Máster en Visión Artifical

Curso académico 2016/2017

Trabajo Fin de Máster

Autolocalización visual 3D usando mapas *RTAB-Map*

Autor:
Alberto Martín Florido
Tutores:
Prof. Dr. José María Cañas Plaza y Francisco Miguel Rivas Montero

Una copia de este proyecto, las fuentes del programa y vídeos de los experimentos están disponibles en la siguiente dirección:

`http://jderobot.org/Amartinflorido-tfm`

Agradecimientos

En especial quiero mostrar mi agradecimiento a mis tutores, José María Cañas y Francisco Miguel Rivas, por su infinita paciencia y dedicación que han demostrado a lo largo de los años. Agradecer a toda la comunidad de JdeRobot por continuar soñando con un proyecto de software libre impulsado desde nuestra universidad.

Por supuesto dar las gracias a mi familia y a mis seres queridos, que son los principales culpables que hoy pueda sentarme aquí a escribir mi Trabajo Fin de Máster.

Finalmente quiero dar las gracias a todas aquellas personas que en algún momento confiaron en mi, espero no haberles defraudado.

Resumen

Uno de los grandes problemas de la Visión Artifical actualmente es la localización 3D de una cámara dentro de una escena. Es un problema compartido con la comunidad robótica, donde se intenta localizar un robot dentro de algún entorno (conocido o desconocido). Se le conoce como SLAM (*Simultaneous Localization and Mapping*) e históricamente se han desarrollado soluciones para localizar robots con distintos tipos de sensores, como los láseres. Con la irrupción de las cámaras y la gran cantidad de información sensorial que proporcionan, la comunidad se centró en solucionar el problema mediante la Visión Artificial, dando lugar a la *localización visual* o *Visual-SLAM*.

Este Trabajo Fin de Máster aborda la localización visual para implementar un autocalibrador para sensores RGB-D que permita conocer con precisión la posición 3D del sensor dentro un entorno conocido previamente. Se ha modificado una de las herramientas más conocidas y potentes para la generación de mapas 3D (*RTAB-Map*), para extraer la información esencial sobre la estructura del mapa. Se ha desarrollado una librería que, a partir de un mapa del entorno, es capaz de encontrar la posición 3D de una cámara unicamente con la información visual (imágenes en color) recibida por ésta. La solución propuesta ha sido validada experimentalmente en distintos entornos de trabajo.

Se han desarrollado también un componente para la plataforma de software libre JdeRobot que recibe imágenes capturadas por una cámara y devuelve una estimación de la posición 3D de la cámara que tomó la imagen. Igualmente un visualizador que permite depurar el funcionamiento del algoritmo y la visualización de mapas 3D. Todo el contenido del Trabajo Fin de Máster se ha publicado dentro del proyecto JdeRobot de software libre para robótica y visión artificial.

Abstract

One of the most important problems in Computer Vision is the camera 3D localization. This problem came from the Robotics community, where they try to localize a robot in an environment (well-known or unknown), without GPS. This problem is known as SLAM (*Simultaneous Localization and Mapping*) and historically a lot of solutions were developed to localize robots with different sensor types, the best results were obtained with laser sensors. With the apparition of the cameras and their capacity to get a lot of sensory information, the community focus on to try to resolve the localization problem with Computer Vision techniques. The self-localization problem only with visual information is known as *visual localization* or *Visual-SLAM*.

This Master's Thesis addresses the visual localization problem to implement an RGB-D self-calibrator to estimate the 3D pose of a RGB-D sensor in a well-known environment. First, a very popular and powerful tool to build 3D maps (*RTAB-Map*) has been modified to extract essential information about the map structure. To the algorithm's implementation purpose in this project, we develop a library that with an environment map as input, it estimates a camera pose only with the picture captured by the camera. This implementation has been experimentally evaluated in different environments with satisfactory results.

Furthermore two tools have been developed in the project; the first one is a component for the free software framework JdeRobot that receives pictures from a camera and returns a pose 3D estimation of the camera that took the picture. The second one is a graphical tool to debug the algorithm implemented and to visualize 3D maps. All content developed in this Master's Thesis will be published in JdeRobot project, in this way, anyone can use it.

Índice general

Capítulo 1

Introducción

La vista es probablemente el sentido más importante para los seres humanos, a través de nuestros ojos percibimos el mundo pero en realidad es nuestro cerebro el que da significado a aquello que vemos. El ojo humano es un órgano que detecta la luz transformando la energía lumínica en señales eléctricas que son enviadas al cerebro a través del nervio óptico. En esencia nuestros ojos son sensores que recogen la luz que rebota en los objetos de nuestro alrededor y es nuestro cerebro el que percibe el mundo que vemos a través de nuestro ojos. Desde niños nuestro cerebro es entrenado para entender todo aquello que nuestros ojos perciben, así si un bebe con el sentido de la vista ya desarrollado viera un coche podría, distinguir la figura del coche pero no podría saber que lo que está viendo es un coche. Para que ese bebe pueda distinguir lo que es un coche o una casa necesitará años de aprendizaje, aunque sus sensores de luz funcionen correctamente.

Desde los presocráticos el ser humano se ha interesado por el funcionamiento de la visión humana. Pero es a partir del siglo XIX con Hermann von Helmholtz cuando comienza el estudio científico de la percepción visual. Desde entonces se han realizado cientos de estudios para intentar dar con la respuesta, y aunque conocemos mucho sobre la visión humana, todavía hoy seguimos sin conocer todos los detalles de este complejo sistema.

Con la llegada de las cámaras, y más concretamente con la llegada del mundo digital, muchos científicos se preguntaron si serían capaces de emular las capacidades del cerebro humano para comprender aquellos impulsos eléctricos que las cámaras podían recoger. En este nuevo campo de investigación los ojos serían las cámaras y el cerebro las computadoras.

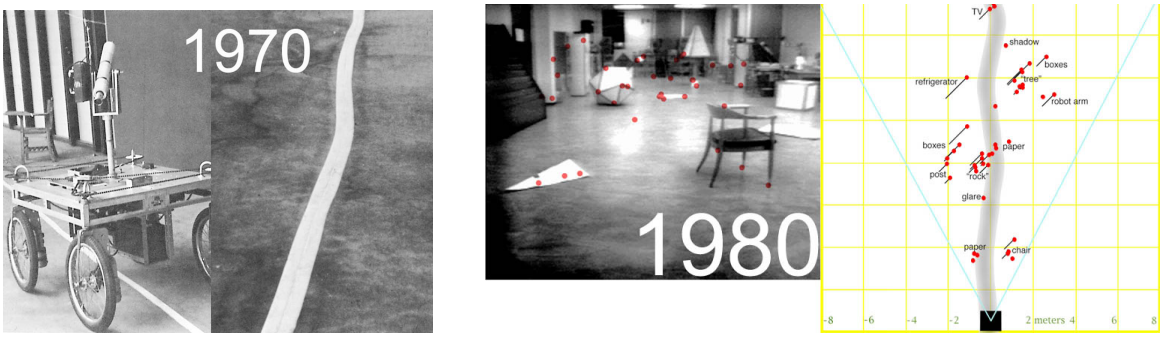
1.1. Visión artificial

Existen varias definiciones de la visión artificial, a continuación pueden ver una definición extraída de la Wikipedia que resume el significado de esta rama de la ciencia:

> La visión artificial o visión por computador es una disciplina científica que incluye métodos para adquirir, procesar, analizar y comprender las imágenes del mundo real con el fin de producir

información numérica o simbólica para que puedan ser tratados por un computador.

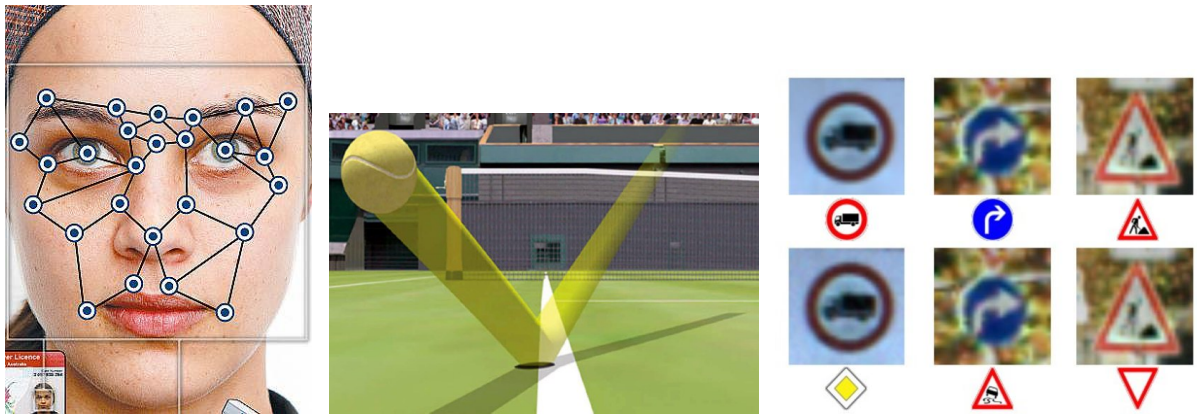
Es en la década de 1960 cuando la visión artifical comienza como una rama de la inteligencia artificial, en [1] se puede ver un vídeo sobre los estudios en visión del MIT en 1959. En 1966 se da a conocer el *project summer*[2] del MIT, que consiste en la construcción de un sistema visual capaz de reconocer patrones.



(a) *Stanford cart* (b) *Stanford cart* mejorado

Figura 1.1: Primeros robots con cámara que hicieron uso de la visión artificial

En los 70 la universidad de Stanford creó el *Stanford Cart* que fue el primer robot móvil controlado por un ordenador. Este robot era capaz de seguir una línea blanca gracias a la cámara que tenía incorporada. Con los datos recolectados por su cámara se comunicaba a través de ondas de radio con una estación de trabajo, que realizaba los cálculos necesarios para su movimiento. En los 80 apareció el *Stanford Cart* mejorado, que de nuevo, gracias a su cámara era capaz de sortear obstáculos en una habitación.



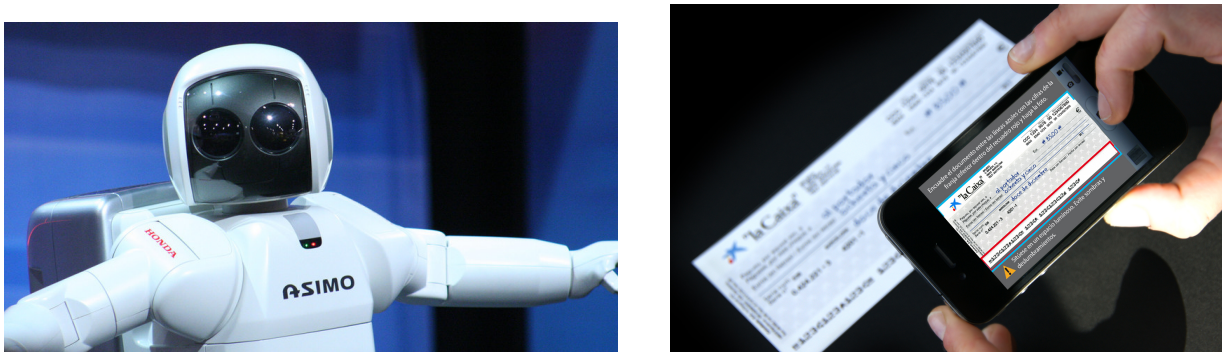
(a) Reconocimiento facial (b) Ojo de halcón (c) Detección de señales

Figura 1.2: Algunas aplicaciones de la visión artificial en la actualidad

Desde aquellos días se ha avanzado mucho en la visión artificial, en gran medida por las mejoras y abaratamiento de los computadores y de cámaras. Hoy día se pueden encontrar multitud de aplicaciones de visión artificial tanto de uso cotidiano como en entornos industriales o empresariales. Además, se ha introducido como una herramienta más en múltiples y diversos sectores como por ejemplo; la tecnología

[1] `https://infinitehistory.mit.edu/video/eye-robot-studies-machine-vision-mit-and-tx-o-computer-1959`
[2] `https://dspace.mit.edu/bitstream/handle/1721.1/6125/AIM-100.pdf?sequence=2`

OCR para el reconocimiento de escritura la digitalización de textos o como sistema de reconocimiento de cheques en bancos, sistemas biométricos que permiten reconocer las huellas de una persona o incluso su cara en sistemas de seguridad, sistemas de detección de señales de tráfico en la industria automovilística, la clasificación de objetos o piezas en sistemas robotizados de montaje, diversos usos en el deporte como el ojo de halcón en el tenis o en el fútbol, etc.



