro de la ella que tengan algo muy especial de manera local. Por ejemplo, encontrar píxeles que den fuertemente la sensación de ser esquinas, o aristas, … Cada una de esas zonas se denominan puntos característicos de la imagen.

Para no entrar en demasiados detalles acerca de ellos, comentar que existen muchas publicaciones de técnicas al respecto: ORB, SIFT, SURF, FAST, … Cada uno de ellos ofrece unas ventajas y unos inconvenientes.

Lo que sí que tienen en común dichas técnicas es que cada una de ellas ofrece un **detector** de puntos (técnica que reconoce que un pixel es un punto característico), un **descriptor** (técnica para poder describir el punto detectado para buscarlo posteriormente) o **ambas**.

3.1.2. Lucas Kanade

Lucas Kanade no es un método de SLAM en sí, pero dentro de los sistemas de SLAM es muy conocido y hay que decir, que la sencillez que ofrece para poder hacer SLAM local, lo hace idónea para aparecer como una técnica dentro del estado del arte. Al no ser una librería de SLAM, no posee conceptos propios del SLAM como por ejemplo “loop closing” ni cosas parecidas a esta.

Se basa en reconocer cierto entorno de vecindad de un píxel para descubrir si dicho píxel se ha movido o no.

Una de las modalidades predilectas de esta técnica, aunque no la única, es que dada una nube de puntos 2D y otra nube tomada en el frame siguiente, es capaz de devolver una transformación que han sufrido los puntos para pasar de un frame a otro. Por lo tanto, de manera explícita no se le tienen que pasar descriptores de puntos 2D.

En realidad la técnica que se usa en sistemas que hacen SLAM se denomina Kanade Lucas Tomasi (KLT). En el siguiente video se aprecia que con OpenCV, de manera muy sencilla se puede hacer un tracking de una cámara en un entorno:

https://www.youtube.com/watch?v=dMo3LrFdMKI

Se puede ver una implementación en la siguiente URL:

https://github.com/avisingh599/mono-vo

3.1.3. SVD

SVD es una técnica de descomposición de matrices en otras que tienen ciertas características interesantes. Concretamente la técnica se basa en la descomposición en valores singulares y entre otras posibles propiedades, las matrices resultantes configuran sistemas ortonormales y matrices en las que aparece el error de resolución.

Se usa para el cálculo de una posible matriz de transformación que ha sufrido una cámara para pasar de un frame a otro.

3.1.4. RANSAC

RANSAC (RANdom SAmple Consensus) es un algoritmo iterativo para calcular los parámetros de un modelo matemático de un conjunto de datos observados. Estos datos pueden contener cierto nivel de ruido y puede contener también valores que no representan el modelo original que se quiere descubrir.

Es un algoritmo no determinista, y está basado en buscar de manera aleatoria un patrón que minimice el error partiendo del modelo matemático que se marca en cada iteración. Por lo tanto, en principio dos ejecuciones consecutivas de RANSAC sobre los mismos datos de entrada, no tienen porqué derivar en el mismo resultado.

3.2. Librerías

Existen algunas librerías que se usan no sólo en sistemas de SLAM, sino en temas de ingeniería en general. Sobre todo abundan librerías de tratamiento de matrices. Algunas librerías reseñables son las que a continuación se citan.

3.2.1. g2o.

g2o es una librería usada para optimizar grafos, de hecho, sus autores escriben como subtítulo lo siguiente: A General Framework for Graph Optimization. Está bajo la licencia LGPLv3.

Su corazón consiste en optimizar las funciones de error no lineales basadas en gráficos. En principio su diseño se pensó para resolver problemas genéricos con pocas líneas de código.

Sus autores son: Rainer Kuemmerle, Giorgio Grisetti, Hauke Strasdat, Kurt Konolige y Wolfram Burgard y el repositorio con el código fuente se puede descargar en la siguiente URL de github: https://github.com/OpenSLAM-org/openslam_g2o

3.2.2. OpenCV

OpenCV es la librería de visión por excelencia. Tiene soportadas un gran número de funciones y algoritmos usados no solo en visión, sino también en temas de ingeniería genérica (resolución de ecuaciones, métodos de minimización, ...)

Sus autores son muy numerosos pues se basa en la idea de que si alguien de cierta importancia pública algo que podría añadirse a OpenCV y es aceptado por su comunidad, se publica.

La licencia se puede leer en su página web: https://opencv.org/ dentro de la cual hay accesos a su repositorio que es https://github.com/opencv/opencv.

3.2.3. Eigen

Ésta es una librería para el cálculo de matrices. Es bastante famosa y se usa en muchas aplicaciones de cálculo matemático que tengan que ver con matrices. Cuando se habla de matrices, también nos referimos a vectores.

Está bajo la licencia MPL2 y su página principal es la siguiente: http://eigen.tuxfamily.org

3.3. Sistemas que proporcionan SLAM

Algunos de los sistemas que proporcionan SLAM que se pueden destacar en la actualidad son los siguientes.

3.3.1. ARKit

En la introducción se ha expuesto que se va a analizar dos librerías de SLAM, una de ellas es ARCore. ARKit de Apple es quizás el competidor más directo de dicha librería, quizás por aquello de lo que publica Google lo lanza Apple para competirle y viceversa.

ARKit es un sistema que, dado un entorno captado con la cámara y una odometría interna del móvil, intenta localizar el teléfono dentro de dicho entorno a la par que intenta calcular información 3D del mismo entorno. Existen algunos videos en youtube, como por ejemplo este: https://www.youtube.com/watch?v=a4YYf87UjAc donde se puede apreciar que van muy a la par. Existen otros tantos videos en donde parece ser que ARCore funciona algo mejor que ARKit. Aún así, las comparativas no me parecen acertadas, puesto que son hardware diferentes con sistemas operativo diferentes, y que por lo tanto, no se puede hacer una comparativa justa.

Sea como fuere, ARCore tiene una ventaja clara sobre ARKit, y es que, aunque es de código privativo (ambas lo son), soporta más plataformas que ARKit, la cual sólo funciona en iOS.

3.3.2. PTAM

Según sus autores, PTAM es una librería con licencia GPLv3 (por lo que podemos ver el código fuente) capaz de hacer SLAM monocular. Comentan los mismos autores que funciona en entornos pequeños y que se basa fundamentalmente en dos trabajos:

- Georg Klein and David Murray, "Parallel Tracking and Mapping for Small AR Workspaces", Proc. ISMAR 2007
- Georg Klein and David Murray, "Improving the Agility of KeyFrame-based SLAM", Proc. ECCV 2008

El código fuente de la librería se encuentra en la siguiente URL:

https://github.com/Oxford-PTAM/PTAM-GPL

3.3.3. DSO: Direct Sparse Odometry

Otro trabajo que desde luego tiene que tenerse en cuenta es DSO. Es un sistema de odometría visual que desde luego parece tener muy buenos resultados. Quizás tampoco se le pueda catalogar de SLAM, ya que en realidad, nunca cierra bucles, por lo que va acumulando errores (aunque pequeños), y eso hace que el mapa sea impreciso a la larga.

Una cosa curiosa es que no se basa en puntos característicos (y al menos de manera explícita parece que no los usa), sino que implementa un sistema basado en modelos probabilísticos y modelos de optimización.

Para ver el video que tienen como presentación, podemos ir a la siguiente URL:

https://vision.in.tum.de/research/vslam/dso?redirect=1

3.3.4. LSD-SLAM

Al igual que DSO, este sistema de SLAM no está basado en puntos característicos. Según su documentación crea mapas de grandes dimensiones y lo que sus autores catalogan de semi-densos. Está bajo licencia GPLv3.

Una de las características que tiene, es que los desarrolladores muestran pruebas realizadas con móviles, las cuales funcionan de manera aceptable. Las plataformas móviles se soportan de manera nativa.

Está basado en los siguientes papers:

- LSD-SLAM: Large-Scale Direct Monocular SLAM, J. Engel, T. Schöps, D. Cremers, ECCV '14
- Semi-Dense Visual Odometry for a Monocular Camera, J. Engel, J. Sturm, D. Cremers, ICCV '13

Se puede descargar desde la siguiente URL: https://github.com/tum-vision/lsd_slam

3.3.5. SVO

Este sistema sí que trabaja con puntos característicos. Al parecer funciona bastante bien, pero leyendo el paper del autor (SVO: Fast Semi-Direct Monocular Visual Odometry Christian Forster, Matia Pizzoli, Davide Scaramuzza), tampoco parece que tenga “loop closing”, aunque se ha de decir, que parece que funciona bastante bien (¿será por la calidad de la cámara?). Está bajo licencia GPLv3.

Su implementación se puede encontrar en la siguiente URL:

https://github.com/uzh-rpg/rpg_svo

4. ARCore

ARCore es una librería gratuita de realidad virtual propietaria de Google pero que es soportada por varias plataformas. Las plataformas soportadas según la documentación son: Android, Android NDK, Unity (Android e iOS), Unreal y iOS.

En las secciones posteriores pondremos de manifiesto sus características y sus limitaciones. Para ver todas las características de la librería se puede visitar la siguiente URL:

https://developers.google.com/ar/

4.1. Características

Al decir que es una plataforma de realidad virtual, lo que hace concretamente es poder localizar el dispositivo que se esté moviendo en un entorno 3D usando dos tipos de información: Información visual proporcionada por la cámara a modo de imágenes e información extraída de los sensores de movimiento, lo que podría ser SLAM.

ARCore proporciona puntos 3D, planos 3D en forma de polígonos y la capacidad de hacer 2 cosas: intersecciones con los objetos que tiene almacenado y poner puntos de anclaje para usarlos de referencia a la hora de poner objetos en un mundo 3D.

Otra característica que ofrece ARCore es que te proporciona una idea de la iluminación global que hay en una escena.

Ofrece varios front-end según el sistema que se utilice, por ejemplo tenemos front-ends en Java, C, Objective-C y Unity. De esta manera les es más sencillo a los desarrolladores adaptar esta librería a sus intereses porque no te obliga a trabajar en C/C++.

En el siguiente video se pone de manifiesto algunas de sus funcionalidades:

https://www.youtube.com/watch?v=ttdPqly4OF8

4.2. Requisitos

ARCore no funciona en todos los móviles. Para ver aquellos dispositivos soportados hay que ir a la siguiente página:

https://developers.google.com/ar/discover/supported-devices

Mirando esa página hay un dato que resulta algo curioso. Si recordamos en la sección anterior, expuse que necesitaba sensores de movimiento. En realidad este segundo requisito

expuesto en dicha sección, el de odometría de movimiento, no forma parte de la publicidad de ARCore, pero claro, si miramos la página citada anteriormente, podremos ver que por alguna razón, no soporta todos los dispositivos Android, y que casualmente, los dispositivos soportados tienen una IMU interna. Por lo tanto, por mi parte infiero a que, una vez visto cómo funciona, la IMU es un requisito indispensable.

4.3. Instalación de ARCore en Android

Para poder utilizar la librería ARCore en android se necesitan hacer algunas cosas en los proyectos que no son muy estándar. En esta sección se describirán todos los pasos para poder configurar un proyecto de Android Studio para que pueda utilizar ARCore a través del NDK.

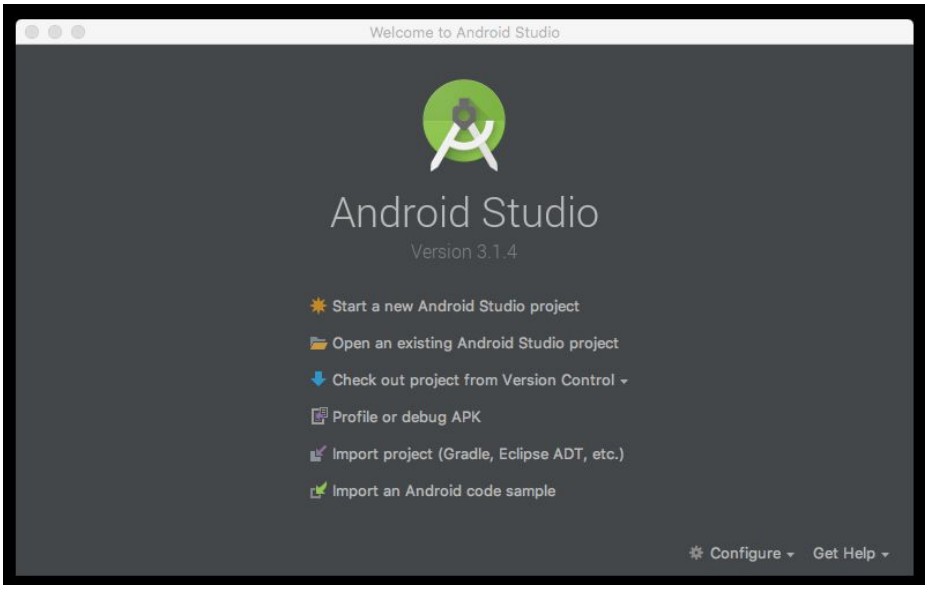
Se va a exponer el proceso completo para que cualquiera pueda ponerse manos a la obra con ARCore, pero, lo que no se va a poner es todo el pipeline de renderizado ni nada extra que no sea la configuración del proyecto, pues esto recae en la parte que el lector podría descubrir por sí solo.

4.3.1. Android Studio

Lo primero que necesitamos es instalar el Android Studio. Este IDE lo podemos descargar a través de la página oficial:

https://developer.android.com/studio/

Depende de la plataforma que tengamos (Windows, MAC o Linux), se deberá seleccionar una manera de descargarlo e instalarlo. Nada más descargarlo al abrirlo deberá aparecer algo parecido a la siguiente imagen:



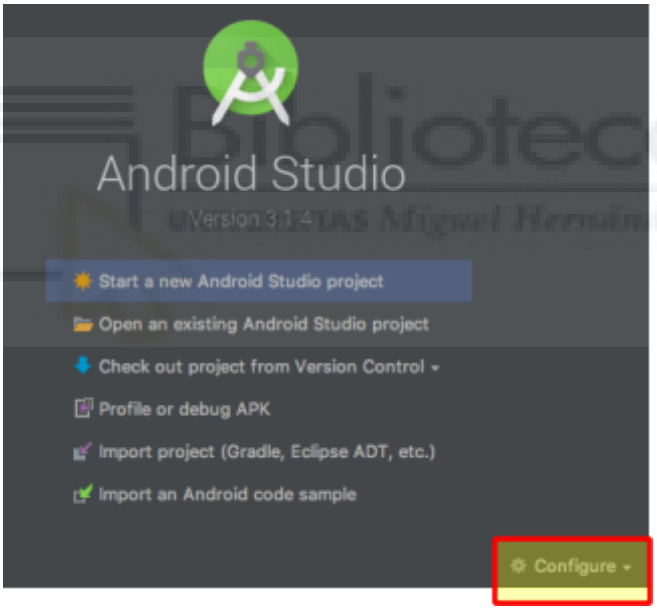
Una vez descargado e instalado, no hemos acabado todavía, porque si bien es cierto que tenemos ya instalado Android Studio, también lo es el hecho de que no podremos debuguear nuestra aplicación llegado el momento. Por lo tanto, tenemos que habilitar en el teléfono que vayamos a usar la opción de modo desarrollador. Para hacer esto hay que seguir los pasos que se indican en la siguiente URL:

https://www.xatakandroid.com/tutoriales/como-activar-las-opciones-de-desarrollo-en-android-4-2-jelly-bean

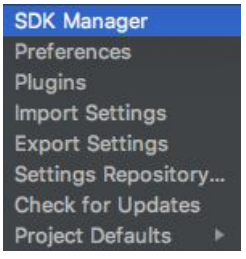
4.3.2. Descargar el NDK

El NDK (Native Development Kit) el el conjunto de librerías y herramientas que hacen posible que el Android Studio pueda usar C++ como lenguaje de desarrollo.

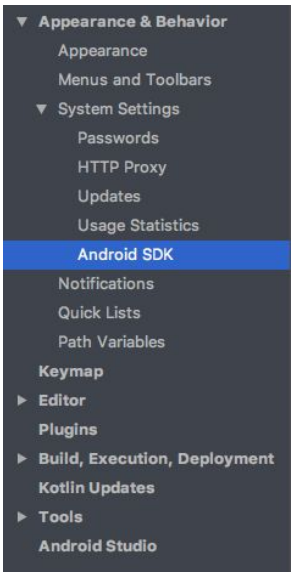
Para poder activarlo hay que desplegar el botón de configure que se muestra en la siguiente imagen:



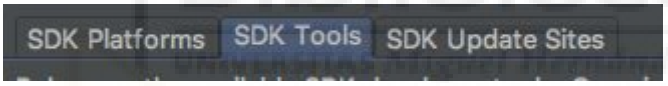
Una vez hecho esto aparece un menú con las siguientes opciones, entre las cuales se encuentra la opción a seleccionar que es SDK Manager:



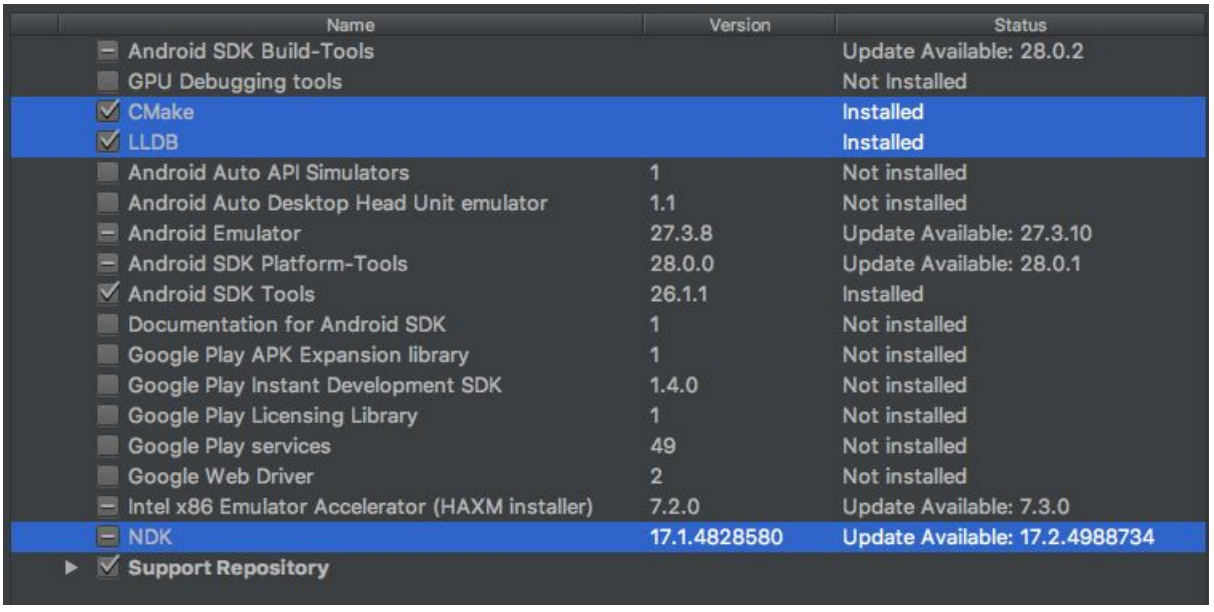
En el menú de la izquierda hay que seleccionar Android SDK



En la parte de la derecha hay que seleccionar SDK Tools como aparece en la siguiente imagen:



Cuando se hace esto aparece un menú desplegable con los muchos plugins que podemos instalar en nuestro Android Studio. Concretamente necesitamos los que a continuación se señalan:

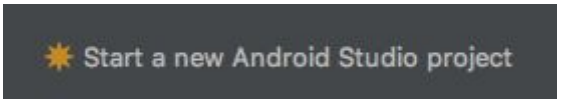


4.3.3. Crear un proyecto nuevo

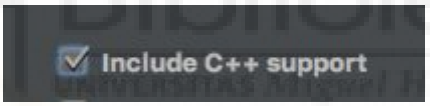
Lo primero que podemos hacer para usar la librería ARCore es crear un proyecto en el cual la vayamos a usar. En este paso, también se seleccionará la opción de que Android Studio utilice C++ en nuestro proyecto. En realidad esta posibilidad se puede hacer a posteriori, es decir, se podría crear un proyecto sin usar el NDK, y activarlo posteriormente, pero por sencillez, se lo vamos a indicar desde el principio.