(a) Robot Asimo (b) Sistema OCR de reconocimiento de cheques

Figura 1.3: Más aplicaciones de la visión artificial en la actualidad

En definitiva la visión artificial es un campo joven al que todavía le quedan grandes y difíciles retos que afrontar. Cada vez más podemos ver sus aplicaciones en el día a día y en un futuro será todavía mayor. El trabajo fin de máster que se presenta en este documento se encuadra dentro de las aplicaciones actuales de la visión artificial.

1.2. Visión tridimensional

Dentro de la visión artificial podemos encontrar la visión tridimensional, la cual se encarga de estudiar las características geométricas de la escena y la cámara, con el fin de obtener información espacial a partir de las imágenes capturadas por una cámara. Aunque las cámaras actuales son muy complejas, se puede trabajar con un modelo simplificado que nos permite extraer las características geométricas de la cámara.

A priori puede parecer que obtener información espacial a partir de una única cámara es una tarea no muy compleja, pero lo cierto es que, con una sola cámara y una sola vista es imposible obtener información en 3 dimensiones. El mejor ejemplo somos los humanos, y es que no es casualidad que tengamos dos ojos en lugar de uno. Con un único ojo (o una única cámara) nos es imposible determinar fácilmente la distancia a la que se encuentra un objeto. Podemos ver el objeto, determinar si está en una determinadas coordenadas X e Y (más arriba a la izquierda), pero, al tener solamente una única vista del objeto no podemos determinar la profundidad a la que se encuentra.

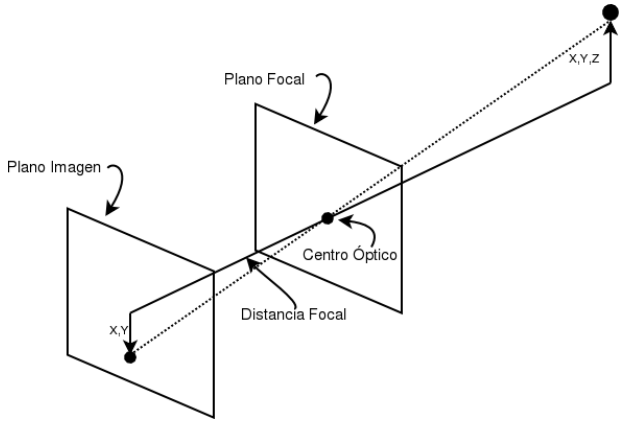


Figura 1.4: Modelo de cámara Pin Hole

La Figura 3.1 muestra el modelo Pin Hole, un módelo geométrico muy simple que representa la geometría de una cámara. Con una sola cámara que apunta por el eje Z y teniendo la geometría descrita en la imagen, podríamos coger el punto 3D (X,Y,Z) y proyectarlo sobre el plano imagen obteniendo así un punto 2D. Sin embargo si cogiésemos ese punto recién proyectado e intentásemos reproyectarlo de nuevo al mundo de 3D, no tenemos información sobre la profundidad del punto 3D original, la reproyección del punto 2D podría estar en cualquier punto que pasase por la recta que pasa por nuestro punto 2D y por el centro óptico.

1.2.1. Sistema de visión estereoscópico

Con un sistema de visión monocular no es posible reproyectar un punto 2D del plano de imagen a un punto 3D en el mundo. No podemos obtener información 3D del entorno con solo una vista de la escena, por este motivo se utilizan varias vistas que nos permitan determinar la distancia (el eje Z) de un punto respecto de las cámaras. La Figura 1.5 muestra el esquema típico de un sistema de visión estereoscópico o simplemente sistema estéreo.

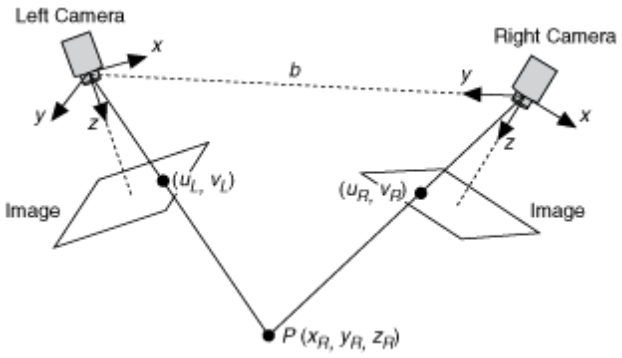
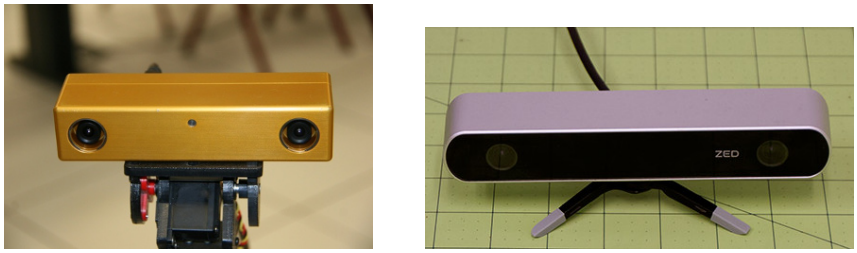


Figura 1.5: Sistema de visión estereoscópico

La solución clásica para reproyectar un punto 2D del plano imagen al mundo es sencilla, dado a que ahora cada cámara puede formar una recta desde el centro de proyección en dirección al punto 3D. Si ambas cámaras observan el mismo punto, en algún momento dichas rectas intersectarán entre sí dando

como resultado un punto 3D. Con esta sencilla triangulación se obtiene información 3D de la escena con dos cámaras.

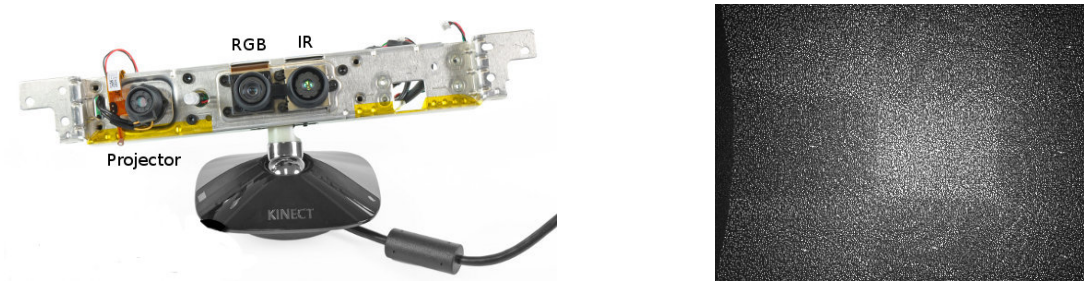


(a) Bumblebee2 (b) Zed

Figura 1.6: Cámaras estereoscópicas

1.2.2. Sensores RGB-D

Un sensor RGB-D es aquel que combina la información de color obtenida con una cámara RGB con la información de profundidad (distancia a un objeto). El ejemplo más famoso es Kinect, que fue lanzada al mercado de consumo en noviembre de 2010 por Microsoft. El sensor fue desarrollado y patentado [23] por *Prime Sense*, que después lo licenció a Microsoft (Kinect) y Asus (Asus Xtion), y finalmente fue adquirida por Apple.



(a) Estructura interna de la Kinect v1 (b) Patrón de luz estructurada

Figura 1.7: Kinect v1 y su patrón de luz estructurada

La Figura 1.7a muestra la estructura interna de la Kinect v1, podemos apreciar un proyector de infrarrojos, una cámara RGB y una cámara de infrarrojos. El proyector emite un patrón infrarrojo de puntos conocido que es capturado por la cámara de infrarrojos y, el propio hardware del dispositivo infiere la profundidad de la escena píxel a píxel, comparando la deformación [68] de dicho patrón con el capturado. De este modo el sensor proporciona dos imágenes correladas con la información RGB y de profundidad en cada píxel de la imagen.

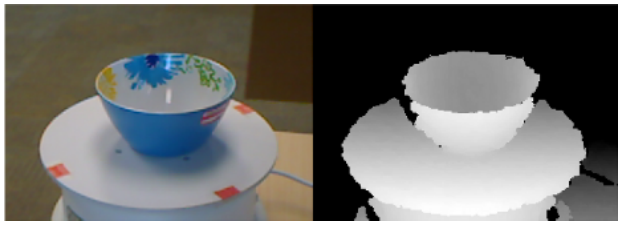


Figura 1.8: Imagen RGB y de profundidad

En 2014 Microsoft lanzó al mercado la versión 2 de Kinect. En esta nueva versión se abandonó el método de la proyección del patrón de infrarrojos y se sustituyó por un método de tiempo de vuelo (*Time-of-flight*). En el que el dispositivo es capaz de medir el tiempo que tarda en llegar y regresar una señal infrarroja (emitida desde la Kinect 2) hasta un objeto cercano, calculando así la distancia que ha recorrido hasta impactar con el objeto. La nueva versión de Kinect también trajo una mejora de las imágenes obtenidas pasando de imágenes de $1280x960$ @12 fps en la primera versión a resoluciones de $1920x1080$ @30 fps.

Otro uso de esta tecnología son las interfaces de usuario. En 2015 *Intel* sacó al mercado su propio sensor RGB-D llamado *Intel RealSense*. Ese mismo año *Asus*, *HP*, *Dell*, *Lenovo* y *Acer*, montaron en algunos de sus portátiles este sensor para permitir al usuario controlar determinadas funciones del ordenador mediante gestos. Además, este mismo sensor ha sido utilizado para distintas aplicaciones robóticas, como la navegación en interiores con robots con ruedas o la navegación en interiores con drones.

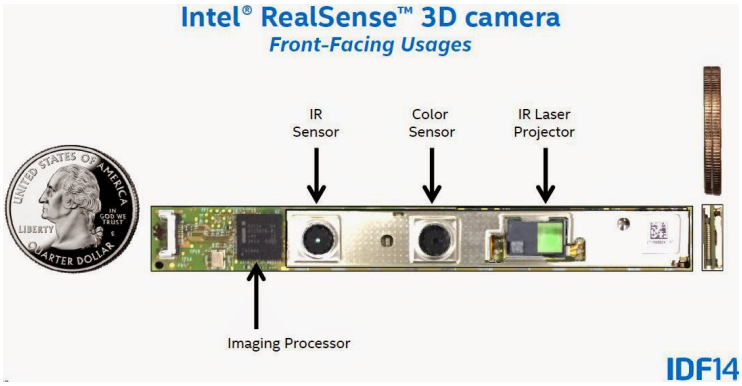


Figura 1.9: *Intel RealSense*

1.3. Autolocalización visual

La autolocalización visual es uno de los grandes problemas de la visión artificial, consiste estimar la posición y orientación 3D de una cámara a partir de las imágenes obtenidas por ésta. En el caso de los sensores RGB-D se utiliza adicionalmente la información de profundidad para estimar la pose de la cámara. Con la información de profundidad el problema se simplifica, pero aún así, sigue siendo un problema complejo que requiere un conocimiento profundo de la geometría de la cámara.

Conocer la pose de la cámara abre el abanico a un gran número de aplicaciones. Un ejemplo claro es la navegación de interiores con robots, un robot puede moverse por un entorno totalmente nuevo para

él sin chocar con los objetos que se encuentre a su paso. Un ejemplo claro es el robot aspirador *iRobot Roomba-980*. Este pequeño robot está dotado de una cámara que le permite realizar un mapa de su entorno y navegar por él con el fin de atrapar toda la suciedad. El robot comienza explorando el entorno recogiendo características del entorno mientras se va moviendo, además también hace uso de otros sensores instalados en la aspiradora como una IMU, un giroscopio y la odometría de sus motores. Con la riqueza sensorial de la cámara y los datos del resto de sensores, este robot consigue crear un mapa del entorno por el que después navegará sin colisionar con los obstáculos que allí se encuentren. Para lograr este objetivo, el robot ejecuta un algoritmo de *Visual-SLAM* que le permite localizarse continuamente dentro del mapa.

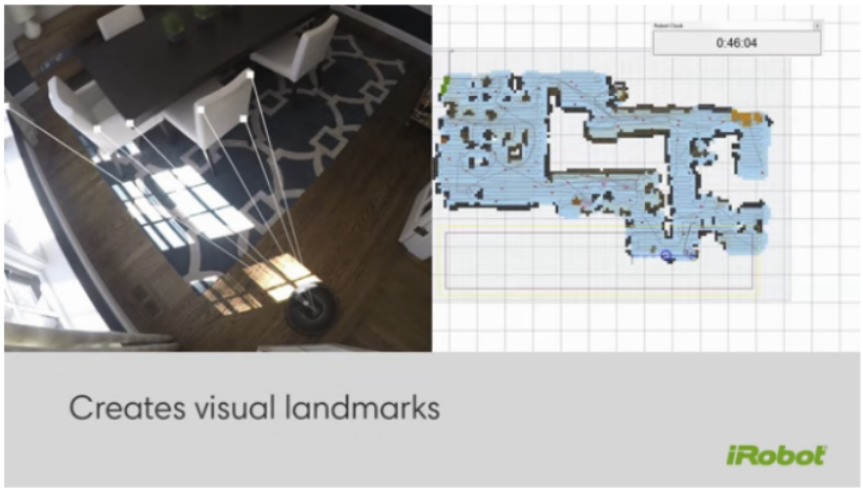


Figura 1.10: Generación de mapa del entorno del *iRobot Roomba 980*

Otro ejemplo más cotidiano es el de la realidad aumentada, para poder visualizar a través de nuestro *smartphone* información sobreimpresa o animaciones 3D, la aplicación que las coloca en pantalla necesita saber la pose 3D de la cámara para que el efecto parezca realista. En el terreno del ocio hay multitud de videojuegos que utilizan la realidad aumentada para entretener a sus jugadores. Uno de los más famosos es Pokemon Go, que aunque no realiza una estimación de la posición 3D, es un claro ejemplo de cómo la realidad aumentada puede llegar a cientos de miles de personas.

Recientemente Sony sacó al mercado su kit de realidad virtual *PlayStation VR*, está formado por dos componentes principales; en primer lugar el visor de realidad virtual que está compuesto por una pantalla de $5,7$ pulgadas la cual lleva incorporado un acelerómetro y un giróscopo. La segunda parte del sistema consiste en un conjunto de cámaras y balizas. Concretamente un sistema esteresocópico y las balizas, que en este caso son unos mandos denominados *Move* que emiten una luz muy característica, igual que el mando de la videoconsola y el visor VR. Gracias a esta configuración el sistema es capaz de determinar la posición del jugador que tiene enfrente de la cámara, y haciendo uso de ella, dotar a su sistema de realidad virtual de mucho realismo.

Figura 1.11: *PlayStation VR*

Además de las aplicaciones de juegos con realidad aumentada, en los últimos años el campo de los *smartphones* también ha visto como es posible el uso de la autolocalización visual para multitud de aplicaciones. El ejemplo más claro es *Google Tango* que, en su versión de *tablet* o *smartphone*, pone a disposición del programador todas las herramientas necesarias, tanto hardware como software, para el desarrollo de aplicaciones de autolocalización visual.

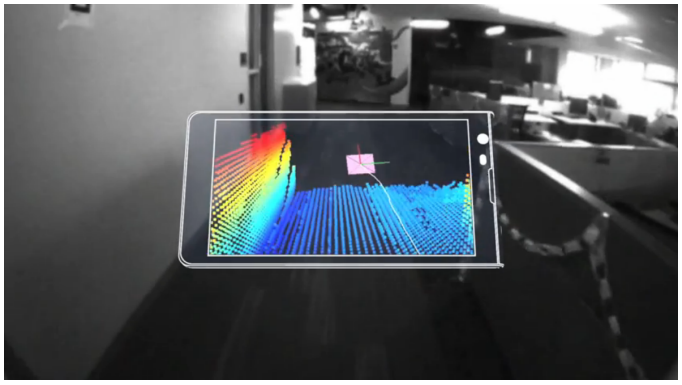


Figura 1.12: *Google Tango*

En la tienda de aplicaciones de Android podemos encontrar desde aplicaciones para la generación de mapas 3D, herramientas que combinan la realidad aumentada y la autolocalización visual para visualizar el modelo 3D de un mueble en el propio salón de casa, aplicaciones para medir objetos. En 2016 fue lanzado al mercado de consumo el primer *smartphone* con esta tecnología. Este dispositivo incluye un sensor RGB-D de luz estructurada (tecnología desarrollada por *Prime Sense*) y todo el software necesario para la obtención de los datos del sensor y otros algoritmos para la autolocalización.

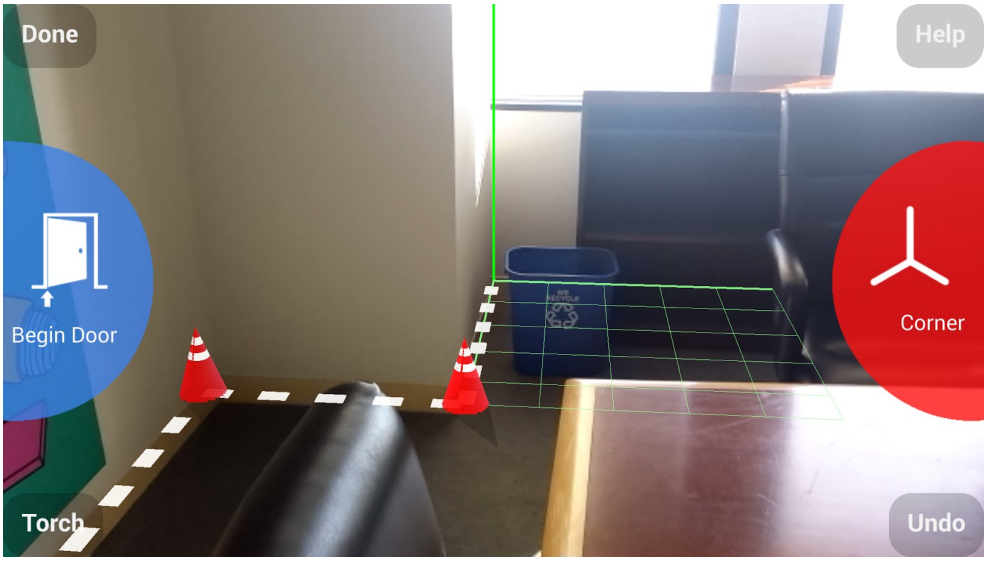


Figura 1.13: Aplicación *Magic Plan*

La Figura 1.13 muestra una captura de la aplicación *Magic Plan*. Esta aplicación, disponible para iOS y Android, permite hacer planos de una habitación unicamente realizando fotografías con la cámara del *smartphone*. Para ello utiliza técnicas de realidad aumentada que le permiten medir con exactitud la habitación donde trabaja.

1.3.1. Antecedentes en el laboratorio de robótica de la URJC

A continuación se describen algunas aplicaciones de autolocalización visual realizadas en el grupo de Robótica de la Universidad Rey Juan Carlos, que sirven de base y contexto inmediato para el Trabajo Fin de Máster desarrollado.

En el 2009, Luis Miguel López implementó para su proyecto fin de carrera [52] un algoritmo de autolocalización visual en tiempo real con una sola cámara (sin visión estereoscópica). En este proyecto se consiguió con éxito la estimación de la pose 3D de la cámara haciendo uso de un Filtro de Kalman extendido. Para su desarrollo, Luis Miguel, basó su diseño en la técnica de monoSLAM propuesta por Andrew Davison[13]. Los resultados obtenidos en su proyecto se pueden encontrar en su mediawiki[3] de JdeRobot.

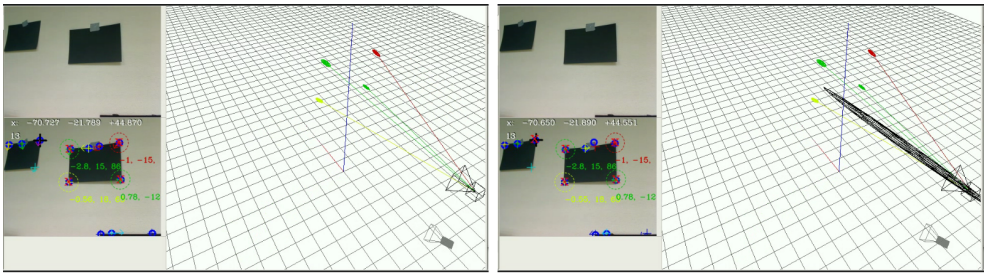


Figura 1.14: Estimación de la pose de una cámara mediante EKF

[3] `http://jderobot.org/Lm.lopez-pfc-teleco`

Alejandro Hernández en su trabajo fin de máster de 2013, Autolocalización visual aplicada a la Realidad Aumentada [8], hizo uso de distintas técnicas como DLT, monoSLAM o PTAM para localizar una cámara en el espacio 3D. Una vez localizada, Alejandro incrustó en la visualización distintas secuencias en 3D dimensiones para crear una aplicación de realidad aumentada[4].

Figura 1.15: Aplicación de realidad aumentada

Haciendo uso de sensores RGB-D, Daniel Martín presentó en 2013 su proyecto fin de carrera donde implementó un algoritmo de odometría visual [48] para estimar la pose y la trayectoria de un sensor RGB-D[5]. En su trabajo Daniel consiguió estimar el movimiento relativo entre dos fotogramas consecutivos, mediante el uso de SVD y RANSAC, lo que le permitió estimar la posición 3D de su Kinect.

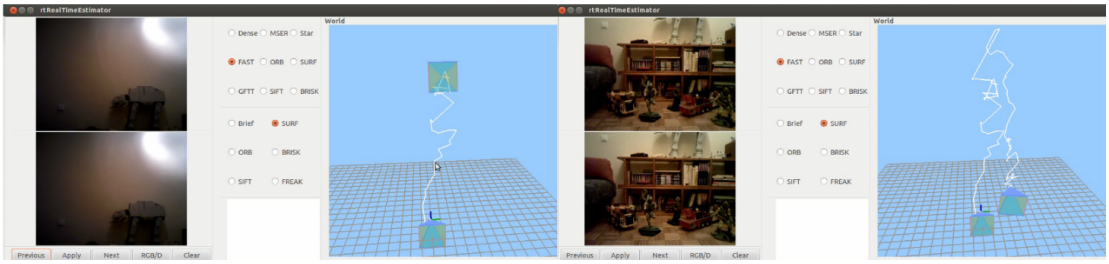


Figura 1.16: Estimación de la pose de un sensor RGB-D

Otra manera de realizar una estimación de la pose de una cámara es haciendo uso de marcadores, en 2014, Yazmin Lucy Cumberbirch implementó un algoritmo de autolocalización visual [9] haciendo uso de marcadores Aruco. Yazmin portó su algoritmo a una aplicación Android consiguiendo así una aplicación de realidad aumentada para *smartphones* que permitía visualizar una película encima de una pantalla real apagada[6].

[4] `http://jderobot.org/Ahcorde-tfm`
[5] `http://jderobot.org/Dmartino-pfc`
[6] `http://jderobot.org/Ycumberbirch-pfc`

Figura 1.17: Aplicación Android de realidad aumentada

En el 2015, Alberto López-Cerón terminó su Trabajo Fin de Máster sobre autolocalización visual basada en marcadores [42], el cuál terminó convirtiéndose en una publicación científica [67]. El trabajo de Alberto consistió en estimar la posición de un *drone*[7], haciendo uso de sus cámaras, a través de marcadores *AprilTags*[8]. Un dato interesante obtenido durante la realización de sus proyecto, es que esta técnica tiene un límite práctico de unos 4 metros.

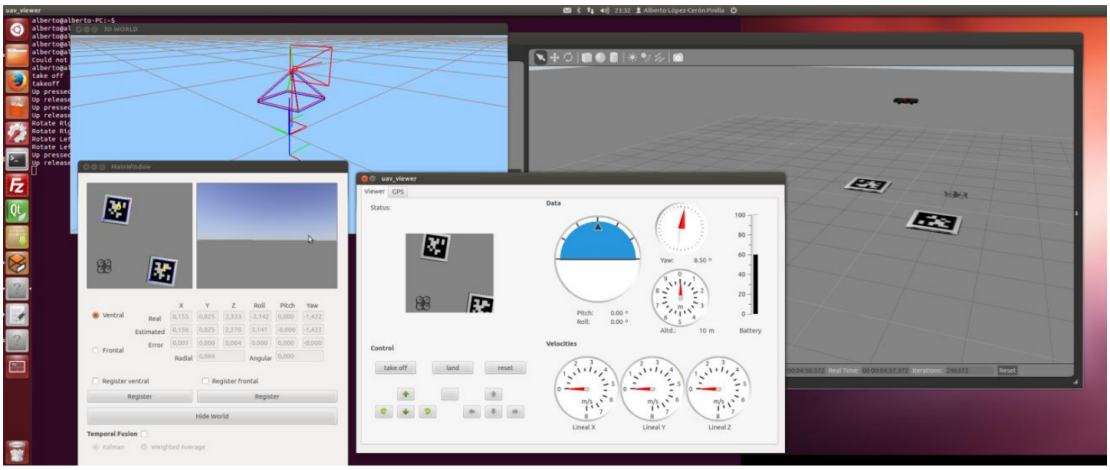


Figura 1.18: Autolocalización basada en balizas de un *drone*

Este mismo año, Eduardo Perdices García defendió su "Tesis Doctoral sobre Técnicas para la localización visual robusta de robots en tiempo real con y sin mapas" [22]. En su tesis Eduardo diseño un algoritmo llamado SVDL de autolocalización visual con una sola cámara capaz de funcionar tanto en entornos con un mapa conocido como sin él y en tiempo real. El algoritmo es capaz de trabajar en entornos con simetrías. Además, es computacionalmente ligero, capaz de funcionar eficazmente en procesadores modestos[9].