4.3.3.1. Crear el proyecto

Presionamos el botón de crear un nuevo proyecto de Android Studio:



Una vez hecho esto se nos preguntará cosas acerca de la configuración el proyecto. Como no es el objetivo de este trabajo, el explicar estos pasos a fondo, vamos a poner lo básico: nombre de proyecto y path de instalación del proyecto, en mi caso utilizaré como nombre del proyecto ARTest. Lo más importante es marcar la casilla de que queremos usar C++:

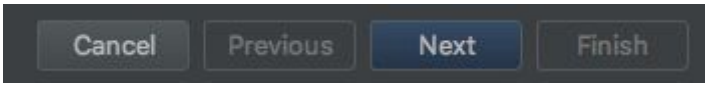


En mi caso, por cómo tengo configurado Android Studio, por defecto me pone el siguiente package: `com.javiermorcillomarin.artest`. El nombre del package es muy importante aquí puesto que para poder hacer llamadas a C++ se usa ese nombre. Cuando se termine de realizar el proyecto, echando un vistazo a las llamadas de C++ desde Java se puede ver el patrón que han de seguir dichas llamadas para poder funcionar

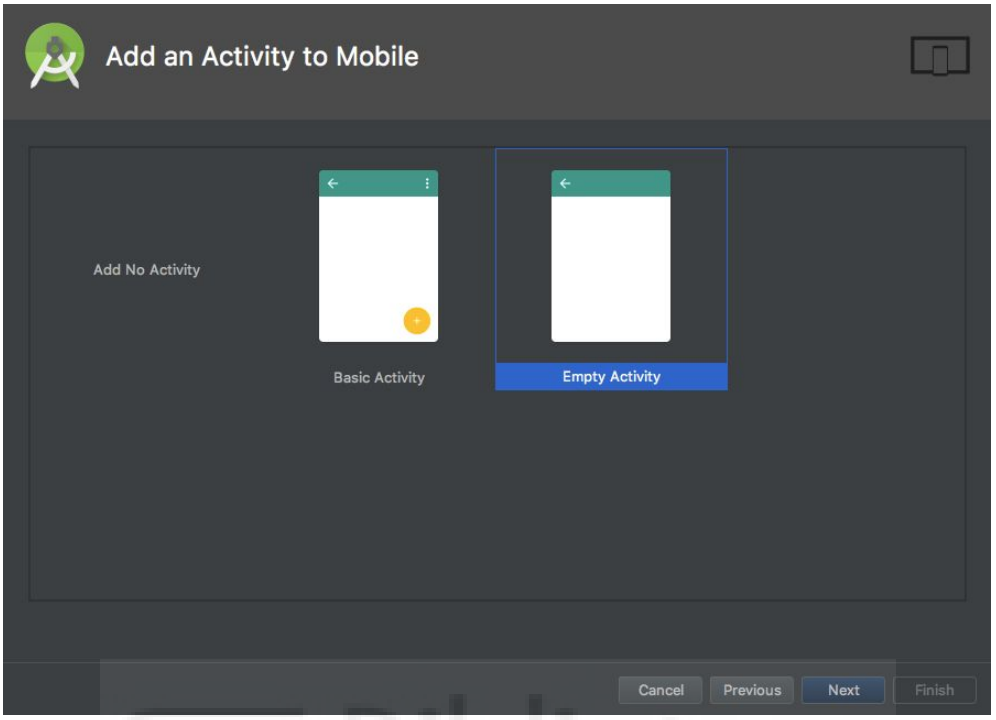
Para más información acerca de este aspecto, se puede consultar el siguiente tutorial que considero que es bastante simple:

https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html

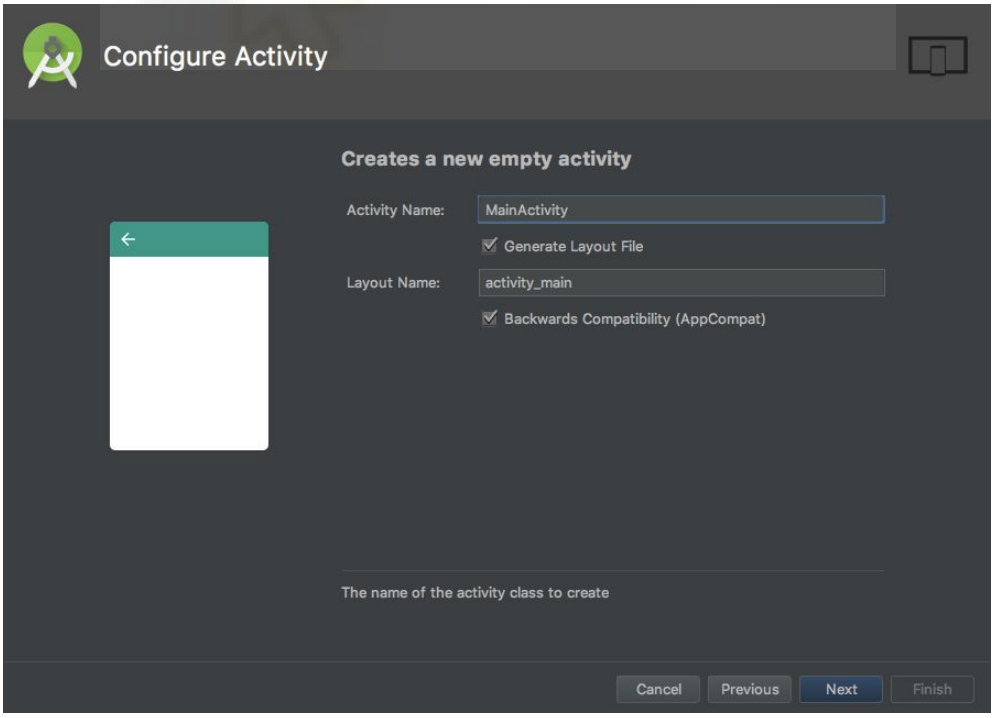
Ahora le damos a siguiente:



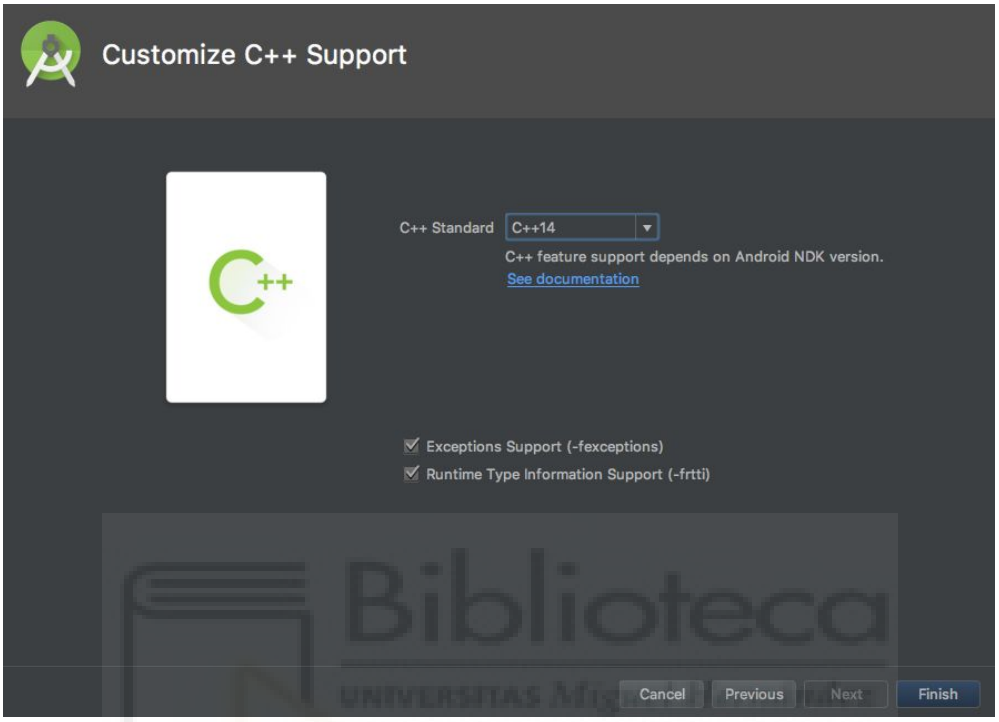
En el menú que ahora aparece nos pide el SDK que queremos utilizar, lo dejamos todo tal y como está y le presionamos de nuevo al botón de siguiente. Entonces aparecerá este menú:



Vamos a seleccionar Empty Activity, le damos a siguiente. En la siguiente ventana podemos especificar el nombre de la clase principal que queremos usar, en este caso lo he dejado tal y como aparece por defecto:

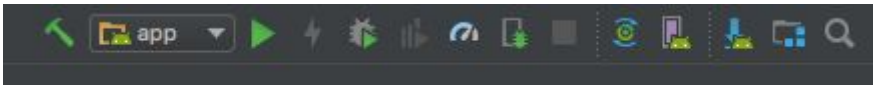


En la siguiente pantalla, que es la última, nos aparece las opciones de C++ que queremos activar. Se puede seleccionar lo que se quiera, pero en este caso aconsejo utilizar las opciones tal y como se muestran en la siguiente imagen:



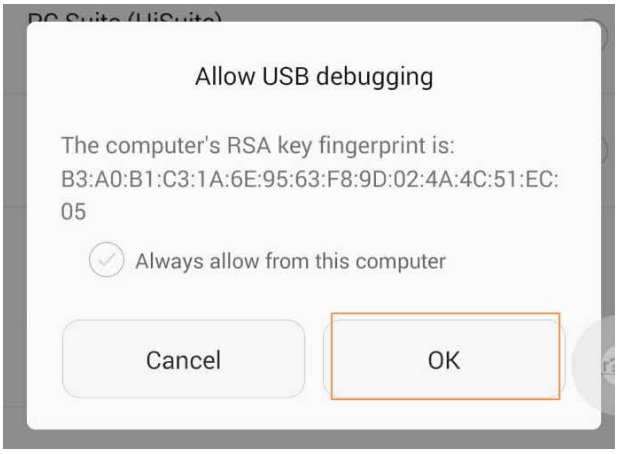
Le pulsamos a finalizar. Al haber hecho todos estos pasos, aparecerá ya el IDE de Android Studio con nuestro proyecto abierto el cual ya podemos compilar y ejecutar en el teléfono.

Para comprobar que todo está bien nada más crear nuestro proyecto, vamos a compilarlo y ejecutarlo, para ello hay que conectar el teléfono al ordenador mediante un cable USB. En la parte superior derecha nos aparece una serie de botones para poder lanzar nuestra aplicación:



El botón que lanzará nuestra aplicación en modo DEBUG es el siguiente:

Hay que tener una precaución, porque cuando le demos al botón de depurar, si es la primera vez que hacemos esto en un teléfono aparecerá un mensaje como el siguiente:

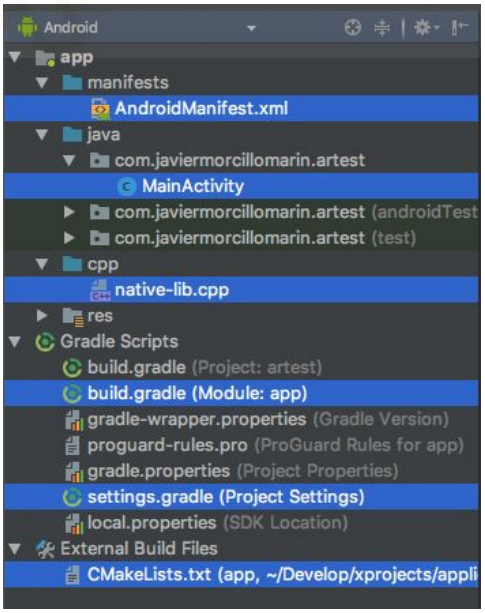


Hay que chequear la opción de “Siempre confiar en este teléfono” y presionar Ok. Veremos como en el teléfono aparece nuestra aplicación.

4.3.3.2. Activar ARCore

Con lo explicado en el apartado anterior tenemos ya un proyecto preparado para lanzarlo en el teléfono y además usando C++ en comunión con Java. El siguiente paso es configurarlo para poder soportar ARCore.

Lo primero es que tenemos que fijarnos en los archivos de nuestro proyecto, los más importantes son los que se marcan a continuación:



Lo primero que hay que hacer es abrir el CMakeLists.txt y añadir las siguientes líneas. Todo aquello que sea específico de ARCore se remarcará, al igual que se remarcará otras dependencias aconsejadas.

```
add_library(arcore SHARED IMPORTED)
set_target_properties(arcore PROPERTIES IMPORTED_LOCATION
                      ${ARCORE_LIBPATH}/${ANDROID_ABI}/libarcore_sdk_c.so)

add_library(   native-lib
               SHARED
               src/main/cpp/native-lib.cpp )

target_include_directories(native-lib PRIVATE src/main/cpp
               ${ARCORE_INCLUDE}
               ${ANDROID_NDK}/sources/third_party/vulkan/src/libs/glm)

target_link_libraries(native-lib
                      android
                      log
                      GLESv2
                      arcore)
```

Configurado de esta manera, además de usar ARCore también se usará la librería GLM que es una librería para el cálculo simple de matrices con fines 3D (matriz de posición, vectores, cuaterniones, ...).

Otra cosa indispensable es configurar el archivo de manifest, se han de incluir las siguientes líneas:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.javiermorcillomarin.artest">

        <uses-permission android:name="android.permission.CAMERA"/>
        <uses-feature android:name="android.hardware.camera.ar" android:required="true"/>
        <uses-feature android:glEsVersion="0x00020000" android:required="true" />

        <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <meta-data android:name="com.google.ar.core" android:value="required" />
        <activity android:name=".MainActivity">
                <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
                </intent-filter>
        </activity>
        </application>

</manifest>
```

Nada más poner lo que anteriormente se ha especificado, si intentamos compilar, fallará. Esto se debe a que debemos configurar nuestro proyecto para que se descargue ARCore y lo

ponga en cierto lugar para poder usarlo. En el archivo **build.gradle** del proyecto (no el de la app), se ha de poner lo siguiente:

```
// Top-level build file where you can add configuration options common to all
sub-projects/modules.

buildscript {
        repositories {
                google()
                jcenter()
        }
        dependencies {
                classpath 'com.android.tools.build:gradle:3.1.4'
                // NOTE: Do not place your application dependencies here; they belong
                // in the individual module build.gradle files
        }
}

allprojects {
        repositories {
                google()
                jcenter()
                mavenLocal()
        }
}

task clean(type: Delete) {
        delete rootProject.buildDir
}
```

En el **build.gradle** de la app (el de la aplicación), ponemos lo siguiente:

```
apply plugin: 'com.android.application'

/*
The arcore aar library contains the native shared libraries.  These are
extracted before building to a temporary directory.
 */
def arcore_libpath = "${buildDir}/arcore-native"

// Create a configuration to mark which aars to extract .so files from
configurations { natives }

android {
        compileSdkVersion 27
        defaultConfig {
        applicationId "com.google.ar.core.examples.c.helloar"

        // 24 is the minimum since ARCore only works with 24 and higher.
        minSdkVersion 24
        targetSdkVersion 27
        versionCode 1
        versionName "1.0"

        externalNativeBuild {
                cmake {
                cppFlags "-std=c++11", "-Wall"
                arguments "-DANDROID_STL=c++_static",
                        "-DARCORE_LIBPATH=${arcore_libpath}/jni",
                        "-DARCORE_INCLUDE=${project.rootDir}/../../libraries/include"
                }
```

```
        }
        ndk {
                abiFilters "arm64-v8a", "x86"
        }
        }
        buildTypes {
        release {
                minifyEnabled false
                proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
        }
        }
        externalNativeBuild {
        cmake {
                path "CMakeLists.txt"
        }
        }
}

dependencies {
        // ARCore library
        implementation 'com.google.ar:core:1.2.1'
        natives 'com.google.ar:core:1.2.1'

        implementation 'com.android.support:appcompat-v7:27.0.2'
        implementation 'com.android.support:design:27.0.2'
}

// Extracts the shared libraries from aars in the natives configuration.
// This is done so that NDK builds can access these libraries.
task extractNativeLibraries() {
        doFirst {
        configurations.natives.files.each { f ->
                copy {
                from zipTree(f)
                into arcore_libpath
                include "jni/**/*"
                }
        }
        }
}