[7] `http://jderobot.org/Alopezceron-tfm`
[8] `https://april.eecs.umich.edu/software/apriltag.html`
[9] `http://jderobot.org/Eperdices_PhD`

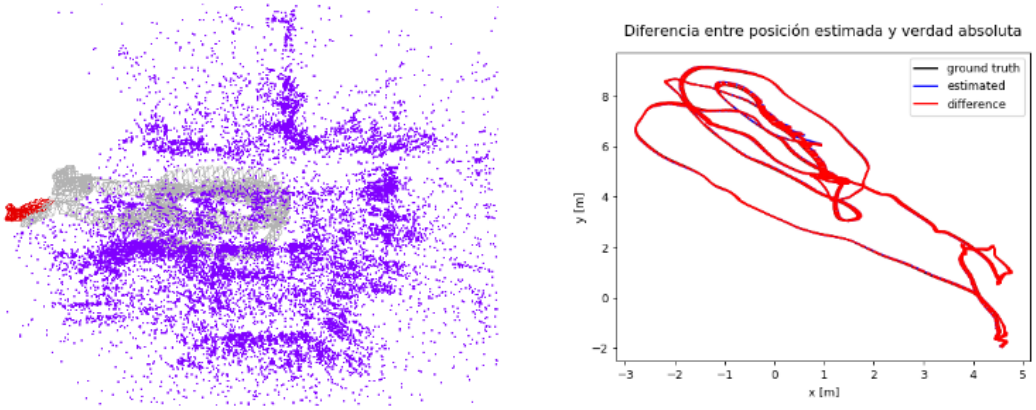


Figura 1.19: Autolocalización visual mono cámara en mapa desconocido con algoritmo SVDL

Otros algoritmos y técnicas disponibles, creadas fuera del laboratorio de robótica de la URJC, también han servido de base para este TFM. Las más relevantes se repasan con detalle en el capítulo de estado de arte (capítulo 3).

En los siguientes capítulos el lector encontrará los objetivos (ver capítulo 2) del Trabajo Fin de Máster, la infraestructura tanto hardware como software necesaria para su desarrollo (ver capítulo 4), una revisión sobre los algoritmos de autolocalización visual (ver capítulo 3), la descripción del algoritmo desarrollado en el TFM (ver capítulo 5), el conjunto de experimentos desarrollados para su evaluación (ver capítulo 6) y finalmente un capítulo dedicado a las conclusiones obtenidas durante el desarrollo (ver capítulo 7).

Capítulo 2

Objetivos

En este capítulo se expone en primer lugar la descripción del problema que se quiere resolver con la realización de este Trabajo Fin de Máster. A continuación los requisitos planteados al inicio del proyecto y finalmente el plan de trabajo empleado durante la realización de este trabajo.

2.1. Descripción del problema

La calibración de sensores RGB-D es una tarea compleja, monótona y aburrida. Existen herramientas para conocer los parámetros intrínsecos de una cámara RGB, pero si quieres trabajar con información de profundidad usando sensores RGB-D necesitas conocer dónde se encuentra la cámara dentro del mapa 3D en el cual trabaja, por tanto el problema no se limita sólo a conocer los parámetros intrínsecos de la cámara, sino que también es necesario conocer su posición 3D.

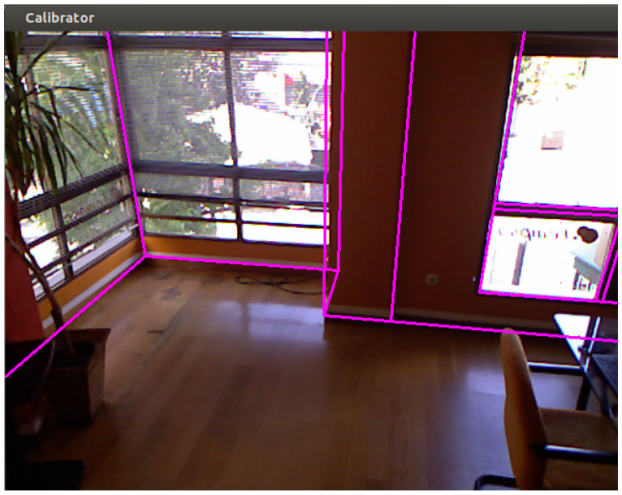


Figura 2.1: Proyección de líneas sobre un despacho

En años anteriores, distintos alumnos exploraron este problema en el laboratorio de robótica de la URJC con el fin de darle una solución. El primer enfoque consistió en proyectar todo el mundo que la cámara ve e intentar realizar un ajuste de su pose para que todos los elementos del mundo proyectado

fueran coherentes. La idea es que si eres capaz de realizar una representación interna del mundo fiel a lo que estás viendo, eso significa que estás en la posición 3D correcta.

Figura 2.2: Patrón de calibración 3D

El segundo enfoque esta basado en un patrón 3D de calibración. Es necesario contar con un mapa del entorno donde se va a colocar la cámara, un punto de referencia dentro del mapa y un patrón 3D. Es necesario realizar mediciones para conocer la posición 3D de la cámara. Es un procedimiento bastante tedioso que requiere mucho esfuerzo para poder ser llevado a cabo con éxito.

Este proyecto plantea un tercer enfoque que consiste en el desarrollo de un algoritmo que dado como entrada un mapa del entorno y una imagen RGB no perteneciente al mapa pero tomada con la misma cámara, sea capaz de estimar la pose 3D de la cámara que tomó la imagen RGB de entrada en el sistema de coordenadas en el cual está expresado el mapa.

Teniendo como objetivo global el desarrollo del software que implementa esta autolocalización 3D visual, lo cierto es que este proyecto puede dar solución a la autocalibración de parámetros extrínsecos de sensores RGB-D, cubriendo así la necesidad del laboratorio de robótica de la URJC. Esta meta la hemos articulado en cuatro subojetivos concretos:

1. El desarrollo de una librería que implementa el algoritmo de autolocalización visual 3D.
2. Desarrollo de un componente, que utilizando la librería anterior ofrezca mediante alguna interface

software a otros módulos la posición 3D de una imagen RGB dada. Este módulo se podrá integrar en una aplicación robótica como módulo de autolocalización.

3. Desarrollo de una aplicación gráfica que visualice pasos intermedios y finales y permita la depuración del algoritmo.

4. Evaluación experimental del sistema desarrollado.

2.2. Requisitos

En las fases iniciales del proyecto, después de plantear el problema, se definieron una serie de requisitos que la solución a desarrollar debía cumplir.
La entrada del algoritmo:

- Un **mapa del entorno**. Un mapa está compuesto por imágenes RGB, imágenes de profundidad, pose de las cámaras desde las que han tomados las imágenes y los parámetros intrínsecos de éstas.
- Una **imagen RGB** capturada con la misma cámara con la fue creado el mapa, tomada en el entorno de trabajo y que no pertenezca al mapa generado previamente.

La salida del algoritmo:

- La **estimación de la pose 3D** de la cámara que tomó la imagen de entrada en formato matriz (matriz de rotación más vector de translación) o un cuaternión.

Para la evaluación del algoritmo se tendrán en cuenta los siguientes errores:

- El **error de reproyección**.
- El **error de translación** en metros.
- El **error en la orientación** de la cámara.

En cuanto a la calidad y rendimiento temporal:

- Se espera que un **porcentaje superior al 50** % de las estimaciones se realice con un **error de reproyección inferior a 5 píxeles**.
- **No es una aplicación para ser usada en tiempo real**, pero igualmente se espera un rendimiento aceptable.

Requisitos del software:

- **Modular**, el software desarrollado debe poder ser utilizado como componente aislado, o como una librería que lo integre en otro proyecto.
- **Robustez**, el software debe ser robusto frente a errores previstos o no previstos. Como condiciones de carrera, entrada inválida o corrupta de datos al algoritmo, tratamiento de excepciones de las distintas librerías incorporadas, etc.

- **Tests**, la librería desarrollada para el proyecto deberá superar ciertos tests que aseguren su correcto funcionamiento.
- **Plataforma**, el software será desarrollado para plataformas GNU/Linux Debian Jessie o Ubuntu 16.04.
- **Integración**, tanto la librería como las aplicaciones desarrolladas se integrarán en el entorno de software libre JdeRobot para robótica y visión artificial, en su versión 5.5.
- **Documentación**, todo el software desarrollado debe estar documentado.

2.3. Plan de trabajo

El proyecto se ha dividido en distintas etapas con el fin de alcanzar determinados hitos de una manera progresiva asegurando así una dificultad incremental durante el desarrollo.

- **Estudio de técnicas de optimización**, en esta fase se comenzará con un acercamiento progresivo al problema de la optimización. Durante algunas semanas se estudiaran, se implementaran y se evaluaran distintos algoritmos de optimización para resolver problemas simples de modo que permitan un aprendizaje continuo. En [1] puede encontrar algunos de los resultados obtenidos.
- **Desarrollo del algoritmo de localización**, en la fase central del proyecto se diseñará e implementará el algoritmo encargado la autolocalización 3D de la cámara propuesto en este trabajo.
  - **Evaluación del algoritmo**, una vez se alcance la primera versión del algoritmo se comenzará de manera paralela con su evaluación. De este modo se puede iterar sobre ambas fases, desarrollo y evaluación, hasta conseguir los objetivos planteados.
  - **Desarrollo de test unitarios**, en el desarrollo de software es muy importante asegurar que los desarrollos funcionan como se esperan, por este motivo se diseñarán una batería de test unitarios que prueben la implementación software del algoritmo.
- **Desarrollo de herramienta de visualización**, con el fin de poder depurar de manera visual el funcionamiento del algoritmo, se desarrollará una herramienta con inteface gráfica de usuario que permita mostrar los resultados del algoritmo.
- **Evaluación de los resultados**, con el algoritmo ya desarrollado se procederá a la evaluación de los resultados obtenidos. Es esta fase se debe comprobar como de buenos o malos son las estimaciones realizadas.
- **Desarrollo de la herramienta**, con el fin de disponer de un programa integrado en JdeRobot que otros usuarios pudieran utilizar, se desarrollará un componente que haga uso de la librería que implementa el algoritmo.
- **Documentación**, en la última fase del proyecto se documentará todo el software desarrollado de modo que en un futuro, otros programadores puedan continuar con el desarrollo del mismo.

[1] `http://jderobot.org/Amartinflorido-tfm`

Durante todo el desarrollo del Trabajo Fin de Máster se mantuvieron reuniones semanales con los tutores de modo que se permitiese un correcto desarrollo del proyecto. Todo el progreso del proyecto se publicó periódicamente en la bitácora del proyecto[1]. Además, haciendo uso de la infraestructura de JdeRobot, todo el código se ha publicado en el repositorio *git* del proyecto[2].

[2] `https://github.com/JdeRobot/slam`

Capítulo 3

Estado del arte

En este capítulo se revisan los algoritmos más representativos de autolocalización visual 3D y de generación de mapas 3D, alineados con la temática del Trabajo Fin de Máster.

3.1. Localización visual basada en geometría

Esta técnica consiste en determinar la posición en la que se encuentra la cámara utilizando para ello únicamente la información visual, un modelo de cámara y conocimiento geométrico adquirido del entorno en el que se mueve el sensor, o de parte de él, como balizas por ejemplo.

3.1.1. Modelo de cámara Pin Hole

Una cámara Pin Hole, es un modelo de cámara muy simple que tiene únicamente un pequeño orificio por donde entra la luz, ni siquiera tiene lente. Si colocásemos papel fotográfico dentro de la cámara y dejásemos que entrará la luz por su orificio, al cabo de un rato, tendríamos una foto invertida de la escena que esté justo enfrente de la cámara.