tasks.whenTaskAdded {
        task-> if (task.name.contains("external") && !task.name.contains("Clean")) {
        task.dependsOn(extractNativeLibraries)
        }
}
```

Una vez hecho todos estos pasos, ya podríamos ejecutar nuestra aplicación utilizando ARCore. Faltaría ver cómo se enlaza C++ con Java a través de JNI, pero en el propio proyecto que se ha creado a través de una template hay ejemplos para ello.

4.4. Características técnicas

Conforme se pueda utilizar un motor de renderización, ya se pueden empezar a hacer pruebas con la librería. En las siguientes secciones se muestran las características encontradas. Hay que remarcar que en realidad no se tiene por qué pintar nada, pues es una librería de SLAM (en principio), pero las ventajas que se tiene a la hora de hacer los primeros

test y ver que efectivamente la librería funciona, no tiene color en cuanto a la comparación de usar el modo gráfico contra el modo texto.

4.4.1. Puntos característicos

Da la sensación de que ARCore es un sistema basado en la detección de puntos característicos, pero en ninguna parte de la documentación se pone de manifiesto qué tipo de puntos usa.

4.4.2. Puntos 3D

La primera versión que se hizo funcionar la librería fue una en la cual ARCore iba dando puntos en 3D y el motor los iba almacenando para poder hacer un mapa del mundo alrededor. Hay una cosa muy buena en la librería, y es que cuando la librería proporciona un punto 3D, además, junto al punto, devuelve un valor de "confianza". El significado de este valor de confianza es cómo de seguro está ARCore de que un punto 3D dado está efectivamente en la posición que nos dice.

Hay que remarcar, que ese factor de confianza pudiera ser el sigma de incertidumbre en el cual el punto 3D pudiera encontrarse. Una característica de ese valor es precisamente que sea solo uno, pues en 3D debieran aparecer un sigma para cada dimensión. Probablemente para facilidad de uso, nos calculará una media que será la confianza final del punto.

Sea como fuere, este valor es de vital importancia porque gracias a él, podemos almacenar y tener en cuenta solamente aquellos puntos que superen un umbral que nosotros marcamos. En las pruebas realizadas, al poner un umbral de 0.75 se conseguían valores bastante aceptables.

Otra ventaja que tiene al respecto, es que como se basa en odometría interna de acelerómetro y giroscopio, en sus cálculos, se tiene intrínsecamente el factor de escala que todo SLAM visual necesita para funcionar. Por lo tanto, las medidas que proporciona, son medidas ya en el mundo real.

4.4.3. SLAM local

Una vez que todo empieza a funcionar, es decir, ya se pueden visualizar la cámara en el teléfono y además se pintan los puntos 3D, quien utilice esta librería, se puede encontrar con una cosa muy curiosa: los puntos 3d desaparecen. Eso mismo me ocurrió a mi y, como buen desarrollador, mi primer pensamiento fue: "*He hecho algo mal*".

Haciendo pruebas se puede observar que el síntoma es: Cuando el móvil no sufre desplazamientos y sólo sufre giros, si un punto 3d que ya ha sido encontrado sale de la pantalla, se pierde para siempre. Da la sensación de que cuando se vuelve al frame anterior (antes de girar el móvil) y se mueve el teléfono, entonces, vuelve a resurgir el punto que

desapareció, pero no es así. Lo que ocurre es que casualmente vuelve a encontrar otro punto en el mismo sitio.

Eso hace pensar que ARCore hace SLAM sólo teniendo en cuenta los dos únicos frames y la odometría interna otorgada por el acelerómetro y el giroscopio. Este enfoque, realmente no es un SLAM conocido como tal, pero si se cataloga de librería capaz de hacer SLAM, hemos de decir que se trataría de un SLAM local.

Se pueden hacer las siguientes pruebas para poder inferir el funcionamiento anteriormente citado:

- Empieza el tracking, y cuando hayas encontrado varios puntos 3D, no muevas el teléfono sólo gíralo. Los puntos 3D desaparecen.
- Empieza el tracking, y cuando hayas encontrado varios puntos 3D, tapa la cámara, mueve el móvil enérgicamente, y vuelve a destapar la cámara. La información 3D que pudieras haber almacenado ya no es válida.
- Empieza el tracking, empieza a andar alrededor de un edificio, a medida que avances, los puntos almacenados dejan de estar en el sitio original, aunque vuelvas al mismo sitio.

Por lo tanto, mientras la cámara está activa y captando, utiliza los mismos puntos detectados para localizarse a modo de migas de pan. Cuando no hay puntos visibles, utiliza el giroscopio y el acelerómetro para seguir haciendo su “SLAM”, pero va perdiendo precisión en la posición de lo que ya había almacenado.

4.4.4. Feature Points

Un aspecto que es un grave error bajo mi punto de vista y que hace que no se pueda evolucionar demasiado con la librería es que no ofrece el descriptor del punto encontrado. La única razón que encuentro para que esto sea así, y no sea un grave error del API es que si se ofrecieran los descriptores, mediante ingeniería inversa se pudiese descubrir qué tipo de puntos característicos usa. Y aunque fuera así, me sigue pareciendo un grave error, puesto que internamente podría utilizar un hash para poder enmascarar el descriptor del punto característico, y devolver ese hash como descriptor a modo de entero.

Esto mismo se demuestra mirando el API de ARKit que se mencionó en el estado del arte. ARKit, pese a funcionar al parecer peor que ARCore, ofrece un número entero que sirve como descriptor del punto característico.

En un principio, antes de conocer el API de la librería, la idea original era:

1. Busco puntos 3D en el entorno.
2. Para cada punto 3D que encuentro, busco de entre los que ya tengo almacenados, y si encuentro un antiguo punto que esté muy cerca de ese nuevo punto y además sus descriptores coinciden, los fusiono según algún criterio (media, media ponderada, ...).

3. Los puntos nuevos que no encontraron coincidencias con los puntos anteriores, se añaden a la base de datos.

Al no contar con descriptores ni nada parecido, el único criterio posible para poder fusionar los nuevos puntos con los que ya hay almacenados es la distancia euclídea. Pero, como ya se ha mencionado anteriormente, si nos alejamos demasiado de una zona, o durante un tiempo perdemos la capacidad de auto-localizarnos con los puntos 3D que ya existen, el error es lo suficientemente grande como para aceptar este criterio.

4.5. Incorporaciones realizadas

En esta sección se va a describir la única incorporación realizada a esta librería. Como las otras ideas acerca de la extensión de la librería no se han realizado por temas de falta de tiempo, se detallarán en la sección de futuros trabajos.

4.5.1. Motor de renderizado

Antes incluso de empezar a funcionar con ARCore, sería muy aconsejable tener alguna idea de cómo pintar cosas en el dispositivo Android. Dado que el API predilecto es OpenGL, y dado a que ARCore lo que te ofrece son imágenes en una textura de OpenGL, hay que aprender un poco acerca de él.

Con la esperanza de poder usar más cómodamente la información gráfica, antes de atacar a la propia librería ARCore, se empezó a implementar un motor simple de renderizado intentando que, aunque fuese sencillo, fuese lo más escalable posible.

El mínimo indispensable para poder hacer funcionar ARCore y poder visualizar los datos que proporciona, es una textura en la cual pintar la imagen que se captura desde la cámara, y un renderizador de puntos. En el caso de este trabajo, los puntos se pintan a modo de esferas o aros de color. El color simboliza qué clase de punto se tiene.

5. ORB-SLAM2

ORB-SLAM2 es una librería desarrollada por profesores de la Universidad de Zaragoza con el fin de hacer SLAM con la odometría extraída únicamente de una cámara. Es la segunda implementación de otra llamada ORB-SLAM, aunque a partir de este momento, cuando hablemos de ORB-SLAM, nos estemos refiriendo a ORB-SLAM2. Los autores que figuran como desarrolladores de ORB-SLAM son:

- Raúl Mur-Artal
- Juan D. Tardos
- J. M. M. Montiel
- Dorian Galvez-Lopez

Sobre este último, Dorian Galvez-Lopez, aparece algo que llama la atención y es que aparece explícitamente como desarrollador del BoW. Si miramos en su repositorio público de github (https://github.com/dorian3d), podemos decir que se muestra, efectivamente, como el autor de BoW (Bag of Words) y algún sistema de tracking de cámaras RGBD. Lo curioso es que tiene también una librería que se llama DLoopDetector, la cual tiene toda la pinta de ser el detector de loop closing. Con respecto a DLoopDetector, hay otro dato curioso y es que recientemente parece haber borrado el código, al parecer porque usaba puntos surf que tiene licencia de propietario.

5.1. Características

Como ya se ha citado, tiene como objeto el proporcionar un sistema de SLAM únicamente basado en información visual. Es Open Source con licencia GPLv3, por lo que gracias a los autores, podemos ver los entresijos de esta librería. El código fuente de la librería se encuentra en esta URL https://github.com/raulmur/ORB_SLAM2.

ORB-SLAM es capaz de trabajar con cámaras monoculares, binoculares y con cámaras RGBD. Al estar escrita en C++, en principio es multiplataforma pura, es decir, no se limita sólo a los ordenadores, sino que en cualquier dispositivo debería de poder funcionar, como por ejemplo los dispositivos móviles.

Como su nombre indica, ORB-SLAM está basado en la búsqueda de puntos característicos ORB, y en la capacidad de encontrar transformaciones entre frames haciendo coincidir dichos puntos característicos. Además tiene un módulo de “loop closing” por lo que continuamente está intentando reencontrarse y corregir el error cometido en el paso del tiempo y el espacio.

Esta librería además de poder ser compilada y ejecutada en cualquier plataforma, funciona en ROS.

5.1.1. Funcionamiento general de ORB-SLAM

Esta librería puede funcionar en dos modos. El primero es el SLAM puro y duro, es decir, la cámara al moverse, va generando un mapa de lo que tiene alrededor a la par que se va localizando dentro de ese mapa. El otro modo consiste en, una vez detectado un entorno, poder moverse localizándose dentro de dicho mapa, pero sin generar ninguna información nueva.

Cuando está en modo de SLAM, en esencia, la librería tiene 3 estados diferentes: "iniciando", "trackeando", y "perdido". Su significado es el siguiente:

- Nada más empezar a trabajar, el sistema se pone como "iniciando" e intenta coger de entre algunos frames información suficiente como para ponerse en marcha. Es decir, intenta extraer información 3D y de la pose de la cámara usando unos pocos frames.
- Una vez encontrado una serie de frames suficientes para iniciarse y con los que puede localizar la cámara y los puntos 3D, se pone en estado de "tracking" e intenta seguir en este modo siempre. Cuando está en este modo, va generando KeyFrames y puntos 3D para modelar el entorno por donde se mueve el dispositivo.
- Si de repente un Tracker no puede encontrar alguna correspondencia con el frame anterior (o anteriores) con la que se pueda generar información 3D, se pone en estado de "perdido", e intenta en todo momento relocalizarse. Si nunca más pasamos ante los lugares donde se ha estado antes, siempre estará en modo de "perdido" hasta que lo reseteemos explícitamente.

Existen varias clases en la librería, pero hay tres especialmente importantes porque son las que ponen en marcha todo. Evidentemente, a parte de las que se comentan en esta sección, hay muchas otras que son también parte esencial, pero son más contenedoras o auxiliares que otra cosa. Digamos que el corazón del sistema comprende las siguientes clases:

- Tracking: Esta clase es la encargada de coger un frame, buscar los puntos característicos e intentar hacer un seguimiento instantáneo del frame actual que se acaba de generar.
- LocalMapping: Esta clase es la encargada de construir el mapa local al Tracker de la mejor manera posible. Genera la visibilidad entre frames y los optimiza para almacenar el mínimo número de ellos.
- LoopClosing: Esta clase es la encargada en todo momento de detectar si hemos cerrado algún bucle de SLAM para poder hacer una corrección de la información 3D.

Cada una de las clases anteriores se ejecuta en un thread por separado separado. Pese a que este sistema, bajo mi punto de vista es el que toca, en su implementación se ha generado una serie de problemas que se relatan en secciones posteriores.

5.2. Requisitos

ORB-SLAM no tiene en realidad ningún requisito más allá de las librerías que usa para poder ser compilada. Como es código de C++, cualquier sistema existente hoy en día soportaría la compilación y la ejecución de la librería.

Si bien es cierto que funciona un poco en el límite del tiempo real, sí que se podría decir que requiere, para poder funcionar bien, un hardware que no sea muy básico. Un doble núcleo, a 2GHz y 4 GB de RAM podría ser aceptable. Como la carga del diccionario que usa para funcionar es un poco lenta, también sería recomendable un SSD.

5.3. Instalación de ORB-SLAM

ORB-SLAM tiene algunas dependencias para poder ser compilada y ejecutada. Las librerías que usa y el por qué son las que a continuación se detallan:

5.3.1. Librería OpenCV

Como ya se ha comentado en el estado del arte, OpenCV es la librería por excelencia para cualquiera que se dedique a la investigación de sistemas basados en visión. Si bien es cierto que es algo más lentas que otras librerías, su uso es sobretodo académico.

ORB-SLAM usa esta librería entre otras cosas para el cálculo del SVD, para la detección de puntos FAST, ...

La forma de usarla es muy diferente y dependerá más de nuestros vicios adquiridos como programadores que marcarán nuestra manera predilecta. En OSX, una forma muy sencilla de usarla es ejecutar “brew install opencv”, esto instalará en el sistema la OpenCV.

Para Android el asunto se complica un poco más, pues hay que descargar una build que contenga toda la OpenCV. La explicación de cómo usarla como “prebuilt library” la podemos encontrar en esta URL: https://opencv.org/platforms/android/. Hay que comentar también que en Android, la OpenCV se tiene que bajar como paquete externo, esto es, desde Google Play, buscamos OpenCV Manager e instalamos el paquete.

De cuaquier forma, siempre se puede descargar desde https://opencv.org/ e intentar hacerla compilar, pero no lo recomiendo.

Existe otro comentario acerca de esta librería en Android, el cual aparecerá en la sección de compilación.

5.3.2. Librería g2o

De esta librería ya se ha hablado en el estado del arte, para hacerla funcionar se puede descargar el código fuente desde https://github.com/OpenSLAM-org/openslam_g2o. En

realidad es suficiente con meter en nuestro proyecto las partes de core, solvers, stuff y types. También hay que crear a mano un archivo tal como este:

```
#ifndef G2O_CONFIG_H
#define G2O_CONFIG_H
// give a warning if Eigen defaults to row-major matrices.
// We internally assume column-major matrices throughout the code.
#ifdef EIGEN_DEFAULT_TO_ROW_MAJOR
#  error "g2o requires column major Eigen matrices (see
http://eigen.tuxfamily.org/bz/show_bug.cgi?id=422)"
#endif
#endif
```

5.3.3. Librería Eigen

Hay que tener cuidado con esta librería. Es necesario descargar la versión 3.3, y aún así no hemos terminado. Nada más compilar, van a aparecernos varios errores de compilación. Es posible que los autores compilen con un compilador diferente al que se ha usado en este trabajo. Concretamente con el Clang de Xcode, la compilación falla. Hay que modificar algunas líneas de código, y poner algún que otro "const".

Por ejemplo, en la implementación original de ORB-SLAM aparece la siguiente línea:

```
typedef map<KeyFrame*,g2o::Sim3,std::less<KeyFrame*>,
      Eigen::aligned_allocator<std::pair<const KeyFrame*, g2o::Sim3> > >
KeyFrameAndPose;
```

Esa línea falla. No es fácil descubrir el por qué, pero hay una razón, evidentemente. La razón es que para Eigen, lo que tiene que ser "const" en este caso concreto, no es el objeto, sino el puntero. Se soluciona cambiando el const de sitio como se muestra a continuación.

```
typedef std::map<KeyFrame*,g2o::Sim3,std::less<KeyFrame*>,
       Eigen::aligned_allocator<std::pair<KeyFrame*const, g2o::Sim3> > >
KeyFrameAndPose;
```

Se ha intentado usar versiones más actuales de Eigen, pero el número de errores encontrados es aún mayor, y más difíciles de resolver.

5.3.4. Librería Pangolín

Esta librería se usa para pintar exclusivamente. La podríamos quitar como dependencia, de hecho no se usa en Android porque no tiene sentido ya que no tenemos una pantalla de desktop como tal.

Para poder usarla, podemos usar el comando "brew install pangolin" o descargarla desde su repositorio https://github.com/stevenlovegrove/Pangolin. Lo cierto es que a la hora de

compilarla y ejecutar sus ejemplos no ofrece ningún problema. Pocas librerías pueden decir esto.

5.3.5. Librería DBoW2

Esta librería pertenece a uno de los autores del ORB-SLAM como ya se ha comentado, concretamente a Dorian. El repositorio donde podemos encontrar esta librería es uno de los que este profesor tiene: https://github.com/dorian3d/DBoW2.

Es una librería que se encarga de implementar un Bag of Words. Esto hace que se pueda generar una especie de hash para algunos descriptores y de esa manera poder hacer una búsqueda muy rápida para hacer matching entre frames y puntos.

5.3.6. Compilación en Android

Una vez conseguida la compilación en Desktop (en el caso de este trabajo, Mac con OSX), el siguiente paso era intentar compilar y usar la librería de ORB-SLAM en Android, pues este era un objetivo a cumplir. Para hacerlo hay que tener en cuenta lo siguiente:

- Si se intenta bajar alguna aplicación de las implementaciones que hay en github de ORB-SLAM, probablemente fallará la compilación.
- Si se consigue que compile a la primera, probablemente el programa fallará cuando se ejecute en el móvil ya que por algún motivo probablemente llevará precompilada la librería de ORB-SLAM y todas fallan aparentemente.
- Si se intenta bajar un repositorio que tenga el código por compilar, probablemente se caiga en trabajo innecesario puesto que las librerías thirdparty que vienen en esos repositorios no son todas necesarias. Por lo tanto se invierte mucho tiempo en descubrir cómo lincar y compilar dichas librerías.
- Después de tener preparadas las librerías thirdparty, la mala noticia viene con que compilar ORB-SLAM para Android tarda entre 1 y 3 minutos dependiendo de qué ficheros toques. Debido a esto, se optó por seguir el desarrollo en el ordenador y llegado el momento, intentar llevarlo a cabo en el móvil. Por cierto, tarda entre 1 y 3 minutos si te das cuenta de que tienes que preparar el gradle para que compile en una sóla plataforma (por ejemplo armv7), si no, la compilación entera tarda unos 10 a 15 minutos.
- Cuando finalmente se logra ejecutar con éxito la librería en el móvil, probablemente fallará porque a la librería le hace falta ciertos archivos para poder funcionar, como es por ejemplo el archivo del vocabulario del ORB.

Una cosa importante a tener en cuenta cuando se usa OpenCV en la versión de Android para hacer funcionar ORB-SLAM, es que va a darnos un error de lincaje. El tema es que parece ser que en la versión de Android no existen las llamadas a los puntos característicos FAST. Hay que usar una clase llamada FastFeatureDetector.

La forma de crear un FastFeatureDetector es la siguiente:

```
mFastDetector = FastFeatureDetector::create(iniThFAST, true);
```

La manera de usarlo es la siguiente. Se pone un ejemplo extraído de ORB-SLAM, de manera que cuando el compilador nos dé un error de lincaje, saber cómo hemos de actuar. Se pone como comentario la forma que da el error, y como código normal la forma correcta a usar.

```
//FAST(mvImagePyramid[level].rowRange(iniY,maxY).colRange(iniX,maxX),
vKeysCell,iniThFAST,true);
mFastDetector->detect(mvImagePyramid[level].rowRange(iniY,maxY).colRange(iniX,max
X), vKeysCell);
```

Otra cosa que hay que hacer es cambiar una línea como la que sigue, pues no existe en la versión de Android:

```
KeyPointsFilter::retainBest(keysCell,nToRetain[i][j]);
```

En vez de dicha línea hay que usar una línea propia que necesita una función programada por nosotros. Para ver lo que hacía esa función, lo que se ha hecho es ir al código fuente de la OpenCV y se ha copiado la siguiente función.

Esta es la llamada correcta:

```
missingRetainBest(keysCell,nToRetain[i][j]);
```

Y esta es la función:

```
struct KeypointResponseGreater
{
    inline bool operator()(const KeyPoint& kp1, const KeyPoint& kp2) const
    {
        return kp1.response > kp2.response;
    }
};

struct KeypointResponseGreaterThanThreshold
{
    KeypointResponseGreaterThanThreshold(float _value) :
    value(_value)
    {
    }
    inline bool operator()(const KeyPoint& kpt) const
    {
        return kpt.response >= value;
    }
```

```
    float value;
};

void missingRetainBest(std::vector<KeyPoint>& keypoints, int n_points)
{
    if( n_points >= 0 && keypoints.size() > (size_t)n_points )
    {
        if (n_points==0)
        {
            keypoints.clear();
            return;
        }
        std::nth_element(keypoints.begin(), keypoints.begin() + n_points,
keypoints.end(), KeypointResponseGreater());
        float ambiguous_response = keypoints[n_points - 1].response;
        std::vector<KeyPoint>::const_iterator new_end =
        std::partition(keypoints.begin() + n_points, keypoints.end(),
                       KeypointResponseGreaterThanThreshold(ambiguous_response));
        keypoints.resize(new_end - keypoints.begin());
    }
}
```

Con esto la versión de Android ya se debería poder compilar y ejecutar.

5.4. Ejecución de ORB-SLAM

Una vez se tiene todo lo necesario para poder compilar y ejecutar el programa, hay que notar que hacen falta los siguientes archivos:

- Un fichero para configurar la cámara y ciertos parámetros. En el ejemplo de monocular que viene con la librería, los podemos encontrar bajo el nombre de TUM1.yalm, TUM2.yalm, ...
- Un fichero de banco de pruebas que se ha de descargar de la siguiente URL: https://vision.in.tum.de/data/datasets/rgbd-dataset/download para poder coger un dataset de prueba. Por ejemplo, se puede bajar del repositorio el dataset llamado “freiburg3_long_office_household” y se puede hacer funcionar con TUM3.yalm.
- El vocabulario a usar por el BoW. Bajo el directorio Vocabulary del repositorio de la librería, hay un fichero llamado ORBvoc.txt.tar.gz, el cual se debe descomprimir y es el fichero que se tiene que usar como vocabulario.

Una vez conseguido todo esto, surgen un par de problemas que se comentan en las secciones siguientes.

5.4.1. Problemas con el drawer

Nada más bajar la librería y todos los ficheros necesarios y poder compilar y ejecutar alguno de los tests que tienen, depende de qué sistema operativo se esté usando (en mi caso OSX), cuando se le da al play para hacerlo funcionar, el programa da un error.

Eso es un jarro de agua fría en toda regla, ya que el hacerlo compilar y ejecutar cuesta bastante tiempo, y una vez se cree que lo se tiene todo, algo que se escapa al control de cada uno falla.

Este fue el primer problema encontrado después de conseguir compilar y ejecutar el test y en realidad fue el primer error de diseño encontrado.

Con total seguridad, la intención de Raúl Mur-Artal, Juan D. Tardos, J. M. M. Montiel y Dorian Galvez-Lopez es didáctica, y por eso han programado la librería sin tener en cuenta ciertos detalles a mi modo de ver. En este caso, en mi humilde opinión, si hubiese tenido que implementar la librería, con una posible comercialización de la misma, no hubiese puesto el drawer como parte interna de la arquitectura del sistema. Dicho de otro modo, para poder hacer SLAM, tal y como se encuentra la implementación actual, se necesita poder pintar.

Es más, para poder pintar, encima se usa una librería que puede no existir en nuestro sistema, y por lo tanto puede que necesitemos descargarla y compilarla por nosotros mismos. Aunque he de recordar, que en este caso, la librería Pangolín, se compila con mucha facilidad.

Creo que hubiese sido mucha mejor idea diseñar el drawer como un módulo totalmente al margen de la implementación de la librería. De esta manera, como usuario de la librería, si quiero pinto, y si no quiero no.

Investigando el problema, y haciendo varias pruebas, todo parecía apuntar a que, o bien la librería que usan para pintar (Pangolin) no soporta multithread, o bien el OSX no soporta multithread para pintar. Esto no es inmediato de ver, sino que se tarda un tiempo realizando pruebas y un tiempo de investigación para poder inferirlo.

En este caso se descubrió que, al parecer, el error radicaba en intentar pintar algo en pantalla desde un thread que no era en concreto el thread principal de la aplicación. Al parecer, esto es es una cosa no permitida en OSX. Debido a este posible error de ejecución (pintar los frames en un thread que no es el principal), es necesario extraer el drawer y llamarlo desde el thread principal. Una vez hecho esto, que no es ni mucho menos inmediato, el programa ya funciona y pinta correctamente.

5.4.2. Tiempo real

Como ya se ha comentado, la implementación se llevó a cabo en el Ordenador de sobremesa. El ordenador en cuestión es un Mac i7 (4 nucleos) 2.2GHz con 8GB de RAM. El framerate conseguido en este equipo fue de entorno a 18-20 FPS cuando el sistema está en modo tracking y en modo RELEASE. Esos frames tienen en cuenta sólo el proceso de tracking, no tienen en cuenta la toma de imágenes de un fichero, ni el pintado ni nada más que no sea tracking puro.

He de decir, que aunque está bastante bien y se puede considerar tiempo real perfectamente, a no ser que tengamos un móvil de última generación es de prever que esos frames bajasen

significativamente. Por lo tanto, se tomó una nueva línea de desarrollo basada en el ordenador de sobremesa dejando la versión de Android para un futurible.

Además, cuando se ejecuta en modo DEBUG, como es en la mayoría de casos de la vida del desarrollo software, los FPS bajaban a 2-3 FPS. Con respecto al tiempo real, se comentará algún detalle más en secciones posteriores.

5.4.3. Resultados de la ejecución

Como el resultado de la ejecución, aunque bueno, parecía no ser suficiente para dispositivos móviles, y sobre todo en modo DEBUG surge una nueva línea de investigación. En esta nueva línea de desarrollo, antes de intentar introducir ORB-SLAM en el móvil, lo que se intentó fue desarrollar una idea diferente, la cual a mi juicio, es quizás más atractiva que hacer funcionar la librería en un móvil: usar el Ordenador como servidor al cual se le pudiesen ir conectando dispositivos y se pudiese hacer el tracking de ellos de manera simultánea. Al mismo tiempo, se podría investigar si sería posible o no el intentar que los diferentes móviles generasen un solo mapa del entorno.

Por lo tanto, en esta nueva línea, es necesario modificar la librería para que pudiese soportar los nuevos requerimientos.

En las secciones posteriores comentaré los problemas encontrados, el punto al cual se ha llegado dentro de este trabajo, y las futuras posibles líneas de desarrollo que se podrían seguir para lograr hacer multitracking con un solo mapa.

5.5. Problemas con la implementación de ORB-SLAM

Sin ánimo de querer menospreciar el trabajo realizado por los autores anteriormente mencionados, ya que han realizado un **trabajo digno de elogio**, sí que creo haber visto algunas cosas mejorables en términos de arquitectura de software. En esta sección relataré lo que a mi juicio son los puntos problemáticos encontrados, o como mínimo discutibles. Estos puntos conflictivos, si es que se pueden llamar así, son los que han hecho difícil la modificación de la librería para desarrollar características nuevas como por ejemplo el multitracking con un sólo mapa. Puede que si estuviese más tiempo desarrollándola, y quizás desde sus inicios, todo hubiese sido más fácil.

5.5.1. Problema de acoplamiento

Bajo mi punto de vista la librería, este es uno de los dos mayores defectos que he encontrado, siempre claro está, bajo mi humilde punto de vista. Lo primero que se debe hacer para entender este punto es definir qué es en ingeniería de software el acoplamiento de un sistema.

Se entiende como el nivel de acoplamiento de un sistema como la interdependencia que existe entre los módulos del mismo. De tal manera, que en un sistema fuertemente acoplado,

la dependencia entre los módulos que lo componen es muy fuerte, mientras que en un sistema débilmente acoplado los módulos del mismo dependen muy poco los unos de los otros.

Puede parecer por las palabras utilizadas que es preferible tener un sistema fuertemente acoplado, pero es precisamente lo contrario, lo deseable en cualquier sistema, es diseñarlo de la manera menos acoplada posible. Esto se debe a que cuando nos encontramos ante un sistema fuertemente acoplado, cuando modificamos un módulo, lo más probable es que debamos de modificar los módulos que se alimentan de él. A su vez, por cada módulo que hayamos modificado, en forma de cascada, probablemente debamos modificar otros módulos a su vez. Por contrapartida, en un sistema débilmente acoplado, cuando modificamos un módulo, es muy improbable que necesitemos modificar otros módulos que dependan de este primero.

Para poner de manifiesto el párrafo anterior y defender esta postura vamos a poner un ejemplo genérico y luego uno específico de la librería. Digamos que tenemos un módulo de tal manera:

Clase A
color: entero nombre: string

Por otro lado tenemos estos dos módulos:

Clase B

método calcularColor(UnObjeto: A): entero
{
  devolver UnObjecto.color;
}

Clase C

método sumarColores(UnObjeto: A, OtroObjeto: A): entero
{
  devolver UnObjecto.color + OtroObjeto.color;
}

Es decir, tenemos una clase A con dos miembros, una clase B con un método que devuelve un color dado un objeto y otra clase C que suma los colores de dos clases A. Por alguna necesidad, en un futuro, se estima que se se ha cometido un error y que el miembro color de

un objeto de la clase A debiera calcularse de la siguiente manera, “Si el nombre está en mayúsculas, color debe de ser negativo”.

Debido a ese nuevo requerimiento, ahora estaríamos obligados a hacer:

Clase B

método calcularColor(UnObjeto: A): entero
{
  si (mayúsculas(UnObjeto.nombre))
  devolver -UnObjecto.color;
  si no
  devolver UnObjecto.color;
}

Clase C

método sumarColores(UnObjeto: A, OtroObjeto: A): entero
{
  color1, color2: entero;
  si (mayúsculas(UnObjeto.nombre))
  color1 := -UnObjecto.color;
  si no
  color1 := UnObjecto.color;
 	
  si (mayúsculas(OtroObjeto.nombre))
  color2 := -OtroObjeto.color;
  si no
  color2 := OtroObjeto.color;
  devolver color1 + color2;
}

Como se puede apreciar, estamos ante un sistema fuertemente acoplado, pues la modificación de algo en un módulo provoca otras variaciones en cascada. La solución a este problema hubiera sido hacerlo un poco menos acoplado, como se ve a continuación:

Clase A
color: entero nombre: string
método devolverColor(): entero {   devolver color;