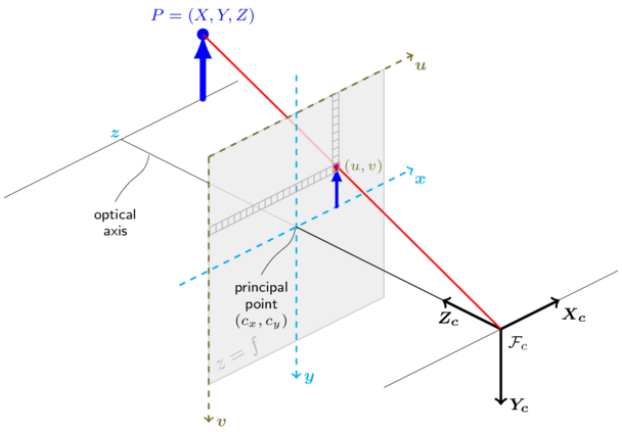


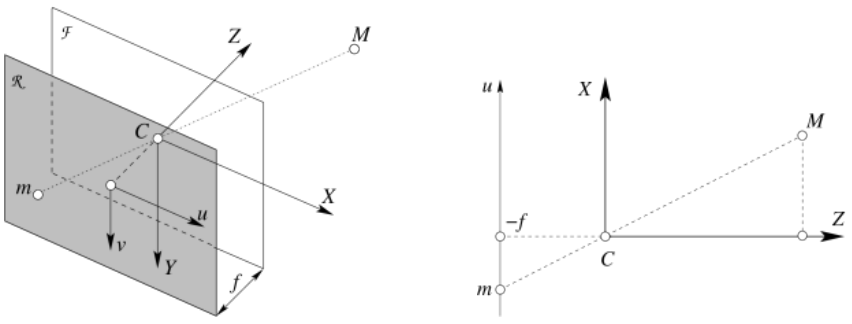
Figura 3.1: Modelo de cámara Pin Hole

La sencillez de este modelo lo hace ideal para crear un modelo geométrico de una cámara. No incluye distorsiones o el emborronado de objetos no enfocados provocados por las lentes. El modelo asume que todos los rayos de luz pasan por un único punto, muy pequeño, provocando así una proyección cónica. Aún con las limitaciones descritas, el modelo tiene validez con las actuales cámaras y, de hecho, es el más utilizado.

La Figura 3.1 consta de los siguientes elementos básicos:

- Un sistema de coordenadas 3D cuyo origen es el centro óptico, cuya Z (el eje óptico) apunta en dirección de la cámara.
- El plano imagen donde los puntos 3D del mundo se proyectan en dirección al centro óptico. Es paralelo a los ejes X e Y y está localizado a una distancia f (distancia focal) del centro.
- Un punto en la intersección del eje óptico y el plano imagen, al que se denomina punto principal o centro de la imagen.
- Un punto 3D en el mundo (X,Y,Z) llamado P.
- El rayo de retroproyección del punto 3D en la imagen que es la línea recta que pasa por el punto P y el centro óptico.
- La proyección del punto 3D en el plano imagen, que está determinado por la intersección del rayo de retroproyección con el plano imagen.
- Un sistema de coordenadas 2D en el plano imagen, con los ejes u y v paralelos a X e Y.

En la vida real las lentes de las cámaras no son ideales. Esto supone que las imágenes capturadas por las cámaras no son perfectas y pueden contener deformaciones. Por ejemplo, una cámara puede capturar una imagen achatada, sin embargo nosotros no apreciamos dicha deformación cuando visualizamos la imagen. Normalmente esta deformación es debida a que la lente no está perfectamente alineada con el plano de proyección, lo que provoca deformaciones en la imagen. Gracias a un proceso conocido como calibración de una cámara, podemos conocer, entre otros, la desviación de la lente respecto del plano de proyección en el eje X y en el eje Y. De este modo podemos corregir la deformación producida por el desalineamiento entre la lente y el plano de proyección.



(a) (b)

Figura 3.2: Semejanza de triángulos

El modelo Pin Hole tiene una propiedad geométrica muy interesante, se la conoce como la semejanza de triángulos y gracias a ella podemos calcular los parámetros intrínsecos de nuestra cámara. La Figura 3.2b muestra esta propiedad, en ella podemos apreciar claramente dos triángulos, el formado por el centro de óptico de la cámara (punto C), el punto 3D M y su proyección sobre el eje Z de la cámara. El otro triángulo está compuesto por la proyección de M sobre el plano imagen (punto m), el centro óptico y el punto en el plano imagen que está a una distancia f (distancia focal) del punto C.

La ecuación 3.1 muestra la matriz K, que es la representación matricial de los parámetros intrínsecos calculados gracias al teorema de la semejanza de triángulos. En ella quedan definidos los parámetros f_x y f_y que son la distancia focal en los ejes X e Y, los parámetros u_0 y v_0 que definen el punto del centro (o punto principal) del plano imagen y el parámetro s (*skew*) que es un factor de escala para la distorsión radial en las imágenes.

$$K = \begin{bmatrix} f_x & s & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.1}$$

Esta matriz permite realizar la proyección de un punto 3D en el mundo a un punto 2D en el plano imagen de nuestra cámara, sólamente necesitamos multiplicar dicho punto por la matriz K.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & s & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.2}$$

La localización 3D o pose se define como la posición 3D de la cámara y su orientación respecto cierto sistema de referencia externo, y se suele expresar como una matriz $3x4$ ($4x4$ si trabajamos en coordenadas homogéneas) que es el resultado de la multiplicación entre una matriz de rotación (R) $3x3$ por un vector de traslación (t) $3x1$.

$$RT = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} tx \\ ty \\ tz \end{bmatrix} \tag{3.3}$$

Finalmente si quisiéramos conocer la proyección (P_{im}) de un punto 3D (P_w) en el plano de imagen de nuestra cámara tendríamos que añadir la matriz RT a la ecuación 3.2.

$$P_{im} = K \cdot RT \cdot P_w \tag{3.4}$$

3.1.2. Perspective-n-Point

Un problema muy conocido en visión artificial es el cálculo de la posición 3D de una cámara a partir de un conjunto de puntos 3D y sus proyecciones 2D (ver Figura 3.3). El caso general es un problema con 6 grados de libertad; la pose y 5 parámetros intrínsecos: la distancia focal, el punto principal, el ratio

de aspecto y el *skew*. Existen varias simplificaciones para este problema, una de las más conocidas es asumir que se conocen los parámetros de calibración intrínsecos. A esta simplificación se la conoce como *Perspective-n-Point* (PnP) y fue desarrollada por Fischler and Bolles [19] en 1981.

En función del número de puntos de los que dispongamos existen distintas aproximaciones:

- Dos puntos o menos: número de soluciones infinitas.
- Tres puntos: ocho posibles soluciones de las cuales, la mitad están frente a la cámara (Haralick *et al.*, 1994 [2]).
- Cuatro puntos: si son coplanares solo existe una solución, en otro caso hay dos soluciones (Horaud *et al*, 1989 [51]).
- Cinco puntos o más: podemos encontrar la solución, pero en función del método que utilicemos, la encontraremos con más o menos error. La técnica clásica consiste en utilizar mínimos cuadrados (Hesch and Roumeliotis, 2011 [29]), existen otras técnicas más complejas que pretenden aumentar la eficiencia y precisión, la propuesta por Lepetit *et al.*, 2009 [65] y la desarrollada por Li *et al.* 2012 [60].

En la librería *OpenCV*, mediante las funciones `solvePnP()`[1] y `solvePnPRansac()`[2], podemos hacer uso de algunos de los métodos listados anteriormente.

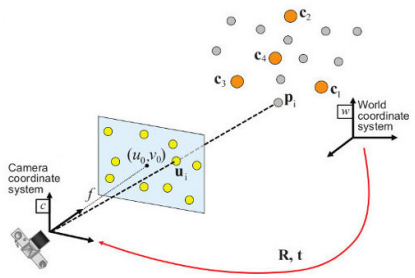


Figura 3.3: Estimación de posición mediante el algoritmo PnP

3.2. Odometría visual

Se conoce como odometría visual al proceso que permite determinar de manera incremental la posición y orientación 3D de una cámara a través de una secuencia de imágenes obtenidas por ésta. El desplazamiento se estima analizando de manera secuencial las observaciones producidas por la cámara. El término "odometría" proviene de la robótica donde se utiliza para referirse a la estimación incremental del desplazamiento de un robot con ruedas donde es necesario conocer la geometría del robot. El término

[1] `http://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#solvepnp`

[2] `http://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#solvepnpransac`

"odometría visual" fue utilizado por Nister (Nister et al. 2004 [11]) y a diferencia de la odometría, que sólo está disponible cuando se conoce la geometría del robot, la visual puede ser utilizada con cualquier tipo de cámara.

Al utilizar imágenes para estimar el desplazamiento de la cámara, la odometría visual es sensible a la iluminación y necesita que las imágenes tengan cierta textura para extraer características de la imagen que permitan calcular el desplazamiento de la cámara. Además, las imágenes con las que se vaya a calcular el desplazamiento tienen que mostrar la misma escena desde distintos puntos de vista.

3.2.1. Odometría con visión monocular

El utilizar una única cámara para realizar una odometría visual es un problema en sí mismo, ya que, la odometría visual necesita varías observaciones de la misma escena desde distintos puntos de vista. Para resolverlo, la odometría visual monocular, toma distintas imágenes de la escena con la misma cámara asimilando así el problema a la visión estéreo. Si, además, se precisa obtener la escala real de la escena, habrá que conocer las medidas de alguno de los objetos que aparezcan en ella (Castle *et al.* 2010 [55]), utilizar otros sensores como referencia (Nützi, 2011 [21]) o hacer uso de sensores RGB-D que proporcionan directamente la profundidad de la escena.

Los algoritmos de odometría visual monocular se pueden clasificar como:

- Basados en características: buscan píxeles característicos en la imagen y realizan un seguimiento ópticos de ellos en cada imagen (Mouragnon *et al.*, 2006 [18] y Tardif *et al.*, 2008 [30]). El movimiento de la cámara se obtiene minimizando el error de reproyección de los puntos emparejados aplicando algoritmos de optimización.

  Al utilizar algoritmos de detección de puntos característicos y de emparejamientos, este método tiene las ventajas y desventajas de estos algoritmos. Existen algoritmos muy robustos capaces de detectar puntos característicos, pero la calidad del emparejamiento de estos puntos determinará la precisión en los métodos basados en características.

- Métodos directos: en lugar de utilizar únicamente la información de determinados píxeles en la imagen (bordes, esquinas, ...) utilizan el valor de intensidad de todos los píxeles (Milford and Wieth, 2008 [45] y Lovegrove *et al.*, 2011 [62]). El hecho de utilizar todos los píxeles de la imagen supone una ventaja cuando se trabaja con imágenes sin textura o borrosas, dado que utilizan toda la información de la imagen. Aunque utilizar todos los píxeles es más lento, estos métodos no necesitan buscar puntos característicos ni emparejarlos.

- Métodos híbridos: son métodos que combinan los dos anteriores (Scaramuzza and R., 2008 [58]).

3.2.2. Odometría con visión estéreo

Los algoritmos de odometría visual con visión estéreo son similares a los vistos en la visión monocular, sólo que ahora tenemos una imagen más. La ventaja de utilizar visión estéreo es la información 3D que proporciona esta configuración. Además de las proyecciones 2D también podemos emparejar puntos 3D,

esto nos da la escala de la escena, y no sólo eso, al utilizar la información espacial podemos medir el desplazamiento de la cámara con unidades métricas, algo que en la visión monocular no es posible ya que sólamente podíamos expresar el desplazamiento en unidades relativas (sin escala).

Este enfoque ha sido utilizado con éxito tanto para la navegación de robots (Matthies and Shaper, 1987 [48] y Olson *et al.*, 2000 [7]) como para algoritmos SLAM (Mei *et al.*, 2009 [6] y Lin and Wang, 2010 [37]).

3.3. Structure From Motion

Se conoce como *Structure From Motion* (SfM) al problema de obtener las posiciones 3D y la estructura de una escena a partir de un conjunto de imágenes (Longuet-Higgins, 1981 [38] y Harris and Pike, 1988 [25]).

Con este enfoque es posible estimar la posición 3D de las cámaras y la estructura de la escena, a partir de imágenes de un mismo lugar obtenidas incluso con cámaras diferentes (McCann, 2015 [44]). La Figura 3.4 muestra el resultado del algoritmo aplicado sobre un conjunto de imágenes del Coliseo de Roma.