```
}
```

Clase B

método calcularColor(UnObjeto: A): entero
{
  devolver UnObjecto.devolverColor();
}

Clase C

método sumarColores(UnObjeto: A, OtroObjeto: A): entero
{
  devolver UnObjecto.devolverColor() + OtroObjeto.devolverColor();
}

Con este nuevo enfoque, cuando nació la necesidad futura y extraña de cambiar el comportamiento del color de A, la clase B y la clase C no sufrirían ninguna modificación, la única modificación que habría que hacer es la siguiente:

Clase A
color: entero nombre: string

método devolverColor(): entero
{
  si (mayúsculas(nombre))
  devolver -color;
  si no
  devolver color;
}

Al diseñar un software, el problema del acoplamiento se puede minimizar usando funciones en vez de accesos a variables internas de una clase, y más aún se minimiza mucho más si en vez de clases se usan interfaces (como por ejemplo las que tiene Java). Además, se minimiza aún más cuando no hay interdependencia entre los módulos.

Por poner un ejemplo concreto de la librería, voy a poner la cabecera de una clase, y la manera en la que, bajo mi punto de vista hubiese ganado menos acoplamiento.

```
class Tracking
{
public:
   Tracking(...);
    cv::Mat GrabImageStereo(const cv::Mat &imRectLeft,const cv::Mat &imRectRight, const
double &timestamp);
      cv::Mat GrabImageRGBD(const cv::Mat &imRGB,const cv::Mat &imD, const double
&timestamp);
   cv::Mat GrabImageMonocular(const cv::Mat &im, const double &timestamp);
   void SetLocalMapper(LocalMapping* pLocalMapper);
   void SetLoopClosing(LoopClosing* pLoopClosing);
   void SetViewer(Viewer* pViewer);
   void ChangeCalibration(const string &strSettingPath);
   void InformOnlyTracking(const bool &flag);
public:
   // Tracking states
   enum eTrackingState{
       SYSTEM_NOT_READY=-1,
       NO_IMAGES_YET=0,
       NOT_INITIALIZED=1,
       OK=2,
       LOST=3
   };

   eTrackingState mState;
   eTrackingState mLastProcessedState;
   int mSensor;
   Frame mCurrentFrame;
   cv::Mat mImGray;
   std::vector<int> mvIniLastMatches;
   std::vector<int> mvIniMatches;
   std::vector<cv::Point2f> mvbPrevMatched;
   std::vector<cv::Point3f> mvIniP3D;
   Frame mInitialFrame;
   list<cv::Mat> mlRelativeFramePoses;
   list<KeyFrame*> mlpReferences;
   list<double> mlFrameTimes;
   list<bool> mlbLost;
   bool mbOnlyTracking;
   void Reset();
...
};
```

Como desarrollador, me hubiese gustado más encontrarme con algo parecido a los siguiente:

```
class Tracker {
    enum State {
        SYSTEM_NOT_READY = -1,
        NO_IMAGES_YET = 0,
        NOT_INITIALIZED = 1,
        OK = 2,
        LOST = 3,
        FINALIZING = 4
    };
    virtual bool    isBussy() const = 0;
    virtual void    addTrackLogger(Logger*) = 0;
    virtual void    removeTrackLogger(Logger*) = 0;
    virtual int     getTrackerId() const = 0;
    virtual auto    track(Frame* AFrame) -> const cv::Mat = 0;
    virtual auto    getInitialFrame() -> Frame* = 0;
    virtual auto    getCurrentFrame() -> Frame* = 0;
    virtual auto    getLastFrame() -> Frame* = 0;
```

```
    virtual auto    getState() const -> State = 0;
    virtual void    forceLost() = 0;
    virtual auto    getLastProcessedState() const -> State = 0;
    virtual bool    isOnlyTracking() const = 0;
    virtual auto    getIniMatches() const -> const std::vector<int>& = 0;
    virtual void    activateLocalizationMode() = 0;
    virtual void    deactivateLocalizationMode() = 0;
    virtual void    notifyReset() = 0;
    virtual void    setLocalMapper(LocalMapping* pLocalMapper) = 0;
    virtual void    setLoopClosing(LoopClosing* pLoopClosing) = 0;
    virtual auto    getLocalMapper() -> LocalMapping* = 0;
    virtual auto    getLoopClosing() -> LoopClosing* = 0;
    virtual auto    getMap() -> Map* = 0;
    virtual auto    getMap() const -> const Map* = 0;
    virtual auto    getDatabase() -> KeyFrameDatabase* = 0;
    virtual auto    getDatabase() const -> const KeyFrameDatabase* = 0;
    virtual void    flush() = 0;
};
```

He puesto este ejemplo concreto de los que podía haber puesto más que nada por la extensión del texto.

Mirando la cabecera original anterior, si soy un nuevo desarrollador del sistema, yo pensaría cosas del estilo: "*Vaya, la variable eTrackingState mState; es una variable pública, eso significa que puedo cambiarla en cualquier momento? o …, cuáles de todos lo módulos será capaz de cambiar esa variable*". En cambio, en la segunda cabecera no hay lugar a dudas, sólo tengo un getState, y además de tipo const, al ver esto pensaría: "*Vale, la clase que implemente al Tracker será la responsable de modificar su estado, y además, llamar a dicha función no altera el estado del objeto*".

Aunque esta sección sea, a mi parecer, una crítica constructiva acerca del acoplamiento de las clases, si que hay que decir algo a favor de los autores. Es posible que, por poner un ejemplo, con la clase Tracker, hubieran pensado en tener varias implementaciones del Tracker. Con la forma que yo he propuesto, eso se complicaría en esencia un poco, mientras que con la suya, se podría heredar de la clase Tracker sin problemas a la de ya. Esto se debe a que la gran esencia y dolencia de las interfaces, es ocultar su definición en las partes privadas del código.

Por ejemplo, imaginemos que tengo una interfaz llamada A, e implemento un backend llamado B que es totalmente privado para salvaguardar mi código. Eso imposibilita la capacidad de heredar de B.

Por otro lado, se puede llegar a un consenso, y es utilizar interfaces, y que los backends de las mismas se piensen para posibilitar de manera muy sencilla el poder heredar de ellas.

Otro factor a comentar en esta sección que demuestra el gran acoplamiento de las clases, y que en C++ es especialmente notorio, es que en las cabeceras de las mismas está plagado de "*forward declarations*". Esto es lo mismo que decir: "*Para que A funcione, se necesita utilizar un B, pero para que B funcione, se necesita utilizar un A*". Por poner un ejemplo del código original, podemos encontrarnos esto:

```
class KeyFrame;
class Map;
class Frame;

class MapPoint
{
...
};
```

Es decir, para que el MapPoint funcione, necesita a KeyFrame, Map y Frame, pero por otro lado, si miramos la cabecera de Frame:

```
class MapPoint;
class KeyFrame;

class Frame
{
  ...
};
```

Es decir, que Frame necesita a MapPoint, y que a su vez, MapPoint necesita a Frame. Bajo mi parecer, se podría haber diseñado el sistema de otra manera que hiciese desaparecer dichas interdependencias.

5.5.2. Problema de cohesión

En mi honesta y humilde opinión, pienso que junto con el apartado de acoplamiento, este es otro gran aspecto de lo que la librería adolece un poco.

En un primer vistazo, cuando se abre la librería para ver el código fuente por primera vez, todo parece estar bien organizado: la clase Frame, la clase KeyFrame, Tracker, … Pero cuando llegamos al demonio de los detalles, creo que cada clase se podría haber separado un poco más de lo que está.

La cohesión de un sistema establece la capacidad de que cada módulo que lo compone ha de ocuparse de resolver sólo un problema concreto. Es evidente que la la alta cohesión es deseable en cualquier diseño software, y es por ello que es deseable que cada módulo resuelva sólo un único problema.

Por poner un ejemplo, si la librería de ORB-SLAM sólo tuviese una sóla clase, estaría muy poco cohesionado y por lo tanto tendría un mal diseño. Contrariamente a esto, como se ha comentado, parece que la librería tiene un nivel adecuado de cohesión, pero a mi parecer, una vez entrados en detalles, falta un poco de cohesión.

Por poner un ejemplo concreto, si nos fijamos en la clase Frame, tenemos lo siguiente (de manera resumida):

```
class Frame
{
   void ExtractORB(int flag, const cv::Mat &im);
   void ComputeBoW();
   void SetPose(cv::Mat Tcw);
   void UpdatePoseMatrices();
   inline cv::Mat GetCameraCenter(){
       return mOw.clone();
   }
   inline cv::Mat GetRotationInverse(){
       return mRwc.clone();
   }
   bool isInFrustum(MapPoint* pMP, float viewingCosLimit);
   bool PosInGrid(const cv::KeyPoint &kp, int &posX, int &posY);
    vector<size_t> GetFeaturesInArea(const float  &x, const float  &y, const float  &r,
const int minLevel=-1, const int maxLevel=-1) const;
   void ComputeStereoMatches();
   void ComputeStereoFromRGBD(const cv::Mat &imDepth);
   cv::Mat UnprojectStereo(const int &i);


   ORBVocabulary* mpORBvocabulary;
   ORBextractor* mpORBextractorLeft, *mpORBextractorRight;
   double mTimeStamp;
   cv::Mat mK;
   static float fx;
   static float fy;
   static float cx;
   static float cy;
   static float invfx;
   static float invfy;
   cv::Mat mDistCoef;
   float mbf;
   float mb;
   float mThDepth;
   int N;
   std::vector<cv::KeyPoint> mvKeys, mvKeysRight;
   std::vector<cv::KeyPoint> mvKeysUn;
   std::vector<float> mvuRight;
   std::vector<float> mvDepth;
   DBoW2::BowVector mBowVec;
   DBoW2::FeatureVector mFeatVec;
   cv::Mat mDescriptors, mDescriptorsRight;
   std::vector<MapPoint*> mvpMapPoints;
   std::vector<bool> mvbOutlier;
   static float mfGridElementWidthInv;
   static float mfGridElementHeightInv;
   std::vector<std::size_t> mGrid[FRAME_GRID_COLS][FRAME_GRID_ROWS];
   cv::Mat mTcw;
   static long unsigned int nNextId;
   long unsigned int mnId;
   KeyFrame* mpReferenceKF;
   int mnScaleLevels;
   float mfScaleFactor;
   float mfLogScaleFactor;
   vector<float> mvScaleFactors;
   vector<float> mvInvScaleFactors;
   vector<float> mvLevelSigma2;
   vector<float> mvInvLevelSigma2;
   static float mnMinX;
   static float mnMaxX;
```

```
    static float mnMinY;
    static float mnMaxY;
    static bool mbInitialComputations;
};
```

Aparentemente la clase no tiene ningún problema de cohesión, pero si nos fijamos atentamente, esta clase tiene:

- Información de la disposición de los puntos característicos.
- Información de la pose del frame.
- Información de la cámara (sobre este tema también se hablará en secciones posteriores)
- Información de los puntos en 3D

Por lo tanto, yo defiendo la postura de que esta clase se debería haber fragmentado mucho más, al menos para satisfacer alguno de los puntos anteriores. Un trabajo que se ha hecho al respecto es separar esa clase en varias, y por el camino, alguna de las clases creadas han servido para otras clases con similar problemática. A continuación, se exponen algunas clases que han aparecido al intentar cohesionar la clase Frame.

ImageFeatures es una clase nueva que contiene información acerca de los puntos característicos. Esta clase es muy importante para el desarrollo del TFM puesto que permitirá que dos frames cualquiera compartan la misma información visual.

```
class ImageFeatures {
    static  ccore::ptr<ImageFeatures>   newImageFeatures(...);
    // Bag of Words Vector structures.
    DBoW2::BowVector mBowVec;
    DBoW2::FeatureVector mFeatVec;
    // ORB descriptor, each row associated to a keypoint.
    cv::Mat mDescriptors, mDescriptorsRight;
    // Vocabulary used for relocalization.
    ccore::ptr<Vocabulary> mpORBvocabulary;
    // Feature extractor. The right is used only in the stereo case.
    ccore::ptr<ORBextractor> mpORBextractorLeft, mpORBextractorRight;
    // Number of KeyPoints.
    int N;
    std::vector<cv::KeyPoint> mvKeysMain, mvKeysRight;
    std::vector<cv::KeyPoint> mvKeysUn;
    // Corresponding stereo coordinate and depth for each keypoint.
    // "Monocular" keypoints have a negative value.
    std::vector<float> mvuRight;
    std::vector<float> mvDepth;
    // Scale pyramid info.
    int mnScaleLevels;
    float mfScaleFactor;
    float mfLogScaleFactor;
    std::vector<float> mvScaleFactors;
    std::vector<float> mvInvScaleFactors;
    std::vector<float> mvLevelSigma2;
    std::vector<float> mvInvLevelSigma2;
```

```
      // Keypoints are assigned to cells in a grid to reduce matching complexity when
projecting MapPoints.
     virtual auto    getGrid() const -> const Grid* = 0;
     virtual auto     getFeaturesInArea(const float &x, const float  &y, const float  &r,
const int minLevel = -1, const int maxLevel = -1) const -> std::vector<size_t> = 0;

};
```

Clase que contiene tanto información 3D del frame como información acerca de la pose y los puntos 3D. Esta clase también es importante en el sentido de que en muchas funciones, lo que en realidad se necesita es saber la información geométrica de un frame, y no la información extra que la librería almacena en un frame.