Figura 3.4: Reconstrucción del Coliseo de Roma mediante una técnica de SFM

Este problema es más genérico que la odometría visual, ya que las imágenes no tienen porqué ser secuenciales y el resultado final puede ser optimizado mediante algoritmos que requieren un gran tiempo de cómputo, dado que esta técnica no es de tiempo real.

3.4. Técnicas de Visual SLAM

El Visual SLAM (*Simulatenous Localization and Mapping*) es como se denomina al problema de la estimación de la posición y orientación de una cámara mientras realiza, de manera simultanea, en un mapa más o menos denso (depende del algoritmo) del entorno en el contexto típico de la robótica: tiempo real y con un flujo constante de imágenes de la misma cámara. Aunque originalmente esta técnica fue desarrollada con el uso de láseres (Hahnel et al., 2003 [15]) o sonares (Newman et al., 2003 [49]), actualmente existen algoritmos de SLAM que utilizan imágenes como entrada al algoritmo, de ahí que se utilice el término "Visual" cuando hablamos de algoritmos de SLAM que utilizan imágenes como entrada.

3.4.1. MonoSLAM

El término MonoSLAM (*MonocularSLAM*) se refiere a la utilización de una sola cámara RGB la la localización y creación de mapas en entornos desconocidos. Fue propuesto y desarrollado por primera verz por Andrew Davison (Davison, 2002 [13] y Davison, 2003 [14]).

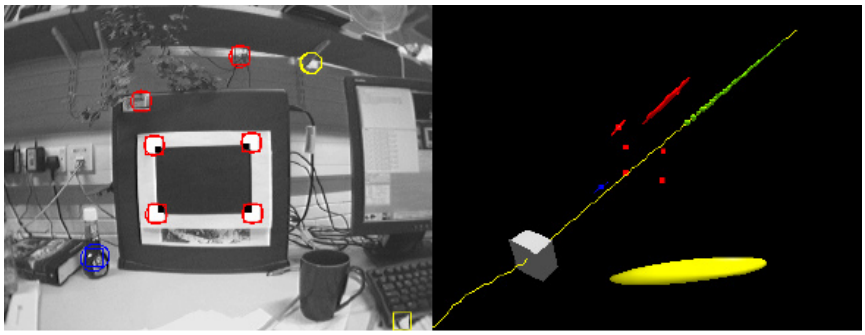


Figura 3.5: Estimación de la posición 3D de una cámara mediante MonoSLAM

Davison propuso el uso de un filtro extendido de Kalman (EKF) para estimar la posición y orientación 3D de la cámara, así como la posición de una serie de puntos 3D que conforman el mapa. Para determinar la posición inicial de la cámara es necesario dotar al filtro de información a priori con la posición 3D de al menos cuatro puntos. A partir de ese momento, el algoritmo es capaz de situar la cámara 3D y de generar nuevos puntos del mapa que sirven como apoyo a la propia localización de la cámara. El algoritmo de MonoSLAM utiliza un mapa probabilístico basado en características, que mantiene el seguimiento de la estimación del estado de la cámara (posición, velocidad angular, velocidad y orientación) y la posición de todas las características. El mapa evoluciona añadiendo nuevas características, eliminando las antiguas y actualizando el estado de la cámara.

3.4.1.1. Filtro de Kalman Extendido

El filtro de Kalman extendido es una extensión del filtro de Kalman para trabajar con modelos no lineales. El EKF aplicado al SLAM estima el estado a partir de una serie de mediciones ruidosas, donde el ruido sigue una distribución Gaussiana. El estado del mundo X, que incluye el estado de la cámara y la posición de las características, se modela como una distribución Gaussiana. En cada iteración del EKF el mapa se representa como un vector x que incluye el estado de la cámara, la posición de n puntos 3D y una matriz de covarianza.

La complejidad del algoritmo es del orden de $O(N^2)$, donde N es el número de características y funciona en tiempo real a $30Hz$ si el número de características a seguir se encuentra alrededor de 100.

3.4.1.2. Inicialización

Antes de comenzar a predecir la posición de la cámara el algoritmo requiere una inicialización antes de comenzar su ejecución. De este modo en la primera iteracion el EKF ya debe conocer la posición de 4 puntos 3D (características) y la posición de la cámara respecto estos puntos. Esto es necesario para conocer

la escala del mapa, además, como estamos utilizando una única cámara, con una única observación, no podríamos conocer la profundidad de las características extraídas por el algoritmo por lo que el mapa no podría ser inicializado al no contener puntos 3D.

El seguimiento de las características se realiza con el algoritmo Shi-Tomasi (Jianbo Shi y Carlos Tomasi, 1994 [59]) y se asume que las características se encuentran en una superficie plana. Esto es debido a que incluso con un pequeño cambio en la posición de la cámara, la apariencia de la característica puede cambiar significativamente.

3.4.1.3. Predicción

Durante la predicción, mientras se captura la nueva imagen y se realiza la etapa de actualización, es posible que la cámara se esté moviendo, pero en este preciso momento el EKF no tiene constancia de ello. Cuando esto ocurre, el estado se actualiza acorde al modelo de movimiento probabilistico que depende sólamente del estado anterior y del movimiento actual que pasa a través de la función de transición de estado del EKF.

En MonoSLAM se espera que el movimiento de la cámara sea suave y a una velocidad constante, esto significa que en cada etapa se asume una aceleración cero y una velocidad lineal y angular constantes.

3.4.1.4. Predicción de características y emparejamiento

Una parte clave del MonoSLAM es la predicción de la posición de las características en la imagen antes de decidir cuál de ellas medir: una vez que el nuevo estado de la cámara ha sido predicho, también se puede predecir qué características conocidas serán visibles y donde se encontrarán en la nueva imagen. En esta etapa también se calcula la ganancia de Kalman, que especifica el grado en que la nueva observación concuerda con la estimación actual del estado. Por último, cuando se realiza la medición de una característica para una nueva posición de cámara, el parche donde se encuentra se proyecta desde 3D al plano de imagen para producir una plantilla para el emparejamiento con la imagen real. Esta plantilla será una versión deformada de la plantilla cuadrada original capturada cuando la característica fue detectada por primera vez.

3.4.1.5. Actualización del mapa

En esta etapa se aplica la ganancia de Kalman y se actualiza la covarianza de la estimación. Ahora el nuevo estado y la nueva covarianza podrán ser usados como la estimación anterior en el próximo ciclo del EKF.

3.4.2. PTAM

PTAM (*Parallel Tracking and Mapping*) es un algoritmo desarrollado en 2007 por Klein y Murray [32] que tenía como objetivo resolver el mismo problema que MonoSLAM, con un enfoque distinto e intentando

resolver alguno de los inconvenientes que tiene el MonoSLAM.

El principal problema de los algoritmos basados en MonoSLAM es que su tiempo de ejecución aumenta exponecialmente con el número de puntos en el mapa. Esto es debido a que en cada iteración se actualiza tanto la posición de cada elemento del mapa (*Mapping*) como la posición actual de la cámara (*Tracking*). Por tanto, aunque el *Tracking* pueda mantenerse en tiempos de ejecución razonables, el *Mapping*, a partir de un cierto número de puntos, impide la ejecución normal del algoritmo en tiempo real.

PTAM parte de la idea de que sólo es necesario funcionar en tiempo real en la parte del *Tracking*, mientras que el *Mapping* no tiene porqué realizarse en cada iteración ni necesita ser tan eficiente. Así, el *Tracking* y el *Mapping* funcionan en dos hilos separados de forma asíncrona.

3.4.2.1. *Tracking*

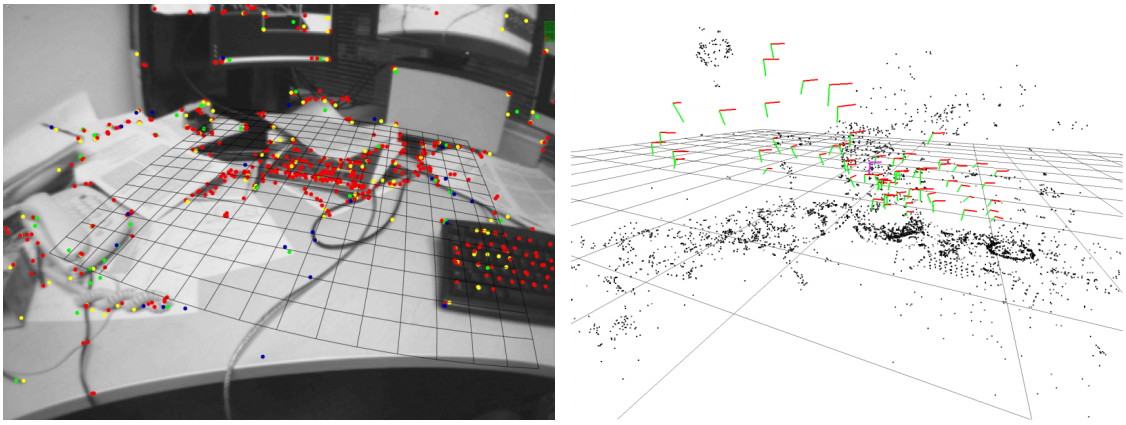
En esta etapa, el algoritmo realiza los siguientes pasos:

1. Preprocesado de la imagen: en este paso se divide la imagen en cuatro subimágenes de distinta resolución (pirámide de la imagen) y se buscan puntos característicos (FAST Rosted y Drummond, 2006 [56]) en cada una de ellas.

2. Actualización de la posición con modelo de movimiento: se actualiza la posición de la cámara aplicando un modelo de velocidad constante, de forma similar a la fase de predicción de MonoSLAM.

3. Actualización de grano grueso: se selecciona un subconjunto de puntos 3D (hasta 60) y se proyectan en la imagen. A continuación se realiza un emparejamiento comparando parches (subimágenes de $8x8$) en un rango amplio alrededor del píxel en el que han proyectado. Con los puntos encontrados se actualiza la posición de la cámara minimizando el error de reproyección aplicando una optimización (Gauss-Newton Kelley, 1999 [31]).

4. Actualización de grano fino: se seleccionan hasta mil puntos del mapa y se sigue el mismo procedimiento que en el paso anterior. En este caso, la posición estimada de la cámara será más precisa y el rango de búsqueda de estos puntos podrá ser menor. De nuevo se realiza una minimización del error de proyección.



(a) (b)

Figura 3.6: Puntos utilizados en el *Tracking* (a) y mapa generado (b) con PTAM

3.4.2.2. *Mapping*

El hilo de *Mapping* obtiene el mapa inicial del entorno mediante una inicialización en estéreo de dos fotogramas que tengan un desplazamiento suficiente. Para ello, el algoritmo realiza un seguimiento en la imagen de una serie de puntos característicos, a partir de los cuales calcula el mapa inicial mediante el algoritmo de cinco puntos (Stewenius *et al.*, 2006 [27]). Los puntos iniciales calculados, así como la posición de la cámara, pueden no tener una posición en 3D totalmente eral, pero representarán a la realidad en una escala diferente.

Una vez concluida la inicialización, el hilo de *Mapping* realiza los siguientes pasos:

1. Añadir nuevos *Keyframes*: los nuevos fotogramas que serán añadidos al mapa deben cumplir una serie de requisitos: (1) que la distancia al otro *Keyframe* previamente almacenado sea suficiente, (2) que hayan pasado suficientes iteraciones desde que se almacenó el último *Keyframe* y (3) que la localización actual sea de buena calidad.

2. Añadir nuevos puntos 3D al mapa: cada vez que se añade un nuevo *Keyframe* al mapa se seleccionan puntos característicos en la imagen y se buscan en los *Keyframes* anteriores. Los emparejamientos obtenidos se añaden al mapa calculando su posición 3D mediante triangulación. Esos puntos son los que utiliza el hilo de *Tracking* para realizar la estimación de la posición de la cámara.

3. Optimizar el mapa: siempre que el hilo del *Mapping* se encuentre ocioso se refina la posición 3D tanto de los *Keyframes* como de los puntos almacenados. Así, se utiliza el algoritmo de optimización *Bundle Adjustment* (Triggs *et al.*, 1999 [3], Hartley y Zisserman, 2003 [26]) cuyo objetivo es minimizar el error de reproyección entre m posiciones en 3D de la cámara y n puntos 3D.

4. Mantenimiento del mapa: se comprueba la coherencia de los datos entre los distintos *Keyframes*, eliminando aquellas asociaciones de puntos mal realizadas y añadiendo nuevos puntos cuando sea posible.

El mapa final obtenido (Figura 3.6b) puede contener miles de puntos que se relacionan entre sí en múltiples *Keyframes* y que, por tanto, guardan coherencia espacial entre ellos, evitando así puntos mal situados debido a localizaciones puntuales erróneas.

3.4.3. DTAM

DTAM (*Dense Tracking And Mapping*) propuesto por Richard A. Newcombe *et al.* en el 2011 [54] es un algoritmo de SLAM monocular de método directo que, en lugar de utilizar ciertas características de la imagen, hace uso de todos los píxeles de ésta. Al utilizar toda la información de las imágenes se consigue un mapa denso, desde el cual se estima la posición 3D de la cámara con una mayor precisión que con los métodos basados en características.

Trabajar con un mapa denso requiere una mayor potencia de cálculo pero, según los autores, DTAM es capaz de trabajar en tiempo real en equipos potentes debido a que el algoritmo es paralelizable y permite su ejecución en GPU.

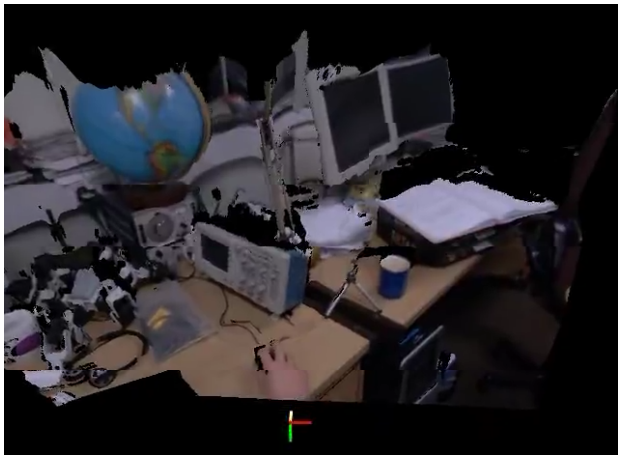


Figura 3.7: Mapa denso generado por DTAM

3.4.3.1. *Tracking*

En este caso el *Tracking* (o *Dense Tracking* que denominan los autores) de la cámara se realiza mediante el alineamiento de *Keyframes* consecutivos aplicando el método propuesto por Steven Lovegrove y Andrew J. Davison [39]. La primera estimación de la pose se refina de un modo similar PTAM con la actualización de grano grueso y fino, solo que en DTAM se minimiza el error fotométrico y no el error de reproyección.

3.4.3.2. *Mapping*

La generación del mapa en este método se realiza mediante la reconstrucción con múltiples vistas. Al ser un algoritmo monocular implica que en la primera fase de ejecución del algoritmo es necesario obtener distintas vistas del mismo objeto. El proceso de reconstrucción se realiza con todos los fotogramas de entrada, y mediante un proceso que pretende reducir la energía de la imagen (definida como la suma del

error fotométrico) se refina la generación del mapa. En este proceso también se decide qué *Keyframes* serán utilizados para la estimación de la pose 3D de la cámara.

3.4.4. SVO

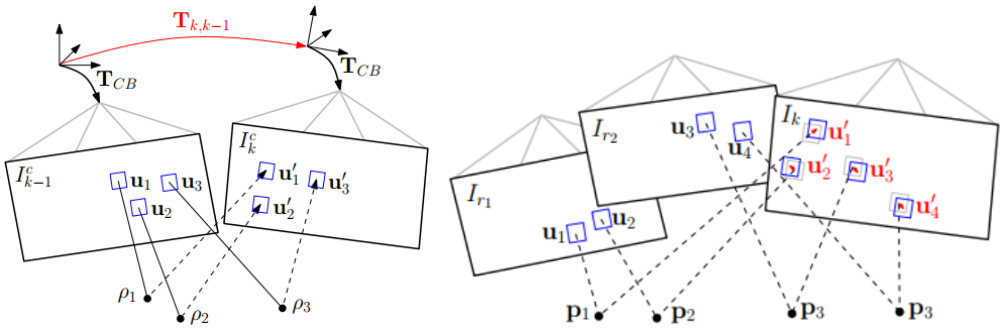
Otro de los algoritmos de SLAM que han destacado en los últimos años ha sido SVO (*Fast Semi-Direct Monocular Visual Odometry*, Forster *et al.*, 2014 [5]). Este algoritmo destaca por su rapidez, pudiendo ser utilizado en tiempo real en ordenadores con poca capacidad de cómputo.

SVO utiliza un método híbrido de los dos métodos directos y los basados en características, lo que reduce considerablemente el tiempo de cómputo del algoritmo. Tiene una estructura similar a PTAM, separando en dos hilos el cálculo del desplazamiento de la cámara (*Tracking*) y la generación del mapa (*Mapping*).

3.4.4.1. *Tracking*

El hilo encargado del seguimiento de la cámara es el que implementa la técnica híbrida. El primer paso para calcular la posición de la cámara consiste en minimizar el error fotométrico entre el fotograma actual y el anterior. Si se utilizase toda la imagen para realizar este cálculo, se necesitaría mucho tiempo de cómputo; por ello, el SVO sólo minimiza el error fotométrico de ciertas partes de la imagen. Estas partes se corresponden con parches de $4x4$ alrededor de los píxeles en los que proyectan los puntos 3D visibles del fotograma anterior (Figura 3.8a).

Posteriormente se ajusta la posición de cada punto 3D por separado teniendo en cuenta el desplazamiento estimado en el paso anterior, minimizando el error fotométrico de cada punto usando parches algo más grandes ($8x8$) que en el alineamiento general (Figura 3.8b). Esta técnica permite obtener el desplazamiento entre imágenes de una forma muy eficiente, con tiempos de cómputo de pocos milisegundos que permiten utilizar el algoritmo a más de 60 FPS.



(a) (b)

Figura 3.8: Cálculo de la posición de la cámara con el algoritmo SVO: Minimización del error fotométrico (a) y ajuste por separado de cada punto 3D (b)

3.4.4.2. *Mapping*

Al igual que PTAM, SVO selecciona alguno de los fotogramas que considera clave (*Keyframes*) y basa la generación del mapa en 3D en esos *Keyframes*. Inicialmente, los primeros puntos del mapa son calculados mediante homografía al igual que en PTAM. A partir de ahí, los nuevos puntos no se añaden directamente al mapa triangulado entre dos *Keyframes* (como PTAM), sino que siguen un proceso más complejo. Cuando se observa un nuevo punto desde dos *Keyframes*, se calcula la profundidad a la que se encuentra el punto mediante una distribución de probabilidad como la que se ve en la Figura 3.9, donde la incertidumbre de la profundidad es inicialmente muy alta. A medida que se obtienen nuevas observaciones del punto 3D se actualiza la distribución de probabilidad hasta que la varianza de la distribución es suficientemente pequeña, momento en el cual se valida el punto 3D y se añade al mapa.

La ventaja de este método es que los puntos que se añaden al mapa han sido observados desde varios fotogramas y, por tanto, su posición 3D es más precisa que en PTAM, obteniendo un mapa con menos puntos pero más fiable.

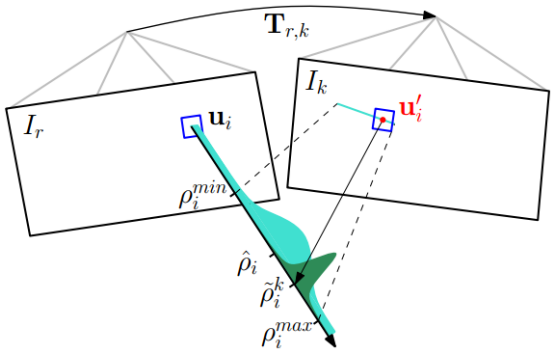


Figura 3.9: Inicialización de puntos 3D con estimación probabilística de profundidad en SVO

3.4.5. ORB-SLAM

El algoritmo ORB-SLAM (Mur-Artal et al., 2015 [53], Mur-Artal y Tardos, 2016 [47]) es un algoritmo de SLAM basado en píxeles característicos que puede funcionar con una sola cámara, con cámaras estéreo o con cámaras de profundidad RGB-D. Su característica principal es que utiliza descriptores ORB (Rublee *et al.*, 2011 [17]) para extraer y emparejar píxeles característicos, así como un modelo de bolsa de palabras (Gálvez-López y Tardos, 2012 [24]) para detectar cierres de bucles y, en su caso, para relocalizarse.

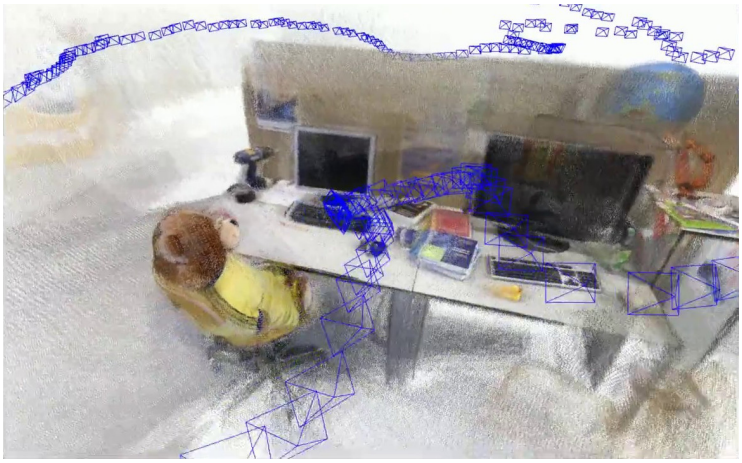


Figura 3.10: Mapa generado con ORB-SLAM

ORB-SLAM está dividido en tres hilos de ejecución. Además de los hilos de *Mapping* y *Tracking* de los que disponen el resto de algoritmos, añade un hilo más para detectar cierres de bucle. Utilizando estos tres hilos, el algoritmo puede funcionar en tiempo real en ordenadores con alta capacidad de cómputo sin necesitad de intervención de GPU.

3.4.5.1. *Tracking*

Para realizar el *Tracking* se aplica un modelo de movimiento y se tratan de emparejar los puntos 3D visibles en el fotograma anterior, calculando la posición actual a partir de los emparejamientos realizados. La diferencia principal de este algoritmo respecto a los anteriores es que los emparejamientos se realizan utilizando descriptores ORB.

En caso de que el robot se pierda, se utiliza un modelo de bolsa de palabras para obtener *Keyframes* candidatos que concuerden con la observación actual. En estos candidatos se buscan correspondencias con los descriptores ORB disponibles y se calcula la posición de la cámara mediante el algoritmo PnP.

3.4.5.2. *Mapping*

La inicialización del mapa se realiza calculando en paralelo un mapa por homografía y otro mediante matriz fundamental. Ambos modelos obtienen una puntuación para determinar cuál será el utilizado para inicializar el mapa, de forma que en aquellas escenas que cuenten con un plano principal se utilizará la homografía y en el resto de casos se utilizará la matriz fundamenteal. Esto evita que el algoritmo asuma siempre que se encuentra ante escenas que contengan un plano, como hacen el resto de algoritmos.

Una vez obtenido el mapa inicial, el hilo de *Mapping* se encarga de procesar los *Keyframes* creando nuevos puntos 3D y de optimizar el mapa localmente mediante *Bundle Adjustment*. Los nuevos puntos 3D se generan mediante triangulación emparejando descriptores ORB.

Otra característica de este algoritmo es que genera un grafo donde los vértices se corresponden con *Keyframes* y las aristas entre vértices se generan siempre que los *Keyframes* tengan varios puntos 3D en común. Este grafo se utiliza entre otras cosas para eliminar *Keyframes* redundantes y evitar así que el número de *Keyframes* crezca innecesariamente.

3.4.5.3. *Looping*

El hilo de *Looping* se encarga de comprobar si se ha producido un cierre de bucle. Para ello, utiliza el grafo de *Keyframes* conectados y el modelo de bolsa de palabras, con el objetivo de buscar *Keyframes* candidatos con apariencia similar a la observación actual. Si se detecta un cierre de bucle, se eliminan los puntos 3D duplicados y se corrige la posición de los *Keyframes* para alinear los extremos del bucle.

3.5. Generación de mapas 3D

En esta sección se describen dos métodos fundamentales para este Trabajo Fin de Máster en cuanto a la generación de mapas 3D se refiere. Los métodos de SLAM generan sus propios mapas para poder estimar la posición 3D de la cámara, sin embargo suelen ser mapas creados para el apoyo del propio algoritmo y no tanto para su uso posterior. Los algoritmos que se presentan en esta sección están pensados e implementados para la generación de mapas 3D fidedignos, dejando en un segundo lugar la estimación de la pose de la cámara.