```
class Frame3D
{
    // Calibration matrix and OpenCV distortion parameters.
    ccore::ptr<Camera>  mCamera;
    // MapPoints associated to keypoints, NULL pointer if no association.
    std::vector<ccore::ptr<MapPoint, false>> mvpMapPoints;
    // Flag to identify outlier associations.
    std::vector<bool> mvbOutlier;
    // Image features
    ccore::ptr<const ImageFeatures> mImageFeatures;
    // Set the camera pose.
    virtual void    SetPose(cv::Mat Tcw) = 0;
    // Returns the camera center.
    virtual auto    GetCameraCenter() const -> const cv::Mat = 0;
    // Returns inverse of rotation
    virtual auto    GetRotationInverse() const -> const cv::Mat = 0;
    virtual auto    getPose() const -> const cv::Mat = 0;
    virtual auto    GetRotation() const -> const cv::Mat = 0;
    virtual auto    getPosePosition() const -> const cv::Mat = 0;
    virtual void    scale(double) = 0;
    virtual void    transform(const cv::Mat) = 0;
      // Keypoints are assigned to cells in a grid to reduce matching complexity when
projecting MapPoints.
    inline  auto    getGrid() const -> const Grid* {return mImageFeatures->getGrid();}
};
```

En realidad, se podía haber fragmentado las clases aún más, pero el tiempo invertido ha hecho que se debiera parar en donde se pudo parar.

Tener las clases bien cohesionadas es muy importante por la razón siguiente: Imaginemos, por seguir con el ejemplo anterior que a una función llamada miFuncion se le pasa un objeto de la clase Frame para que calcule alguna cosa. Aunque el nombre de miFuncion es evidentemente irreal y puesta a modo de ejemplo, en algunas ocasiones me he encontrado con el mismo problema. Bien, ¿que hace la función llamada miFuncion? ¿Trabaja con los puntos 3D? ¿Toca algo acerca de la pose del frame? ¿necesita los puntos característicos? …

Pero si por el contrario, a la función llamada miFuncion, en vez de pasarle un Frame, le paso un objeto de la clase ImageFeatures, ya se puede acotar mucho más lo que puede estar haciendo o no una función.

Es muy importante que una función tome como parámetros el mínimo indispensable para funcionar. Si el sistema está poco cohesionado, nunca se sabrá a ciencia cierta qué es lo que hace una función por debajo, aunque evidentemente, el nombre de la función nos dé muchas pistas.

Por poner otro ejemplo en concreto acerca de este tema, vamos a ver otra función de la librería. En la clase KeyFrameDatabase nos encontramos la siguiente función:

```
std::vector<KeyFrame*> DetectRelocalizationCandidates(Frame* F);
```

Lo que hace esta función, gracias al nombre que le pusieron, es evidente, pero, conteste el lector a las siguientes preguntas: ¿Qué información del Frame está usando esta función? ¿Lo modifica o no?

En vez de la definición anterior, yo lo cambié a la siguiente definición:

```
virtual auto detectRelocalizationCandidates(const DBoW2::BowVector& BOW) const ->
std::vector<const KeyFrame*> = 0;
```

Como se puede apreciar, esta definición acota mucho más los datos que necesita esta función para poder funcionar. Ahora sabemos que para poder funcionar, esta función no necesita por ejemplo el id del frame, ni sus puntos 3D, …

5.5.3. Datos estáticos

Usar datos estáticos es de manera general una mala decisión (al igual que los singletons), aunque he de reconocer que en algún caso, te resuelven un problema en un momento. En otros casos es casi inevitable, pero son unos casos muy reducidos y muy concretos.

En la librería, además hacen imposible ciertas cosas que deberían, por naturaleza de la misma, ser posibles. En otra sección se defiende y se demuestra que es posible utilizar más de un Tracker para llevar de manera simultánea el tracking de dos sistemas en paralelo.

En la clase Frame original existen las siguientes variables:

```
    static float fx;
    static float fy;
    static float cx;
    static float cy;
    static float invfx;
    static float invfy;
```

El hecho de que dichas variables sean estáticas obliga a que en el tracking solo pueda usar solo un tipo de cámara, es más, no solo un tipo, sino una cámara en concreto con sus coeficientes de distorsión característicos.

En este caso concreto esto se soluciona, y además con un sistema mucho mejor creando una clase llamada Camera con la siguiente información:

```
class Camera {
    enum SensorType {MONOCULAR = 0, STEREO = 1, RGBD = 2};

    virtual auto    getSensorType() const -> SensorType = 0;
    virtual int     getWidthPixelCount() const = 0;
    virtual int     getHeightPixelCount() const = 0;
    virtual float   getFX() const = 0;
    virtual float   getFY() const = 0;
    virtual float   getCX() const = 0;
    virtual float   getCY() const = 0;
    virtual float   getInvFX() const = 0;
    virtual float   getInvFY() const = 0;
    virtual float   getK1() const = 0;
    virtual float   getK2() const = 0;
    virtual float   getK3() const = 0;
    virtual float   getP1() const = 0;
    virtual float   getP2() const = 0;
    virtual float   getStereoBaseLineMultiplicatedByFX() const = 0;
    virtual float   getStereoBaseLine() const = 0;
    virtual float   getDepthThreshold() const = 0;
    virtual float   getDepthMapFactor() const = 0;.
    virtual float   getFPS() const = 0;
    virtual auto    getK() const -> const cv::Mat = 0;
    virtual auto    getDistCoef() const -> const cv::Mat = 0;
   virtual void   undistortPoints(cv::InputArray distorted, cv::OutputArray undistorted)
const = 0;
};
```

A partir de la especificación de la clase cámara, en cualquier momento se tiene una información mucho más completa acerca de la cámara con la que se ha creado un frame, o un KeyFrame, ya que cada frame puede almacenar un puntero a la cámara con la que se generó.

5.5.4. Duplicado de información

En la librería existe duplicado de información en varios sitios. Por poner un caso en concreto, la clase KeyFrame y Frame comparten muchos datos. Lo que en esta sección se expone viene dado precisamente por el punto anterior de la cohesión.

Si las clases se hubiesen cohesionado con más profundidad, en vez de duplicar datos, seguramente las clases compartirían referencias a módulos que realizasen la tarea que tienen en común.

Como ya se ha comentado en este ejemplo, la clase Frame y la clase KeyFrame, contiene información casi idéntica (excepto por un array) acerca de los puntos característicos que fueron usados para generarse. En apartados anteriores, al haber abstraído dicho concepto en

una clase concreta, la clase ImageFeatures, esos datos ya no se duplican más. La clase Frame y la clase KeyFrame tienen ambos un puntero que apuntan al mismo objeto ImageFeatures.

5.5.5. Duplicado de código

Si en la sección anterior se ponía de manifiesto la existencia de duplicados de información en varias clases, existe otro tipo de duplicado que es, a la par de innecesario, más peligroso: el duplicado de código.

En diferentes secciones de la librería se duplica código casi de manera gratuita, y en algunas ocasiones, el código duplicado es de más de 50 líneas de código (con la salvedad a alguna línea). Da la sensación que por comodidad, como se ha necesitado la misma funcionalidad en sitios diferentes, al autor en cuestión le ha resultado más sencillo copiar y pegar código de un sitio a otro, en vez de intentar rediseñar el sistema para que no existiese la necesidad de duplicar el código.

La explicación de porqué la duplicación de código no es mala, sino peligrosa, la encontramos a continuación:

- Imaginemos que hemos implementado una función dentro de una clase que realiza una tarea específica.
- Como esa misma funcionalidad me hace falta en otra parte, en vez de rediseñar el sistema, lo que hago es copiar y pegar la función en la nueva clase.
- Presumamos también, y creo que no erro demasiado, que en esta librería que nos hemos inventado, el tiempo necesario para su implementación fue de más de 6 meses a jornada completa.
- Al cabo de los 6 meses, me doy cuenta de que dicha función tenía un fallo que arreglo. Por experiencia propia, puedo casi asegurar que en el momento de haber encontrado el fallo ya no me acuerdo ni remotamente que esa función fue copiada y pegada en otro sitio. Tengo por lo tanto dos funciones, una que tiene el problema arreglado y otra que no lo tiene. No digamos ya si son varios los desarrolladores de la librería.

Nunca hay que copiar y pegar código precisamente por lo expuesto, en el caso de encontrar un fallo, al solucionarlo, el fallo se debe arreglar de forma global en todo el sistema, y no de forma local.

Además de lo expuesto, cuando un desarrollador se encuentra código duplicado, es un poco desconcertante, puesto que cree que hay algo muy diferente que separa las dos versiones del copy/paste y no se da cuenta de qué es.

5.6. Cambios propuestos

Al margen de la sección anterior, en este punto se sugerirá algún otro cambio menor que podrían hacer más atractiva la librería a la par que reducir el tiempo de cómputo del SLAM. Aquí encontraremos, por lo tanto, sobretodo mejoras de gestión y costes temporales.

5.6.1. Mutex

Verdaderamente se puede decir que la intención del equipo de Tardós tiene una idea clara de lo que quiere hacer. Eso se demuestra en el hecho de que se atreve a usar multithread en su sistema.

Hacer que un sistema funcione con multithreading es algo que es bastante pro a nivel de arquitectura de software. El multithreading puede ofrecer una ventaja clara con el paralelismo, pero tiene una necesidad inherente, que es la protección de los datos.

No recuerdo haber visto ningún fallo en la utilización de los mutex, pero si bien es cierto eso, también es cierto que en algunos lugares del código, en vez de utilizar un mutex, yo hubiese propuesto el uso de un spinlock.

Para protecciones efímeras, se suele aconsejar el uso de spinlocks ya que es del orden de 50 veces más rápido que el mutex estándar.

5.6.2. Memoria Dinámica

Otro aspecto importante que destacar, es que el código está lleno de copias de memoria de un sitio para otro. Seguramente (por no decir seguro) esto se debe al uso del multithreading. Sobre todo, en un frame, se crea y se destruye mucha memoria en llamadas a funciones que devuelven arrays y mapas de objetos.

La memoria dinámica no es para nada gratuita. Cuando se le pide al sistema un nuevo bloque de memoria puede incurrir en una llamada al sistema operativo, que no son precisamente muy rápidas. Ciertamente el runtime de c debería acelerar dicho proceso, pero aun así, son demasiadas peticiones de memoria, con lo que seguro que hay una penalización por su uso.

Creo que rediseñando los sistemas con el traspaso de memoria en mente, se podrían haber ahorrado mucho uso de memoria dinámica. Una manera de reducir esto, aunque puede que ocupe un poco más de memoria, son los pools. Por ejemplo, si una función ha de devolver un array de objetos, en vez de crear y destruir ese objeto con una nueva llamada, se podría usar un pool donde se almacenen dichos arrays. Con esto, cuando la función quiere devolver un array le dice al pool: "Oye, *necesito un array, ¿Tienes alguno de sobra por ahí para usarlo temporalmente?, luego te lo devuelvo*"

5.6.3. std::shared_ptr y ccore::ptr

Desde antes de 2011 existe en el framework estándar de C++ los "smart pointers". Desde su aparición es desaconsejable usar memoria dinámica de manera explícita en lo referente a objetos, ya que, por descuidos de programación, podemos encontrarnos con memory leaks a la par que hemos de añadir código extra para la gestión de la destrucción de objetos.

Con los std::shared_ptr, el paradigma de la creación y destrucción de objetos cambia radicalmente a mejor. Antes, las cosas se construían y se destruían, ahora, los objetos se crean, se usan y se dejan de usar. Es precisamente el uso de los smart pointers que hacen posible despreocuparse de cuando se destruye un objeto, es más, es este tipo de característica lo que hace que lenguajes de alto nivel como Java y C# sean más sencillos de usar que C++.

La idea en la que se basan los sistemas de "smart pointers" es el siguiente:

- Cada objeto tiene un contador de referencias indicando cuántos punteros apuntan a él.
- Cuando, por el recorrido del programa, un puntero de algún sitio apunta a un objeto (lo referencia), incrementa el contador de referencias del objeto en cuestión.
- Cuando, por el contrario, un puntero deja de apuntar a un objeto, se decrementa su contador de referencias.
- Si el contador de referencias de un objeto llega a ser cero, el objeto es destruido.

La clase std::shared_ptr pertenece al estándar de C++, mientras que la clase ccore::ptr es de cosecha propia. Por defecto, si hay que tomar la decisión de qué sistema usar, hay que elegir std::shared_ptr. Por ejemplo, si la librería usase los "smart pointers", esta sección no existiría. Por otro lado, en este trabajo se usó ccore::ptr, porque me ofrecía un control de destrucción de objetos que el std::shared_ptr no ofrece.

La diferencia entre las dos opciones de smart pointers radica en que std::shared_ptr lleva él mismo la gestión de conteos de referencia, mientras que ccore::ptr delega dicho comportamiento al objeto de la clase que usan.

5.6.4. Uso de const

Es, a mi juicio, muy importante etiquetar de const aquellas variables que no se vayan a modificar, y aquellos métodos que no modifican a los objetos. Esto otorga cierta auto-protección al programador a la par que sabemos cuando un objeto va a ser modificado o no por la llamada a una función o a un método.

Ciertamente, en la librería hay variables que están protegidas con const, pero hay muchísimas otras que no lo están y que debieran estarlo.

Aún más, no existe en la librería casi ningún método catalogado de const, cuando en esencia, todos los getters, por ejemplo, deberían estar catalogados con esa protección.

5.6.5. Uso de auto

Aunque es cierto que la librería ya tiene algunos años, en 2011-2012 apareció en C++ la capacidad de utilizar auto como declarador de variables. No voy a defender el uso de auto cuando se declaran variables temporales en una función, pero sí que lo haré y de manera muy enérgica en los bucles.

Por poner un ejemplo de qué tipo de código es más legible, miremos el siguiente ejemplo encontrado en la librería:

```
for(vector<MapPoint*>::iterator vit=vpMP.begin(), vend=vpMP.end(); vit!=vend; vit++)
{
    MapPoint* pMP = *vit;
    ...
}
```

Hasta hace unos años los programadores se veían casi obligados a usar ese patrón de bucles, pero creo que con la aparición de auto, el bucle quedaría más legible de la siguiente manera:

```
for(auto pMP : vpMP)
{
    ...
}
```

Es cierto que más que un requisito de eficiencia, tiene más que ver con la claridad de código, pero también es cierto, que cuanto más claro sea un código, más sencillo de mantener es, y más sencillo lo tiene un nuevo desarrollador a la hora de hacer modificaciones.

5.6.6. Guia de estilo no homogénea

Da la sensación de que los autores se han puesto poco de acuerdo en qué guia de estilo usar, cada uno ha programado por su cuenta, aunque he de decir que parecida. Esto, al margen de generar una sensación un tanto “mala”, no tiene nada de malo en sí, por supuesto, pero lo que sí que tiene algo de poco amigable es el nombre que se han usado en algunas variables. Por poner un ejemplo:

```
class KeyFrame
{
   ...
   long unsigned int mnId;
   long unsigned int mnFuseTargetForKF;
   long unsigned int mnBALocalForKF;
   long unsigned int mnBAFixedForKF;
   long unsigned int mnLoopQuery;
   int mnLoopWords;
```

```
    float mLoopScore;
    long unsigned int mnRelocQuery;
    int mnRelocWords;
    float mRelocScore;
    const float fx, fy, cx, cy, invfx, invfy, mbf, mb, mThDepth;
    // Number of KeyPoints
    const int N;
    ...
};
```

¿Alguien podría decir qué es:

- mnLoopQuery?
- mbf?
- mb?
- mThDepth?
- N? Claro, con respecto a N, aquí viene bien claro lo que es gracias a un comentario, parece no haber problema, pero si estamos analizando lo que hace una función y vemos "myFrame->N", pues no tenemos ni idea de lo que significa.

El código está repleto de zonas que no se entienden a la primera de cambio, se necesita un estudio completo de la clase para saber su semántica.

5.6.7. Tiempos

En realidad, los autores no son los responsables de lo que en esta sección se comenta, ya que como buenos ingenieros tienen todos los procesos mentales al 100% para resolver un problema de la mejor manera posible.

Existen varios problemas con los tiempos en la librería: tiempo de compilación y tiempo de carga.

Las librerías externas que se usan tienen un alto uso de templates, y a casi ciencia cierta ellas tienen mucha responsabilidad en cuanto al tiempo que se toma la librería para compilar. Para reducir esto, tanto como se pueda, hay que incluir las cabeceras en los archivos con extensión "cc" y no en los "h".

La librería se alimenta de un "Bag of Words" (BoW) para poder hacer el matching entre frames y puntos, la carga del BoW es extremadamente lenta. Cierto es que en principio existe alguna versión del BoW en binario y no en modo texto que debería cargar más rápido, pero no pude dar con ella. Esto probablemente es fallo mío.

Con todo lo expuesto, por poner un ejemplo, digamos que hago una modificación en un fichero .h, y para ver que no he metido la pata, ejecuto el programa, digamos unos frames para ver que la cosa sigue funcionando correctamente. Concretamente, en el test de

"rgbd_dataset_freiburg3_long_office_household", esperamos hasta que la cámara haya dado la vuelta. Pues bien, desde que termino de escribir en el .h (vamos a poner un simple espacio para que no dé problemas de compilación), hasta que la cámara llega a ¼ de su recorrido en el test pasan 3 minutos y 40 segundos en modo DEBUG. Debido a ello, cada mínimo cambio tarda muchísimo tiempo en ser testeado. Si bien es cierto que este tiempo baja si lo ponemos en modo RELEASE a 2 minutos, pero este enfoque no es correcto puesto que cuando estás programando algún cambio, si el sistema peta, necesitas toda la información al respecto de ese fallo. Es por ese motivo que en cambios, se programa en modo DEBUG.

Se ha de insistir en que los autores no tienen culpa ni de lejos de este hecho, pero es un hecho. Como cada ladrillo hace pared, quizás si no hubiese tanto traspaso de memoria, tanto mutex activo, tanto template, y los .h tuviesen lo mínimo para poder funcionar, este tiempo se podría haber reducido.

5.6.8. Variables de clase y de algoritmo

Un factor de lo que pecan algunas clases es utilizar variables de algoritmo en ellas. Una variable de objeto, es una variable que sirve para definir un objeto, mientras que una variable de algoritmo es una variable temporal que usa un algoritmo la cual, cuando el algoritmo termina, es destruida o borrada.

Veamos un ejemplo en concreto:

```
class KeyFrame
{
    ...
    // Variables used by the KeyFrame database
    long unsigned int mnLoopQuery;
    int mnLoopWords;
    float mLoopScore;
    long unsigned int mnRelocQuery;
    int mnRelocWords;
    float mRelocScore;
};
```

Como bien dice en el comentario, estas variables se usan en la base de datos de KeyFrames. Si vamos a ver cómo utiliza la base de datos de KeyFrames dichas variables, nos damos cuenta de que son todas temporales, y que el algoritmo se puede diseñar para que dichas variables no residan en el KeyFrame.

Por ello, y por muchas otras cuestiones citadas anteriormente, da la sensación de que han tenido cierta prisa en acabar la librería, como si no llegasen a una death-line que se hubieran impuesto.

5.6.9. Visibilidad entre KeyFrames

Con el paso de los años, me he dado cuenta que muchas veces he sido víctima de mi propio código. Por lo tanto, incluso más que el hecho de que un sistema sea óptimo de manera local,

priorizo mucho la legibilidad del código fuente, e intento respetar las normas de acoplamiento y cohesión.

En la implementación de la librería, un KeyFrame guarda información acerca de qué KeyFrames son visibles desde él. Esto se debe a que dicha información es muy relevante para el proceso de SLAM.

Por mi parte, como considero que esto es un problema que no debe marcar cómo se define un frame, a la par que, las referencias entre hermanos son difíciles de manejar, yo hubiera creado una clase separada con tal propósito.

Es decir, podría existir una clase aislada cuya función sea gestionar la visibilidad entre KeyFrames. De esta manera, cuando un frame es creado, se registra en este sistema, y cuando es borrado se desregistra.

Con este punto de enfoque, lo que hago es aumentar la cohesión en el sentido de que a partir de este momento, existe una clase con un único cometido, el de llevar a cabo la gestión de la visibilidad entre frames. De esta manera, si veo que en algún lugar del código existen referencias a esta clase, sé que se necesita la información de la visibilidad de los frames, y si no la hay, ya sé que dicha información no es necesaria.

Este mismo concepto lo aplicaría al par KeyFrame y MapPoint, pues ocurre algo parecido con ellos.

5.7. Añadiendo funcionalidades a ORB-SLAM

A continuación, se expondrá algunos puntos que se han conseguido implementa los cuales, a mi humilde opinión, le pueden dar a la librería cierta funcionalidad interesante.

5.7.1. Multitracking

El primer reto era conseguir que ORB-SLAM admitiese más de un Tracker. Para poder conseguirlo, lo primero que había que hacer era, como se ha comentado en secciones anteriores, eliminar todas las variables static que se pudiesen. Concretamente se creó la clase Camera y la clase Factory con el fin de eliminar variables static.

La primera que entró en juego fue la clase Camera, la segunda fue mucho más difícil de detectar y se implemento en las últimas etapas de estos cambios.

La clase Camera tiene su importancia porque en la implementación por defecto de la librería, los KeyFrames tienen información de la cámara pero, en vez de ser variables normales, son de tipo static. Ello conlleva necesariamente en que en todo el sistema pueda haber una y solo una cámara.

Eso sumado a que en varios sitios se requiere la información de la cámara, y además esta información no varía en toda la vida de la cámara, hizo necesario crear una clase con información pertinente de la cámara. Por ejemplo, las constantes de distorsión, su tamaño, …

La segunda clase fue bastante más difícil de ver, como ya se ha comentado. Haciendo pruebas con el código, mirando cosas y moviendolas de sitio, observé una variable llamada "mnId" en el KeyFrame. Tal y como su nombre indica, parecía y de hecho lo puede ser, un identificador del frame. Al principio pensé ¿Para que querrán un identificador del frame? Será para hacer alguna especie de hash o diccionario. Por lo tanto no le di mayor importancia. Además observé algo bastante estándar en la generación de claves principales, y es que el identificador cogia su valor de una variable estática, la cual aumentaba su valor por cada frame creado.

Cuando se preparó el sistema para funcionar con multitracking, al ponerlo en funcionamiento, no marchaba todo lo bien que se podía esperar. Es más, se comportaba de una manera muy poco determinista. Llevó un tiempo ver el por qué, y es que el id del KeyFrame, lo utilizan las clases principales de tracking para hacer sus tareas a modo de tiempo de vida. Es decir, el id es de un KeyFrame conlleva de manera implícita el momento en que se creó.

Esto se solucionaba, de entre las muchas posibles soluciones que hay, con crear una clase factoría para cada elemento que llevara consigo un generador de Id's propio. Tanto la clase frame, la clase KeyFrame y la clase MapPoint, llevan un identificador, por lo tanto, para cada una de esas tres clases implementé una factoría. Dicha factoría lleva dentro de sí lo que antes era una variable estática para generar Id's incrementales.

De esta manera, cada Tracker tenía su propio generador de ids. En realidad lleva 3 generadores de Id's: Frame, KeyFrame y MapPoint.

Por temor a que no llegara a funcionar del todo bien, se implementó un código que a mi juicio faltaba por hacer: eliminar variables de algoritmo. En una de las secciones anteriores, se ha explicado qué son las variables de algoritmo. El KeyFrame tiene varias de estas variables, por lo tanto se repasaron estas variables para ver en qué sitio eran utilizadas. Sí que se observó algunas, que también están relacionadas con la relocalización. Por lo tanto, se rediseñó levemente los algoritmos que utilizaban esas variables, y se hizo que desaparecieran de la clase.

5.7.2. Subtracking

Como se ha comentado en secciones anteriores, el sistema obedece a 3 estados: "iniciando", "trackeando" y "perdido". Es de suponer, que los autores han querido dejarlo finalizado así por algún motivo.

Por lo tanto, como característica de la librería se tiene que, cuando un Tracker se pierde, hasta que no vuelve a encontrarse entre los frames que ya tenía almacenados, no hace tracking

nunca más. Insisto en que creo que esto debieron decidirlo los autores así, por facilidad de uso, o por otros requerimientos.

En este trabajo se ha modificado la librería para que porporte un concepto al que se le llamó SubTracker. Cada Tracker puede contener a uno o más SubTrackers dentro de sí mismo. El comportamiento de los Trackers y los SubTrackers es el siguiente:

1. Al empezar, un Tracker crea a un SubTracker.
2. Inmediatamente, el SubTracker intenta inicializarse con información de la cámara. Es decir, intenta iniciar el SLAM.
3. Una vez iniciado el SLAM, el SubTracker es el que hace el tracking del sistema.
4. Cuando el SubTracker se pierde, intenta encontrarse durante unos cuantos frames.
5. Si pasados ese número de frames no se ha logrado encontrar, el Tracker crea otro SubTracker. Pero ojo, no detiene al primero.
6. Con estos 2 SubTrackers en marcha, uno intentando inicializarse y otro intentando encontrarse, pueden pasar dos cosas, que gane uno, o que gane el otro (bastante evidente).
    a. Si el primer SubTracker logra rencontrarse, el Tracker detiene al segundo SubTracker para que no continúe intentando inicializarse. De este modo, como el primer SubTracker se relocalizó, continúa su tracking de manera normal.
    b. Si, por el contrario, el segundo SubTracker logra inicializarse antes de que el primero se relocalice. Entonces el Tracker detiene al primer SubTracker que intentaba relocalizarse y continúa con el nuevo creado.
7. En este punto tenemos, por lo tanto, dos Trackers, uno detenido y otro en marcha. Imaginemos ahora que el Tracker que está en marcha se pierde. El sistema le da unos frames de cortesía para intentar rencontrarse, y si lo consigue, continúa con él. Si no lo consigue, entonces pone en marcha todos los SubTrackers que tiene latentes e intenta que alguno se relocalice. En este momento pueden pueden pasar dos cosas:
    a. Si pasados unos frames determinados ningún SubTracker logra relocalizarse, se crea otro nuevo y se vuelve al punto 5.
    b. Si algún SubTracker logra relocalizarse, continúa con él.

Esta forma de gestionar el tracking hace posible que el sistema esté siempre localizado, lo cual considero interesante. Si bien es cierto que se generan muchos mapas diferentes, en principio esto estaba ideado así para que el último punto de esta sección lo solucionase.

5.7.3. Servidor de imágenes

Para poder llevar a cabo el multitracking, era necesario habilitar que cada uno de los potenciales Trackers se alimentara de un servidor de imágenes propio, es decir, si tengo 2 Trackers, cada uno debe coger sus imágenes desde un sitio diferente.

Para ello se creó la clase ImageServer y se modificó la librería para que el Tracker se alimentara de ella. Además, fué diseñada a modo de interfaz por lo que pude implementar varios backends.

La interfaz de la librería responde a la siguiente implementación:

```
class ImageServer
{
    virtual auto    getCamera() -> Camera* = 0;
    virtual auto    getCamera() const -> const Camera* = 0;
    virtual cv::Mat acquireImage(double& ATime) = 0;
};
```

Como se puede observar la interfaz es muy sencilla. Tan sólo tiene la cámara que utiliza para recoger imágenes y una función que devuelve la imagen disponible en cada momento.

Al margen de esa interfaz se creó otra que hereda de la primera. El sentido de esta es que en la implementación de Android, la cámara tiene que pasarle las imágenes al servidor para que se las vaya dando al Tracker.

```
class ImageQueueServer : public ImageServer
{
    virtual void enqueueImageYUV(int width, int height, const void* YUVImage) = 0;
};
```

Por lo tanto de la clase ImageServer existen 4 backends:

- Un backend que es el estándar que coge imágenes de un repositorio de pruebas y las va sirviendo una a una. Este es el comportamiento por defecto de los test que se ofrecen en la librería.
- Un backend que coge las imágenes de un video y las va sirviendo fotograma por fotograma.
- Un backend que es un servidor UDP. Un dispositivo debidamente programado, puede conectarse y enviar imágenes para que el servidor haga el tracking. El funcionamiento es el siguiente:
    - El servidor espera una conexión.
    - Cuando le llega una imagen de una dirección en concreto crea un thread y espera a que le lleguen objetos llamados DAO que se explicarán a continuación.
    - Cada vez que le llega un DAO, lo abre, recoge la imagen que hay dentro y se la pasa al servicio que hay registrado con la IP de origen del paquete.
- Un backend que espera imágenes externas en formato YUV y las guarda hasta que el Tracker se las pide.

Los objetos llamados DAO (Data Access Object) es un formato inventado por mi que trata de simular al XML o al JSON pero en formato binario. Al principio pensé en transferir imágenes

en codificación JPEG-BASE-64 contenidas en un JSON o en un XML, pero como quería que ocupase el mínimo indispensable, me cree una librería para abstraer el formato contenedor, y de camino soportar dicho formato binario.

5.7.4. Mezclar mapas

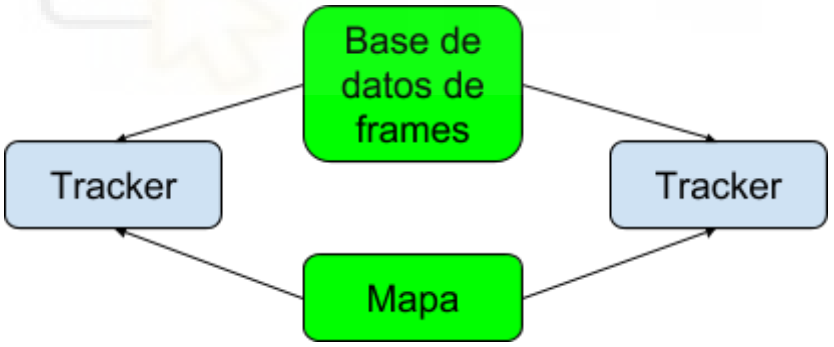
Este fue el punto fuerte desde el primer momento cuando cambió la línea de desarrollo. Antes de nada, hay que comentar que por temas de tiempo y de ciertos aspectos que se detallarán, no se pudo llegar a la implementación de la idea primera para lograr este objetivo. Después de muchos enfoques, se llegó a un resultado, como ya se verá.

5.7.4.1. Primer enfoque → Compartir la base de datos de KeyFrames

La primera intención de todas fue intentar idear el sistema para que n Trackers se pudieran alimentar de la base de datos de frames. El porqué necesitaban alimentarse de la misma base de datos base de datos se debe a que precisamente, cuando un Tracker está perdido, ataca a la clase KeyFrameDatabase para relocalizarse. Por lo tanto, sea como fuere, la intención era que todos los KeyFrames fueran almacenados en el mismo sitio.

Además, también deberían compartir la instancia del Mapa que es el lugar donde se almacenan puntos 3D y KeyFrames.

La cosa quedaría así:



Quizás el trabajo más costoso de todos fue preparar el sistema para que este diagrama fuera posible. Como se ha comentado en secciones anteriores, la librería no usa “smart pointers”, por lo tanto hay que llevar una gestión explícita de los objetos. Si hubiese seguido la línea de la librería, la propia gestión de las creaciones y destrucciones de objetos hubiese sido una odisea.

Por ello se optó por refactorizar toda la librería para poder usar “smart pointers” y que eso dejase de ser un problema. Apareció el dilema de qué sistema usar, si std::shared_ptr o ccore::ptr, pero al final me se optó por este último para que me fuese más fácil hacer ciertas cosas en un futuro, como por ejemplo, un garbage collector o una factoría.

También hay que decir, que el uso de "smart pointers" no es gratis, tienen cierto overhead asociado, pero si se compara con la cantidad de cómputo que se necesita para calcular el SLAM, es nimio.

Como ya se ha comentado, refactorizar la librería para soportar "smart pointers" fue una de las cosas más difíciles de todas, si no la que más. Ello se debe a que en muchísimos lugares del código se usan punteros (como es lógico) y contenedores de punteros (vectores, listas, conjuntos, ...). Cada vez que hacía un cambio en algún sitio, afloraban otros 10 donde debía hacer ese cambio.

Una vez conseguido esto, entre otras, las clases MapPoint, KeyFrame y Frame, pasaron a tener conteo de referencias y por lo tanto ya no hacía falta destruirlas explícitamente.

5.7.4.2. Resultados

Todos los esfuerzos por hacer que este enfoque funcionara no sirvieron de nada al final, o más bien de poco. Este fue el peor momento de todos. En principio el sistema debía funcionar correctamente, no encontraba lógica de su no-funcionamiento. Se empezó a investigar el motivo por el cual fallaba el nuevo sistema.

Resulta que la librería entera está basada en los identificadores de KeyFrames. Ellos, como ya se ha comentado, son, efectivamente, identificadores de KeyFrames, pero se usan como tiempo en el que se creó un frame. Fue en este desarrollo cuando me dí cuenta de esto. Se intentó rediseñar algoritmos, pasar los identificadores a variables de algoritmo, poner variables pseudo-locales, ... pero el sistema es tan dependiente de este punto que en el tiempo dedicado para esto, fue totalmente imposible reenfocar el sistema para que se dejara de usar los ID de los KeyFrames de esta manera. El primer y gran objetivo fracasó por motivos de diseño. Es posible que si se rediseñara la forma de hacer el "loop closing" y el "local map", esto se pudiera conseguir, pero un reenfoque de estos puntos puede suponer, por sí solo, un trabajo de varios meses.

De todas maneras, lo de los ID de los KeyFrames no fue el único aspecto que influyó en el fracaso de este enfoque, nacieron otras dificultades a la hora de hacer la base de datos única. Fue sobre todos los derivados a cómo gestionar el momento de fusionar los KeyFrames en el mismo mundo.

Debido a esto, se tuvo que seguir ideando otro sistema para si se podía implementar.

5.7.4.3. El tercer Tracker

En este momento un surgió una idea cuya base fue la que se utilizó en los intentos siguientes, y por tanto, como se puede leer entre líneas, este trabajo tampoco dio sus frutos.

La idea es la siguiente:

- Tenemos dos Trackers: A y B. Los dos con su vida y base de datos propia, y por supuesto, con su sistema de referencia local.

- Cada uno de ellos, A y B, va generando frames y si son frames localizables, crea nuevos KeyFrames para poder gestionarlos y almacenarlos.
- Se lanza un thread que va por separado. Este thread va cogiendo el último frame, llamémosle F, por ejemplo del Tracker A e intenta localizar ese frame dentro de la base de datos del Tracker B.
- Si se encuentra posible el punto anterior, es decir, hay un frame en A y en B
    - Buscamos la transformación T para pasar del frame B al frame A.
    - Generamos un tercer Tracker en el cual vamos incorporando los KeyFrames almacenados en A y después los de B como si fuesen frames nuevos. Los frames de B vendrían transformados por T.

5.7.4.3.1. Resultados

Este sistema no se llegó a implementar por varias razones:

1. Como el "local map" de cada Tracker borra KeyFrames redundantes, se temía por el hecho de que esto provocase que el tercer Tracker cayera en un estado de perdido. Si esto pasara, el tercer Tracker sería inválido.
2. La primera de las razones es que se estudió el tiempo necesario para ir generando este tercer Tracker y era demasiado alto. Debido a ello, si detenemos A y B para poder generar este tercer Tracker, se perderían muchos frames provenientes de la cámara, por lo que el resultado es que A y B se perderían.
3. En cambio, si se opta por ir acumulando los frames de A y B mientras estos están parados para generar el tercer frame, se puede generar demasiada memoria y se puede, por consiguiente, acabar con la memoria del sistema, por lo tanto, tampoco parece una idea muy buena.
4. Si se opta por no parar los Trackers A y B, este último Tracker puede no llegar a adelantarlos nunca (de hecho es muy posible), por lo que nunca se podría hacer la fusión de ellos, aunque se hubiese detectado que es posible.
5. Otra idea, pero esta puede llevar mucho más tiempo de implementación como para tenerla en cuenta en este TFM, es no ir acumulando frames, sino que por el contrario, meter todos los KeyFrames directamente, los de A sin transformar y los de B transformados por T. El tema es que el "loop closing" mantiene información de grupos de visibilidad para precisamente ejercer su cometido. Manejar esta información tiene cierta complejidad.

5.7.4.4. Traslación de mapa

Esta fue otra nueva idea derivada de las anteriores, pero lamentablemente, no funciona bien del todo (se comentará el porqué).

Como ya se ha comentado está basado en el apartado anterior, por lo que se copiará algunos puntos para que quede más claro. La idea es la siguiente:

- Tenemos dos Trackers: A y B. Los dos con su vida y base de datos propia, y por supuesto, con su sistema de referencia local.

- Cada uno de ellos, A y B, va generando frames y si son frames localizables, crea nuevos KeyFrames para poder gestionarlos y almacenarlos.
- Se lanza un thread que va por separado. Este thread va cogiendo el último frame, llamémosle F, por ejemplo del Tracker A e intenta localizar ese frame dentro de la base de datos del Tracker B.
- Si encuentra que es posible se hace lo siguiente:
    - Se paran los 2 Trackers.
    - Se obtiene la matriz de pose de F en la base de datos del Tracker A, llamémosla TA.
    - Se obtiene la matriz de pose de F en la base de datos del Tracker B, llamémosla TB.
    - El pensamiento es el siguiente, como en principio, el mismo frame se ha podido localizar en ambos sistemas, si consigo una matriz que es capaz de llevarse un sistema hacia el otro, transformo uno de los sistemas, y consigo que los dos estén en el mismo sistema de referencia.
    - Por consiguiente, lo que he de hacer es transformar todos los frames, KeyFrames y puntos 3D para poder pasar las coordenadas del sistema B a las coordenadas del sistema A.
    - A partir de este momento, ambos sistemas debería estár al menos durante un tiempo en el mismo espacio geométrico aunque no en RAM, esto hay que remarcarlo.

Para poder calcular la pose instantánea de un frame en un Tracker se utiliza la siguiente función:

```
bool   KeyFrameDatabase::detectRelocalization(Frame3D* AFrame, double
KeyMatchesThreshold, double SecondOportunityKeyMatchesThreshold);
```

Como se puede observar, para poder relocalizar a un frame dentro de una base de datos de KeyFrames, sólo se necesita la información geométrica del mismo. Antes dicha función ni siquiera se encontraba en el KeyFrameDatabase, se encontraba en el Tracker y además dependiente del trackeo actual. Al ponerla de esa manera, desde cualquier sitio y en cualquier momento puedo calcular si quiero detectar la relocalización de un frame.

Puede existir un problema para calcular la conversión del sistema A al sistema B o viceversa, y es la escala. Aunque los dos sistemas deberían tener la misma escala puesto que nacen de la odometría visual de la cámara que contiene los puntos reproyectados, no lo voy a dar por hecho y voy a ponerme en el peor de los casos, y es que las escalas no coinciden. Por lo tanto:

- Imaginemos que que el sistema A se ha creado con la escala A. Aunque esto sea así, las matrices de orientación de las poses calculadas son unitarias, por lo que en realidad dichas poses no tienen ni idea de con qué escala se ha creado.

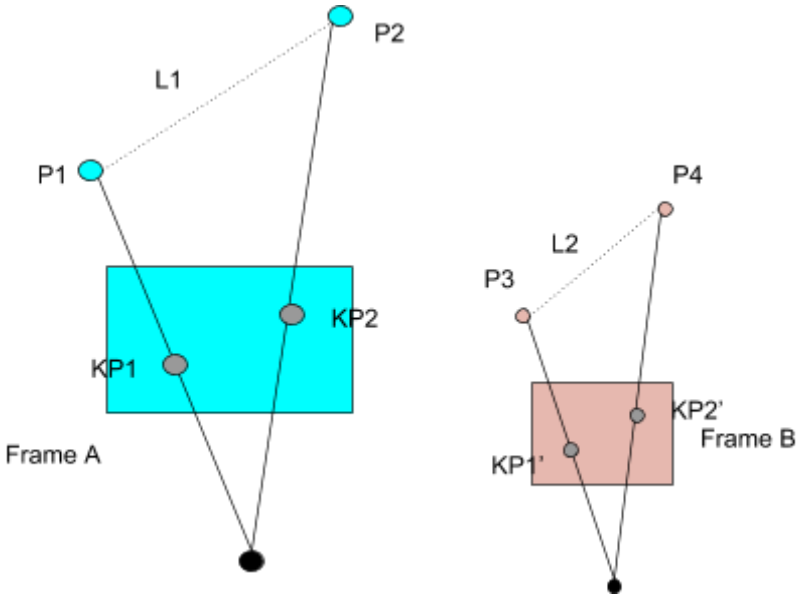
- Por otro lado, el sistema B se ha calculado con una escala de B y tiene la misma problemática que A.

Se ideó un experimento para ver si las escalas que debieran coincidir, realmente lo hacían o no. En el propio cálculo de la pose de un frame, también se calculan la posición 3D de los puntos característicos, por lo que cuando a un Tracker le pido la pose de un frame, también obtengo como salida la posición de los puntos en 3D relativos a los puntos característicos. Pero claro, no todos los puntos característicos se han podido triangular para poder conseguir su posición en 3D.

El experimento para el cálculo de la escala es el siguiente:

1. Como paso previo hemos de recordar que hemos descubierto con el proceso anterior que el frame F se puede localizar en los dos mapas.
2. Creo un frame FA perteneciente al sistema A con la información de puntos característicos de F.
3. Creo un frame FB perteneciente al sistema B con la información de puntos característicos de F.
4. Al relocalizar FA en el Tracker A, tengo su pose y la posición 3D de los puntos característicos, pero no de todos.
5. Al relocalizar FB en el Tracker B, tengo su pose y la posición 3D de los puntos característicos, pero no de todos.
6. Busco a ver qué puntos característicos han podido triangularse para generar puntos 3D en FA, y que también se generaron en FB.
7. Si encuentro 2 pares de puntos en FA y en FB que se triangularon correctamente ,en principio se suponen que son la misma recta, pero cada uno en su espacio.

En la imagen siguiente puede observarse lo que se intenta buscar:



En la imagen anterior vemos como el frame A ha podido mapear los puntos característicos KP1 y KP2 en su sistema de coordenadas, pudiendo generar la línea L1. El frame B ha podido hacer lo propio en su sistema de coordenadas formando la línea L2.

Si ambos sistemas estuviesen a la misma escala, y no hubiese error de triangulaciones, L1 y L2 deberían medir lo mismo. En principio, si no lo hacen el factor de escala se obtiene dividiendo la longitud de L1 entre la longitud de L2.

Una vez descubierto el factor de escala, en el proceso de transformar la información 3D del espacio B al espacio A, como paso previo hay que escalar todos los sistemas. Aunque en realidad existen 3 caminos posibles para aplicar las transformaciones:

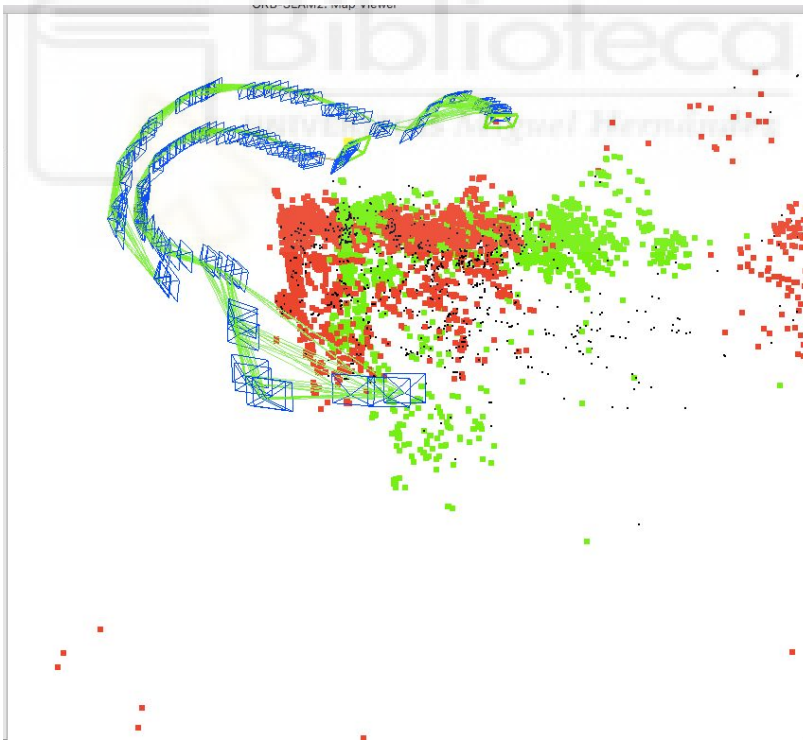
1. Método 1
    a. Se calcula la pose A.
    b. Se calcula la pose B.
    c. Se calcula la escala para pasar de B a A.
    d. Se aplica la escala a B, teniendo en cuenta que los vectores unitarios no se vean modificados por esta transformación.
    e. Se calcula la matriz T para pasar de B a A.
    f. Se aplica la transformación T a todos los elementos de B.
2. Método 2
    a. Se calcula la pose A.
    b. Se calcula la pose B.
    c. Se calcula la escala para pasar de B a A.
    d. Sea B' la inversa de B. A todo el sistema de B se le aplica la transformación de B'.
    e. Se escala B.
    f. Se multiplica todo el sistema B por la pose de A.
3. Método 3
    a. Se calcula la pose A.
    b. Se calcula la pose B.
    c. Se calcula la escala para pasar de B a A.
    d. Se calcula T como la transformación de pasar de B a A. Se multiplica T por la escala.
    e. Se transforma todo el sistema B por T
    f. Se normalizan los vectores unitarios.

De los tres métodos, el que se eligió fue el métodos número dos, ya que tiene menos cálculos y son más seguros. Como la información 3D de un sistema está muy dispersa (Frames, KeyFrames y MapPoints), en este caso resultó de mucha utilidad el tener una factoría de cada clase. En vez de pasar por los tres máximos responsables del SLAM (Tracker, LocalMap y MapClosing) para hacer las transformaciones oportunas y ver en qué sitio he de transformar la información geométrica, lo que se hace es lo que se explica a continuación.

Dado que existen 3 factorías (Frames, KeyFrames y MapPoints) que son las que crean los objetos que contienen información geométrica referente a cada uno de los Trackers y, dado que, cuando crean un objeto guardan una referencia a él para poder gestionar su destrucción, lo único que hay que hacer es recorrer todos los objetos que estén registrados en cada factoría y aplicar los cambios. De esta manera, de manera implícita, cada frame (por ejemplo) de un sistema, esté donde esté, si se aplica una transformación a la factoría entera, se quedará transformado.

Una vez programado esto, el sistema parecía tener bastante error al llevarse el espacio de B al espacio de A tal y como se puede ver en la imagen.

Repasando los datos una y otra vez, no encontraba el error. Al consultarlo com mi director de TFM, me dijo que cierto error podría ser normal. Continué haciendo pruebas y llegué a la conclusión de que al parecer la transformación funciona correctamente, pero por culpa del factor de escala, los dos Trackers no coinciden cuando se juntan. Si nos fijamos bien, las trayectorias coinciden en un punto determinado y conforme va pasando el tiempo, la escala del Tracker B es mayor que la del Tracker A. Ese punto donde coinciden es justamente el frame el cual se detectó como perteneciente a los dos sistemas.



Por lo tanto, se deduce que el escalado no funciona. Al hacer el escalado, lo más probable es que se quedara algún sitio sin actualizar. Se ha intentado buscar, pero por temas de tiempo, no se ha podido continuar la línea de desarrollo.

5.7.4.5. Sistema global

En tiempo de descuento ya, y después de haber frustrado las anteriores líneas de desarrollo, una idea me vino a la cabeza mientras conducía. Desde luego, esta debió haber sido la primera pues me parece tan sencilla que da risa. He de decir en mi defensa, si la hubiere, que cuando se está dentro de una idea que crees acertada, es muy difícil abandonarla a tiempo para acometer otros puntos de vista que no se ven desde el principio. Al menos eso es lo que a mi me suele pasar.

La idea es la que a continuación se relata. Supongamos que hemos encontrado, efectivamente, un frame que esté en A (un frame llamado FA) y en B (un frame llamado FB), y además, por el punto anterior, conocemos el cambio de escala. El tema se encuentra en buscar una transformación para hacer que el mundo de B pase al mundo de A, por lo tanto creamos una matriz que es la concatenación de estas tres matrices:

1. La matriz inversa de FB, comúnmente llamada pose de FB. Con esta transformación, todos los datos de B pasan a estar centrados con el frame F en el origen.
2. La matriz de escalado para escalar B hacia A.
3. La matriz directa de FA. Con esta transformación, todos los datos que están dentro del sistema de referencia de FB, pasan a estar bien colocados dentro del sistema de referencia A.

El kit de la cuestión está en no hacer nada más. En vez de intentar aplicar transformaciones en uno u otro lado, lo que se ha hecho es añadir un miembro más a la clase Tracker que es la matriz de transformación del mundo local del Tracker, al sistema global.

Claro, una vez hecho esto, cada Tracker es capaz de saber lo que hay en un entorno si lo desea, pues puede interrogar a otros Trackers pasando por su matriz global.

5.7.4.5.1. Resultados

Antes de comentar los resultados, este sistema tiene un pero. El tema pendiente es que los Trackers siguen sin estar conectados como pretendía la idea original, eso es cierto, por lo que si un Tracker se pierde, jamás se encontrará. Esto en principio se suple, aunque no de la manera deseada, con el subtracking.

Dicho esto, desde luego este sistema es el más sencillo de implementar, porque se basa en el cálculo de una sola matriz y ya. Además tiene una característica que lo hace totalmente inocuo a los Trackers.

Para poder calcular las poses del frame en común F en A y en B, se paraban los Trackers A y B, se les forzaba a ponerse como perdidos, se pasaba a modo de sólo localización y se esperaba a que el “map closing” y el “local map” terminasen sus tareas pendientes en curso. Pues bien, aunque es posible que genere un resultado con un poquitín más de error, todo ello no es necesario gracias a la siguiente función:

```
bool  detectRelocalization(Frame3D* AFrame, double KeyMatchesThreshold = 0.75,
double SecondOportunityKeyMatchesThreshold = 0.9);
```

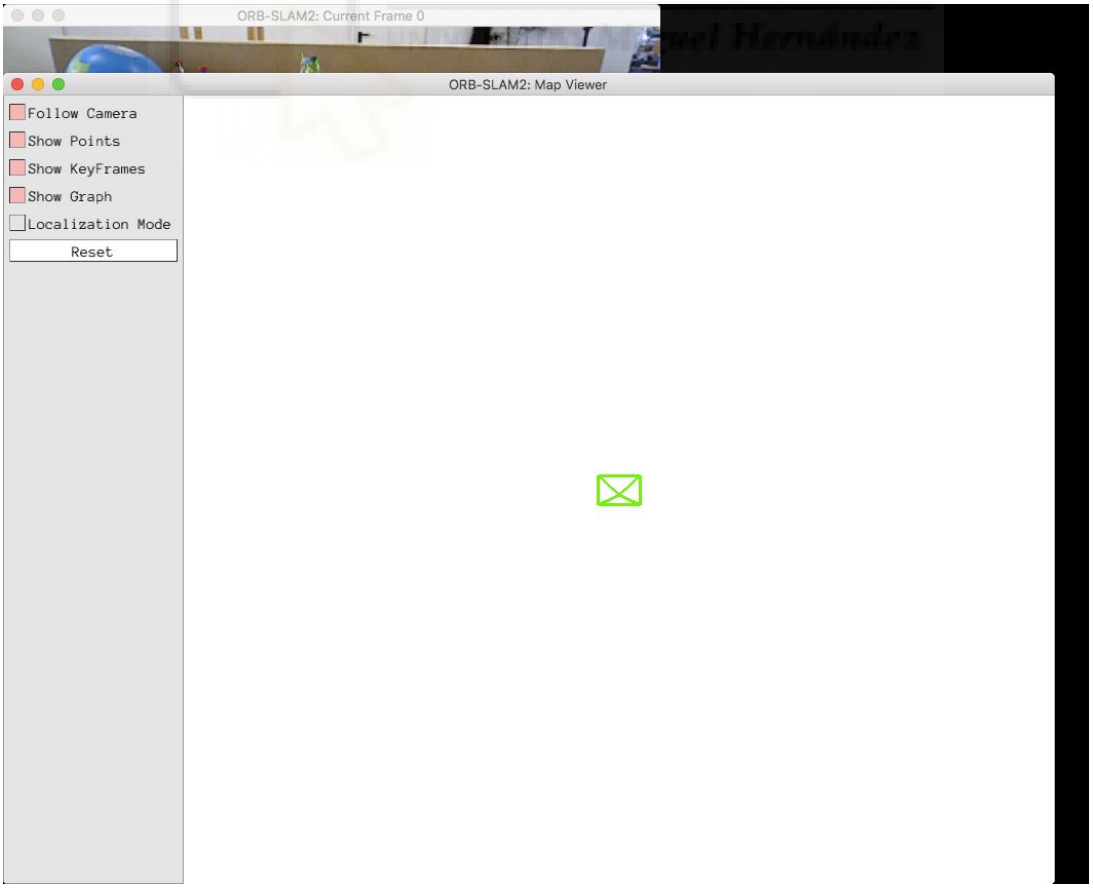
En esta función, el Frame3D que se le pasa a la función, una vez que la función devuelve true, contiene los puntos 3D triangulados que se han encontrado. Por lo tanto, la propia función nos dice si se ha detectado una "detección" y además configura el frame que le pasamos.

Es muy importante destacar que, viendo esa función, se sabe con total seguridad que esta función no lleva a cabo ninguna otra gestión como por ejemplo actualizar los datos de visibilidad de un KeyFrame, por lo tanto, es inocua.

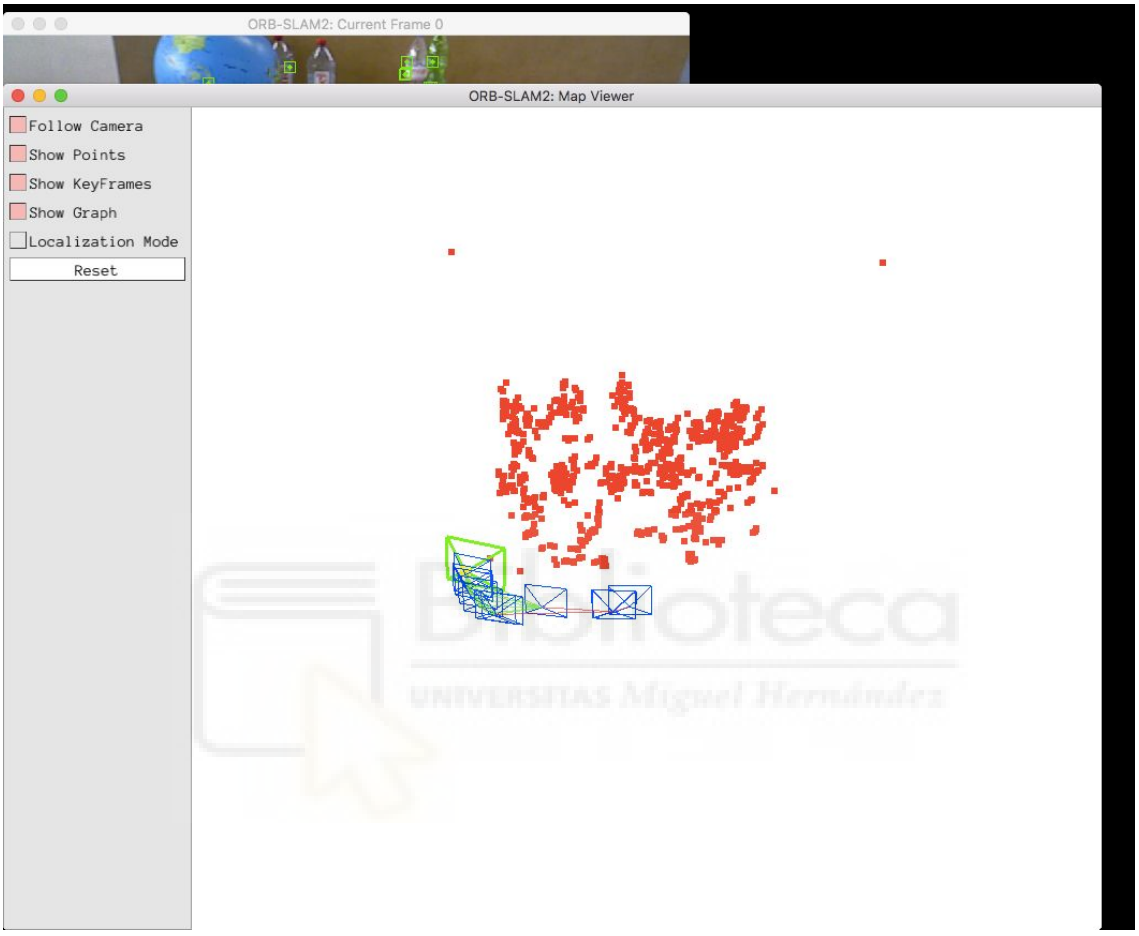
Como se puede ver en las imágenes anteriores, el resultado es el correcto, salvo por errores de imprecisión inherentes al todo el sistema de tracking. En la sección siguiente se comentarán ideas para mejorar este enfoque.

Antes de mostrar las imágenes, hay que decir que no siempre se obtienen los mismos resultados por falta de precisión, en algunas ocasiones el error era considerable (digamos un error de entorno al 5-10%) y no se ajustaba tan bien como se mostrará en las imágenes.

En esta primera imagen aparece el Tracker iniciándose, todavía no tiene nada localizado.

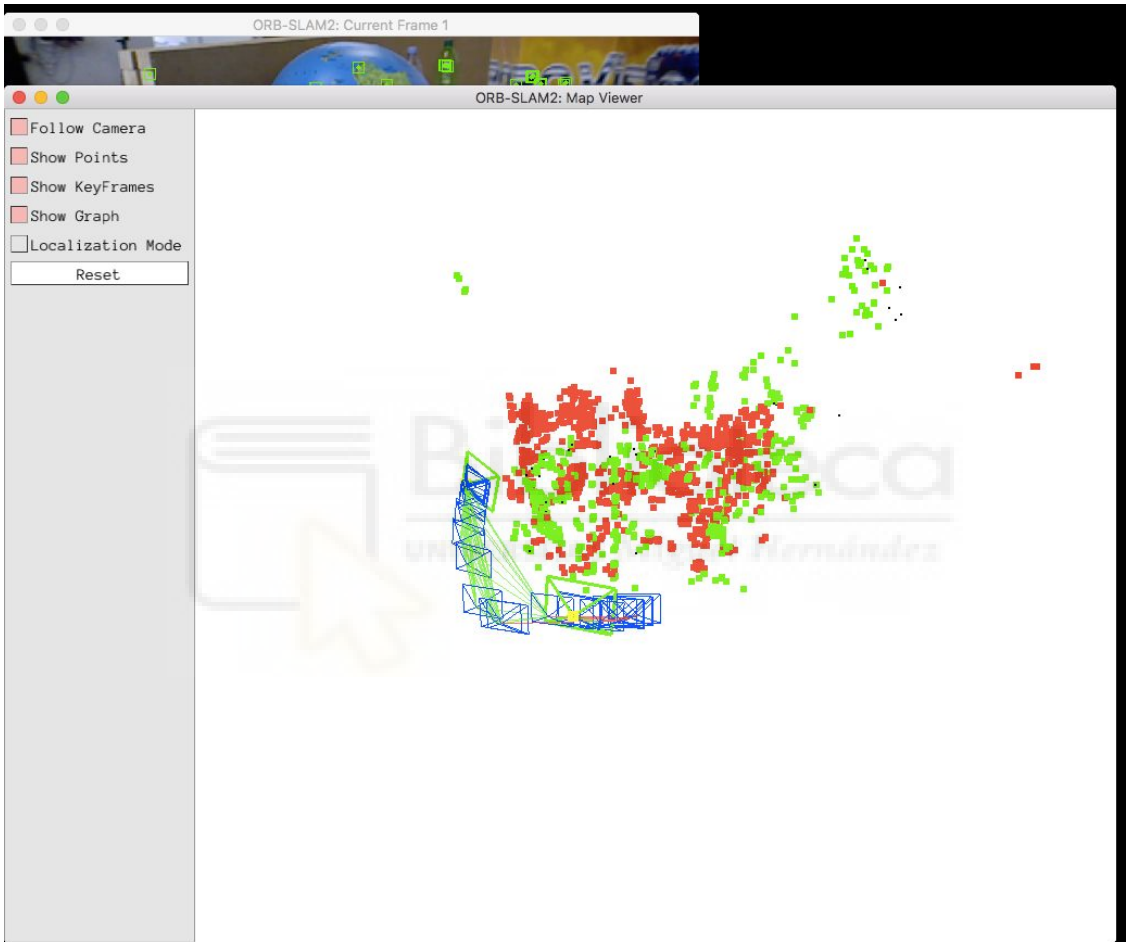


En esta otra el Tracker ya se ha encontrado y ha empezado a hacer el tracking. Como se puede observar hay una especie de vacíos de KeyFrames que han sido eliminados para optimizar el grafo.

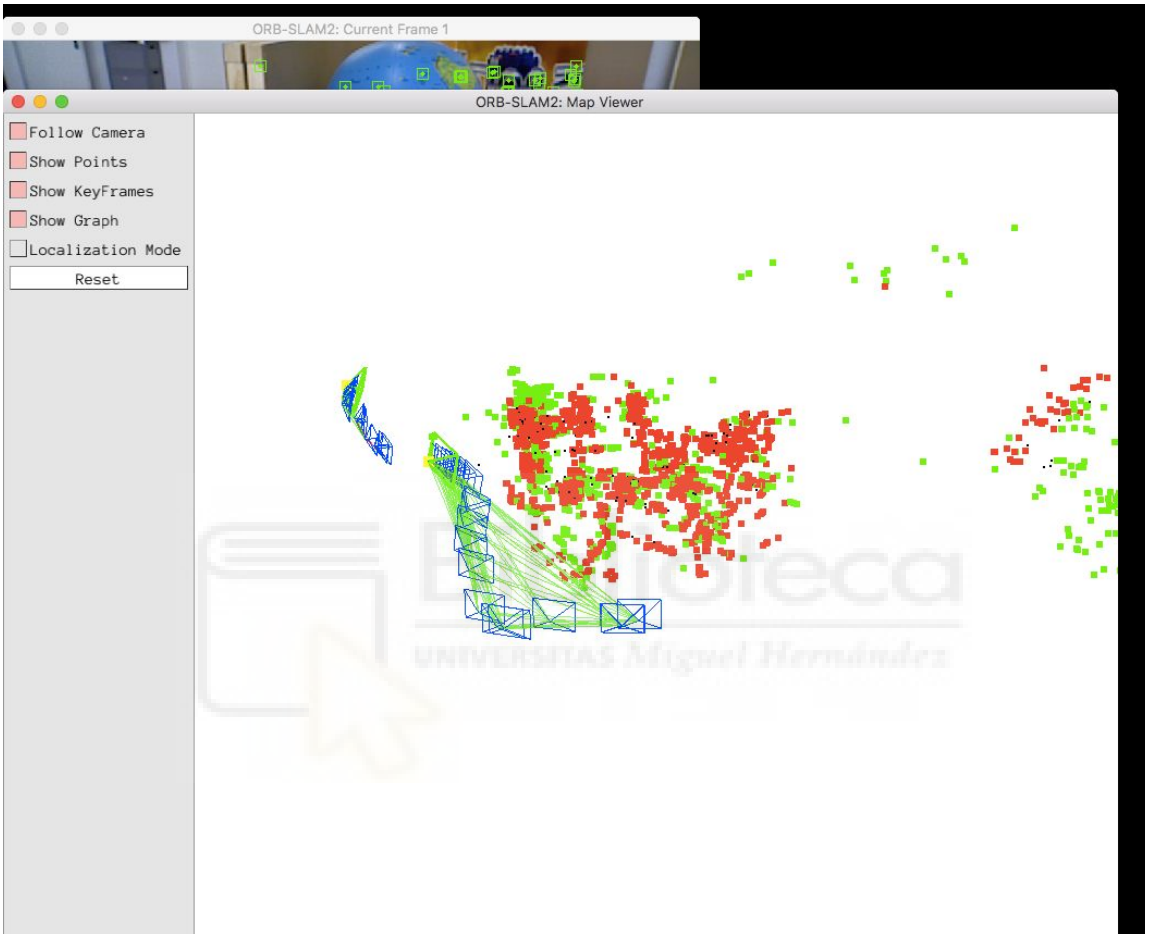


En esta otra imagen se puede observar lo siguiente:

- Ha empezado otro nuevo Tracker.
- Este nuevo Tracker pinta los puntos que encuentra en verde.
- Como es evidente, nada más comenzar, no se encuentra ningún Frame que pueda servir de punto de anclaje entre los dos Trackers.
- El segundo Tracker, al igual que el primero, comenzó desde el punto cero.

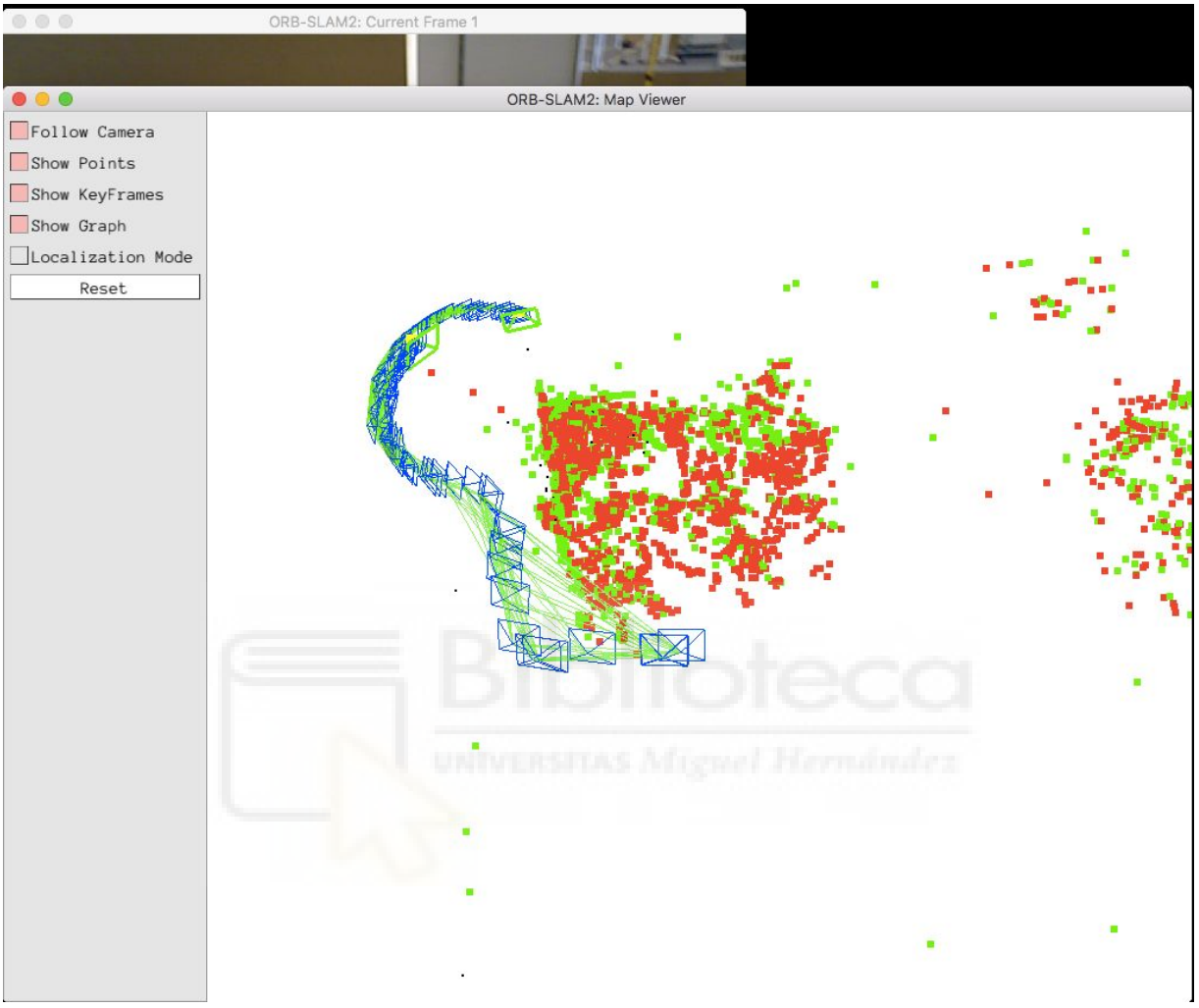


Llegado el momento, como se puede apreciar en la imagen, se encuentra un punto de anclaje entre los dos Trackers. En ese momento, el segundo Tracker modifica su matriz global que hasta ahora era la identidad para poder ponerse en el sistema de coordenadas del primer Tracker.

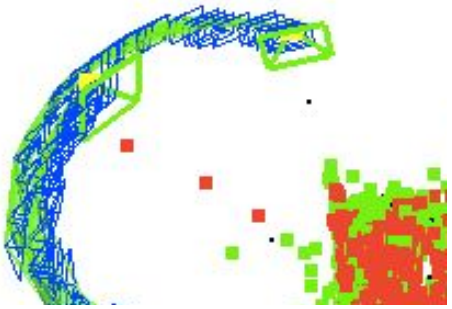


En las siguientes imágenes se observará cómo realmente hay una diferencia de escala.

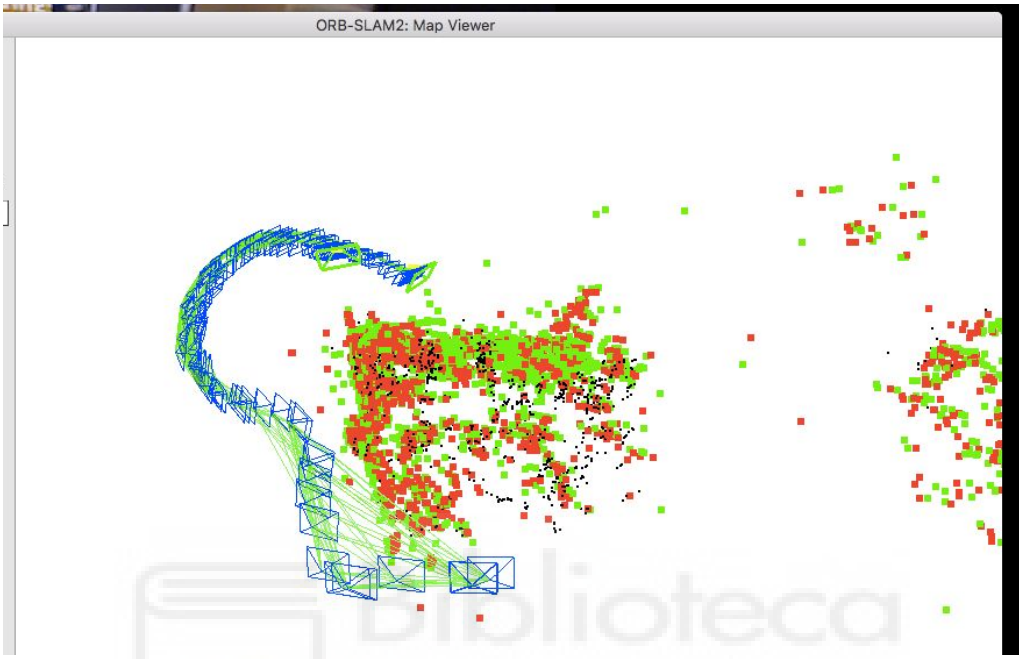
En esta imagen se aprecia cómo los dos Trackers siguen el mismo recorrido.

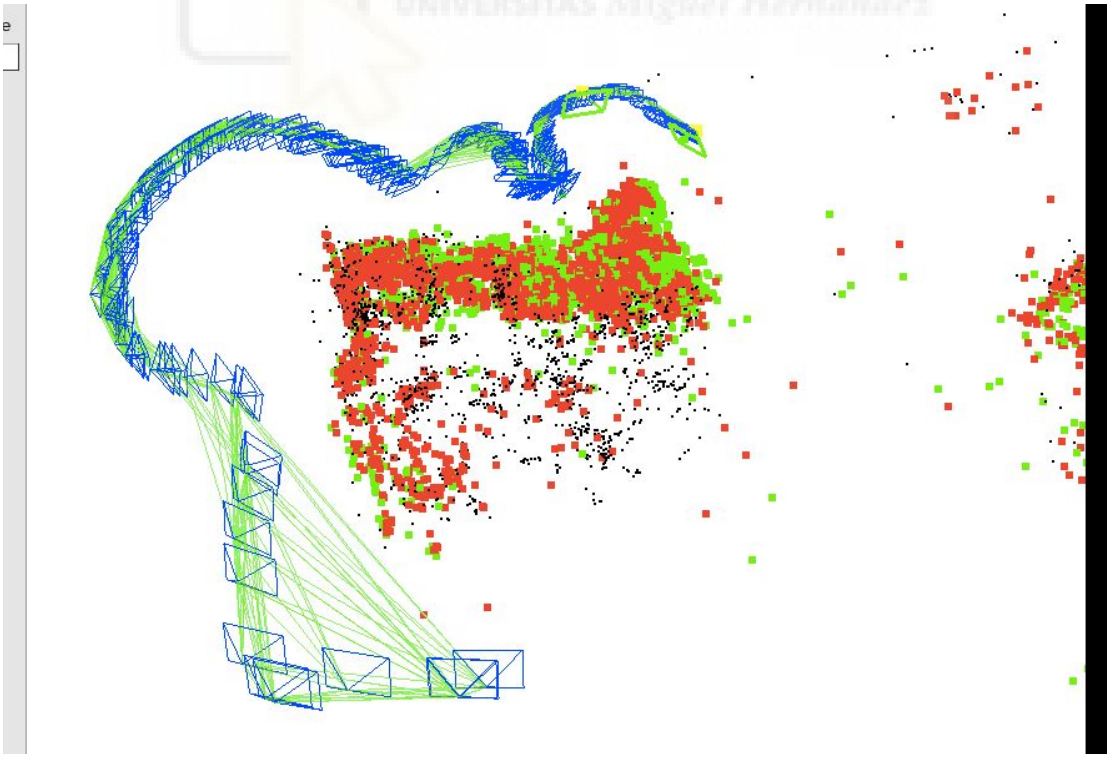


Si nos fijamos en los dos Trackers, vemos que son de tamaño diferente, eso se debe a que el segundo ha tenido que hacer un cambio de escala. Concretamente ha tenido que encogerse para poder adaptarse al sistema global.



En las dos imágenes siguientes se aprecia como los Trackers siguen la misma trayectoria, y no solo eso, sino que los puntos verdes y los puntos rojos, más o menos, coinciden en el espacio.





6. Conclusiones

Vistas ambas librerías, lo cual era el objeto de estudio de este trabajo, se puede llegar a algunas conclusiones sobre en qué momentos una librería es mejor que la otra. A continuación se pondrá de manifiesto el resumen de las conclusiones obtenidas.

ARCore es una librería muy **sencilla** de usar pero que tiene serias **limitaciones** para hacer SLAM a gran escala. Necesita un hardware relativamente moderno y caro para poder funcionar y ofrece una tasa de FPS bastante aceptable, entre 30 y 40. Esto la hace idónea para **entornos pequeños** como videojuegos con realidad aumentada, o realidad virtual. Otra gran virtud de esta librería es que soporta los lenguajes nativos de las plataformas soportadas a parte de C++.

ORB-SLAM es una librería que ofrece un resultado muy bueno a la hora de hacer **SLAM a gran escala** gracias a su sistema de "loop closing". Es más, precisamente este "loop closing" es el gran caballo de batalla de ORB-SLAM, puesto que pocos son los sistemas que lo incorporan. Soporta diferentes topologías de cámaras. Por contra sus **FPS no son tan buenos** como ARCore.

Otra ventaja que tiene, como se ha demostrado, es que **ORB-SLAM es escalable** y se le pueden introducir mejoras. Tiene, bajo mi punto de vista algunos problemas como los que se han relatado en sus respectivos apartados, pero puede servir perfectamente de base para desarrollar un nuevo sistema, ORB-SLAM3, donde se tengan todas estas cosas en cuenta.

A mi parecer, se podría decir que **ORB-SLAM podría convertirse en un producto perfectamente comercial**, pero ha de mejorar ciertos aspectos y se algo más rápida en el tiempo real.

6.1. Trabajos futuros

Por temas de tiempo, como ya se ha comentado, ha sido imposible el realizar las mejoras que a continuación se proponen, pero se prevé que no haya problemas a la hora de incorporar dichas mejoras.

6.1.1. ARCore

Como ARCore es un sistema totalmente cerrado, poca cosa se puede mejorar en ella. Aún así, se van a exponer un par de mejoras que podrían llevarse a cabo.

6.1.1.1. Histograma de puntos

Dado que en ARCore sólo contamos con puntos 3D, la primera idea hacer un histograma de los puntos 3D de los puntos que tiene alrededor un punto. El proceso sería algo parecido a este:

1. Para cada punto P_i que veamos en un momento determinado
    a. Hacemos un histograma por distancias de los puntos 3D Q_j que son visibles desde una escena en la que se vea P_i.
    b. Esto generaría algo parecido a [0, 1, 2, 0, 10, 11, ...]
        i. Esto significa que desde el punto P_i
            1. En el rango [0 cm → 10 cm[hay 0 puntos.
            2. En el rango [10 cm → 20 cm[hay 1 puntos.
            3. En el rango [20 cm → 30 cm[hay 2 puntos.
            4. En el rango [30 cm → 40 cm[hay 0 puntos.
            5. En el rango [40 cm → 50 cm[hay 10 puntos.
            6. En el rango [50 cm → 60 cm[hay 11 puntos.
            7. ...
2. Si encontramos 2 puntos que tienen un histograma parecido, pues podríamos iniciar un segundo test para saber si son o no el mismo punto, por ejemplo uno geométrico.

Pero este método tiene varios problemas:

- Si dos puntos están muy cercanos y tienen el mismo histograma, ¿cómo diferenciarlos?
- Esto valdría para ciertos puntos muy significativos en los que el histograma fuera muy diferenciable.
- Solo sería válido en casos en los que hubiésemos perdido de vista ciertos puntos y los hubiésemos vuelto a encontrar.
- Otros no previstos que pudieran aparecer en la implementación.
- El tiempo de computación crece exponencialmente con los puntos. Por lo que sería necesario el uso de alguna técnica de partición del espacio para poder llevar a cabo esto en tiempo real.

6.1.1.2. Puntos característicos personalizado

Otra opción para salvar el hecho de que ARCore no ofrezca sus puntos característicos es la posibilidad de ser nosotros mismos los que calculemos los puntos característicos con sus descriptores en 2D.

Es decir, dado que ARCore nos proporciona los puntos 3D, y dado que podemos calcular los puntos 2D donde se encuentran dichos puntos 3D, nada, o bueno, casi nada nos impide llamar a OpenCV para calcular por ejemplos los descriptores BRIEF de dichos puntos 2D de la cámara.

Lo que ocurre es que, aunque la idea puede ser buena buena, la implementación puede no ser tan directa o sencilla como pudiera parecer, hecho por el cual no se ha incorporado en este trabajo. De todas maneras se explican a continuación los procedimientos.

ARCore, te ofrece la imagen capturada por la cámara en una textura de OpenGL, y no hay otra opción en el NDK. Debido a ello, el primer pensamiento puede ser: "*Bueno, pues copio la textura de OpenGL a una zona de la RAM y trabajo sobre la RAM*". Esto tiene el problema de que traer memoria de la zona de OpenGL a memoria RAM cuesta mucho en términos temporales aunque sea sólo hacer una llamada. Previsiblemente, puede que la aplicación bajase a 10 FPS con sólo ese proceso. Si además, después de eso hemos de pasar un detector de puntos, más un descriptor, ..., pues se pierde toda la gracia del tiempo real.

La segunda opción que sí que podría funcionar, es intentar hacerlo todo por GPU. El inconveniente de hacerlo todo por GPU es que se tarda mucho tiempo en implementarlo. Por poner un ejemplo, Seth George Hall hace un FAST llamado ColourFAST en su **tesis** GPU Accelerated Feature Algorithms for Mobile Devices. Por lo tanto el tiempo requerido para esto sobrepasa con creces el disponible para hacer un TFM.

Al margen de lo anterior, también resulta un poco con mal sabor de boca, el tener que calcular los puntos característicos aunque sea sólo de ciertas zonas, ya que es trabajo doble puesto que ARCore seguramente lo ha calculado internamente.

6.1.2. ORB-SLAM

Por otro lado, en ORB-SLAM, como tenemos el código fuente, podemos seguir incorporando las características que se nos puedan ocurrir. En concreto, siguiendo la línea del TFM, se van a proponer algunos cambios.

6.1.2.1. Muelle

En realidad esta idea se basa en una técnica que se llevó a cabo en la asignatura de robótica móvil. Gracias a las clases del Máster, en un problema parecido, se le nos mostró un sistema bastante original de arreglar una problemática como esta.

La idea es la siguiente:

- Como ya se ha dicho, la idea original trata de buscar un frame compartido entre dos Trackers. Pues en vez de esto, cada cierto tiempo, una vez encontrado que dos Trackers comparten espacio, se genera un thread que cada "x" tiempo, busca otra coincidencia.
- En caso de encontrar otra coincidencia, tendremos que esos dos Trackers, el A y el B, están unidos por dos puntos en vez de por uno.
- Si se sigue se van a poder encontrar N coincidencias entre los Trackers A y B.
- Pues bien, imaginemos que esos puntos encontrados, son masas y las unimos con unos muelles.

La idea es hacer un sistema de minimización de energía compuesto de masas, muelles y reductores entre los sistemas A y B, tomando como referencia los puntos comunes encontrados.

6.1.2.2. N-Mejores

Siguiendo la línea anterior, y debido a que es posible que se encuentren multitud de puntos de intersección, otra posible idea es la comunmente se llama como la supresión de los no máximos. Es decir, Todas las coincidencias de frames no van a ser iguales, habrán frames en los cuales se hayan encontrado más coincidencias.

Podría ser buena idea el determinar un número determinado de puntos de anclaje máximo, como por ejemplo 10, los cuales serán los que mejores coincidencias se encuentren. Esto se podría combinar junto con el apartado anterior para configurar las K de los muelles. Cuanto una coincidencia sea más fuerte, la K será mayor.

6.1.2.3. El ID del Tracker

Existe un punto que puede ofrecer algo de confusión y es, cuando dos Trackers se “conocen”, ¿cual de los dos se pone como sistema inmóvil? Para este caso, como idea, se puede usar el id del Tracker. El Tracker que tenga menor ID se deja como sistema inmóvil y es el que tiene el Tracker con ID superior, el que se pasa al sistema de referencia del otro.

6.1.2.4. Thread de unión

Otra idea es especializar un thread para que vaya haciendo todos los cómputos pertinentes a la minimización de energía, buscar qué frame es el mejor y recalcular las poses relativas.

Este frame puede tener muy poca latencia, de esta manera tendríamos un thread que detecta puntos en comunes entre Trackers y el otro thread que se encargaría de gestionarlos.

7. Bibliografía

La bibliografía que se puede usar para investigar sobre el tema es la siguiente:

- Raúl Mur-Artal, J. M. M. Montiel and Juan D. Tardós. ORB-SLAM: A Versatile and Accurate Monocular SLAM System. IEEE Transactions on Robotics, vol. 31, no. 5, pp. 1147-1163, 2015. (2015 IEEE Transactions on Robotics Best Paper Award).
- Dorian Gálvez-López and Juan D. Tardós. Bags of Binary Words for Fast Place Recognition in Image Sequences. IEEE Transactions on Robotics, vol. 28, no. 5, pp. 1188-1197, 2012.
- B. D. Lucas and T. Kanade (1981), An iterative image registration technique with an application to stereo vision. Proceedings of Imaging Understanding Workshop, páginas 121--130
- Bruce D. Lucas (1984) Generalized Image Matching by the Method of Differences
- J. Y. Bouguet, (2001) . Pyramidal implementation of the affine lucas kanade feature Tracker description of the algorithm. Intel Corporation, 5.
- A Review of a Singularly Valuable Decomposition: The SVD of a Matrix, Reed Tillotson, June 6, 2013
- Conceptos y métodos en visión por computador, Enrique Alegre, Gonzalo Pajares y Arturo de la Escalera. Junio 2016
- Computer Vision: Algorithms and Applications. Richard Szeliski September 3, 2010
- Georg Klein and David Murray, "Parallel Tracking and Mapping for Small AR Workspaces", Proc. ISMAR 2007
- Georg Klein and David Murray, "Improving the Agility of KeyFrame-based SLAM", Proc. ECCV 2008
- LSD-SLAM: Large-Scale Direct Monocular SLAM, J. Engel, T. Schöps, D. Cremers, ECCV '14
- Semi-Dense Visual Odometry for a Monocular Camera, J. Engel, J. Sturm, D. Cremers, ICCV '13