3.5.1. GraphSLAM

El *GraphSLAM* (F. Lu y E. Milios, 1997 [41]) es un algoritmo de SLAM proveniente de la robótica que es utilizado para la generación de mapas. Es un enfoque totalmente distinto a los expuestos anteriormente, es un algoritmo *offline* que plantea el problema del SLAM como un grafo disperso de restricciones. El algoritmo extrae de los datos restricciones blandas representadas por un grafo disperso. Crea el mapa y la ruta seguida por el robot resolviendo estas restricciones de manera global. Las restricciones normalmente son no lineales, pero durante el proceso son linearizadas, resultando así en un problema de mínimos cuadrados que es resuelto con algoritmos de optimización.

El *GraphSLAM* puede manejar un gran número de características (10^8) por lo que lo hace ideal para escenarios muy grandes como la reconstrucción de estructuras urbanas (S. Thrun y Michael Montemerlo, 2005 [63]).

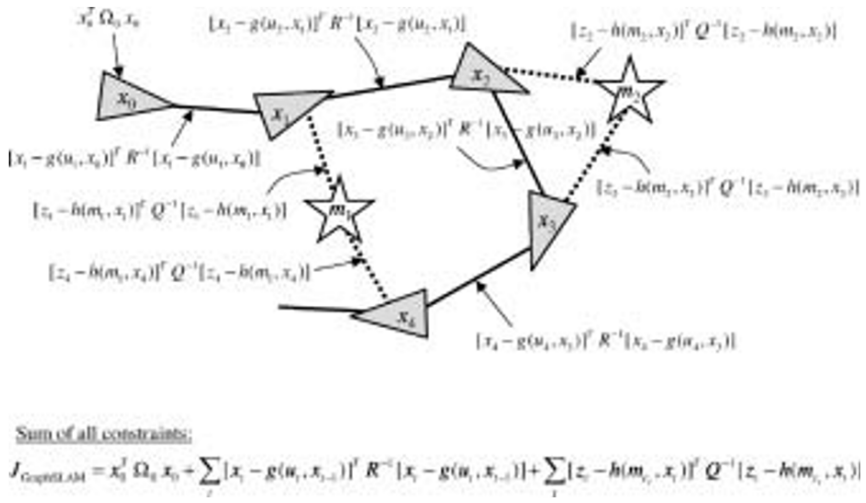


Figura 3.11: Grafo generado en el algoritmo *GraphSLAM*

En la Figura 3.11 se puede apreciar el funcionamiento del algoritmo. En ella se muestra el grafo que *GraphSLAM* extrae a partir de cuatro poses marcadas como $x_1, ..., x_4$ y dos características del mapa m_1, m_2. El grafo tiene dos tipos de arcos; arcos de movimiento y arcos de mediciones. Los arcos de movimiento enlazan dos posiciones consecutivas del robot y los arcos de mediciones, enlazan las poses del robot con las características del mapa que fueron obtenidas desde dicha posición. Cada arista del grafo representa una restricción no lineal. Cada restricción es modelada como el logaritmo negativo de la verosimilitud de las mediciones y el modelo de movimiento del robot. La suma de todas las restricciones resulta en un problema no lineal de mínimos cuadrados. *GraphSLAM* lineariza estas restricciones, resultando en una matriz dispersa y un vector. Aplicando el algoritmo de eliminación de variables (N.L. Zhang and D. Poole, 1994 [69], Adnan Darwiche, 2009 [10] y D. Koller y D. Friedman, 2009 [33]) el grafo se reduce a uno más pequeño definido unicamente por las poses del robot. Finalmente mediante técnicas de inferencia se extrae la ruta que siguió el robot en el mapa.

3.5.2. RTAB-Map

RTAB-Map (*Real-Time Appearance-Based Mapping*) es la herramienta de generación de mapas 3D que implementa un algoritmo de GraphSLAM con sistema de bucle cerrado global propuesto por Mathieu Labbé y François Michaud en 2014 [36] para sensores RGB-D. A pesar de que el GraphSLAM es un algoritmo *offline* gracias a un enfoque para la gestión de memoria en mapas grandes, propuesto por los mismos autores (Mathieu Labbé y François Michaud , 2011 [34]), han conseguido que este algoritmo funcione en tiempo real. El segundo aporte es la posibilidad de crear mapas en distintas sesiones, pudiendo así generar mapas muy grandes.

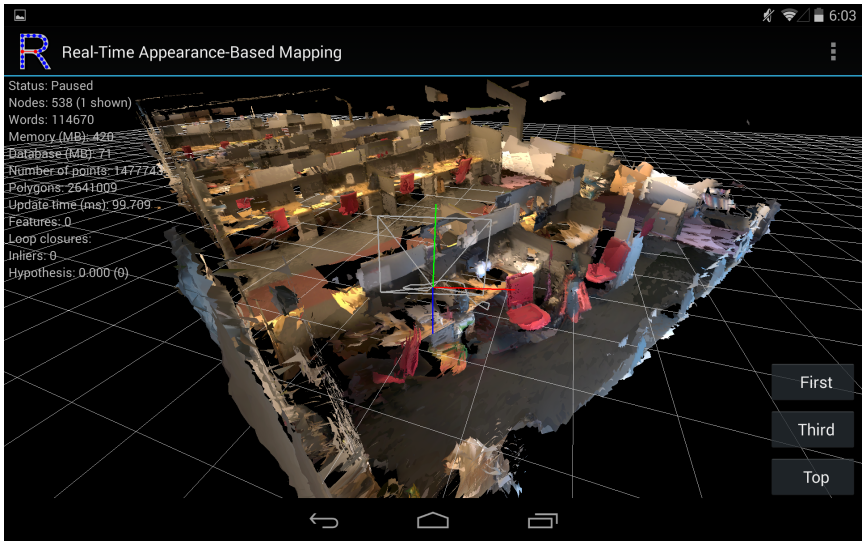


Figura 3.12: Mapa generado con RTAB-Map en su versión de *Google Tango*

En el grafo generado por este algoritmo los nodos almacenan información sobre la odometría para cada localización del mapa, además de contener más información como imágenes RGB, de profundidad o lo que los autores denominan como *visual words* (Gálvez-López y Juan. D. Tardos, 2012 [24]). Los arcos almacenan transformaciones geométricas rígidas entre los nodos. Existen dos tipos de arcos; los arcos vecinos que son añadidos entre dos nodos consecutivos y los arcos de cierre de bucle que se añaden cuando se detecta un cierre de bucle (cuando el sistema detecta que la cámara está pasando por un lugar previamente visitado).

3.5.2.1. Sistema de detección de bucle cerrado

Para el sistema de cierre de bucle global se utiliza un enfoque de bolsa de palabras (*bag-of-words*) propuesto por J.Sivic y A. Zisserman, 2003 [61]. Este mecanismo usa un filtro bayesiano para evaluar la hipótesis de bucle cerrado sobre todas las imágenes previas. Cuando se detecta un cierre de bucle, las *visual words* (que son características SURF[28] almacenadas) se utilizan para calcular la verosimilitud requerida por el filtro.

Para estimar la posición 3D de la cámara, se extrae una imagen RGB de las *visual words*, se registra con su correspondiente imagen de profundidad. Por cada punto 2D en la imagen, se calcula su punto 3D usando la matriz de calibración (K) y la información de profundidad de la imagen obtenida por el sensor RGB-D. De este modo se conoce la posición de la *visual word*. Cuando se detecta un cierre de bucle se calcula la transformación geométrica rígida entre las imágenes mediante RANSAC[19] usando las correspondencias 3D de las *visual words*. Si se encuentra un número mínimo de *inliers* el nuevo fotograma se acepta como cierre de bucle y se crea un nuevo arco con la transformación geométrica calculada.

3.5.2.2. Optimización del grafo

Para la optimización del grafo se usa el algoritmo TORO (*Tree-based netwORk Optimizer*) propuesto por G. Grisetti *et al.* [20]. En este algoritmo los nodos y los arcos se usan como restricciones. Cuando se

detecta un cierre de bucle es posible que se introduzcan errores de odometría y propaguen por los arcos. TORO se encarga de minimizar estos errores manteniendo la coherencia del grafo.

3.5.2.3. Sistema de gestión de memoria para la creación de mapas en múltiples sesiones

De los puntos anteriores se desprenden dos problemas: el tiempo necesario para la detección del cierre de bucle y el tiempo necesario para la optimización del grafo. A medida que el mapa crece estos tiempos comienzan a crecer de manera que hacen inviable la generación en tiempo real del mapa. Para resolver estos problemas se introdujo un sistema para la gestión de memoria que permite mantener el mapa en un tamaño suficiente para funcionar en tiempo real.

El sistema está compuesto por una *Short-Term Memory* (STM), una *Working Memory*(WM) y una *Long-Term Memory* (LTM). La STM, que tiene un tamaño fijo, es el punto de entrada para los nuevos nodos que se añaden grafo. Cuando se alcanza el tamaño máximo en la STM el nodo más viejo se mueve a la WM. Los nodos en la STM, al añadirse nuevos nodos cada vez que se procesan las imágenes, no son considerados para la detección de bucle cerrado pero los nodos que se mueven a la WM sí son tenidos en cuenta para dicha detección. El tamaño de la WM depende del tiempo, transcurrido un tiempo T los nodos de la WM se mueven a la LTM manteniendo un tamaño de WM constante. Los nodos de la LTM no se utilizan para la detección de bucle ni para la optimización del grafo, sin embargo, cuando se detecta el cierre de bucle se consulta al LTM para buscar vecinos del nodo que ha sido seleccionado para el cierre de bucle.

Este enfoque permite trabajar con un tamaño de grafo fijo independientemente del tamaño del mapa que se esté generando, consiguiendo así que al algoritmo de GraphSLAM funcione como un algoritmo de tiempo real.

Capítulo 4

Infraestructura

En este capítulo se describen los elementos hardware y software empleados durante la realización del Trabajo Fin de Máster que han sido de ayuda para la consecución del proyecto.

4.1. Sensor RGB-D

Dada la naturaleza del proyecto fue necesario contar con un sensor RGB-D que capturase tanto la información visual del mundo (imágenes RGB) como la información de distancia (imágenes de profundidad).

Figura 4.1: Kinect v1

La Figura 4.1 muestra el sensor RGB-D utilizado en el proyecto. Es el sensor Kinect en su versión 1. Desarrollado por *Microsoft* gracias a la licencia de *PrimeSense* fue lanzado al mercado en 2010 como un revolucionario complemento para la videoconsola Xbox 360. Este dispositivo contiene un emisor de rayos infrarrojos, una cámara RGB, un sensor de profundidad basado en infrarrojos, micrófonos y un motor que permite mover la cámara. Las especificaciones para este modelo son:

Ángulo de vista	43° vertical por 57° horizontal
Rango de inclinación vertical	$\pm°27$
Velocidad	30 fotogramas por segundo tanto para imágenes de color como de profundidad
Resolución	$320x240$, $640x480$ y $80x60$ todas ellas a 30FPS
Formato de imagen	Bayer, BGRA, RGBA, Yuv y Yuy2
Audio	$16-kHz$, $24-bit$ mono PCM
Encoders	de 2G/4G/8G configurado para el rango 2G con 1° de precisión
Conexión	Conector propietario de *Microsoft*, es necesario disponer de un adaptor a USB

4.2. Entorno JdeRobot

Este Trabajo Fin de Máster ha sido desarrollado empleando el entorno de programación multiplataforma de software libre JdeRobot[1] para robótica y visión artificial mantenido y mejorado a lo largo de los años por el Laboratorio de Robótica de la Universidad Rey Juan Carlos. La filosofía de esta plataforma es crear un ecosistema de *drivers*, bibliotecas y herramientas que facilite la programación de aplicaciones de robótica y visión artificial.

JdeRobot está programado fundamentalmente en C/C++, aunque existen componentes escritos en otros lenguajes de programación como Python o Java. Cada aplicación se organiza como un conjunto de componentes comunicándose con el esquema RPC cliente servidor (o publicación suscripción), donde cada componente se ejecuta como un proceso. Ofrece una interfaz sencilla para la programación de sistemas de tiempo real y resuelve problemas relacionados con la sincronización de los procesos y la adquisición de datos. JdeRobot simplifica el acceso a los dispositivos hardware desde el programa, permitiendo obtener el valor de un sensor únicamente leyendo una variable local incluso aunque el sensor esté en otra máquina. Ofrece un amplio repertorio de *drivers*. En el caso de las cámaras, ofrece la misma interfaz de imagen para distintas fuentes de vídeo, lo cual proporciona una gran flexibilidad y portabilidad a las aplicaciones. Todos los componentes se comunican a través de interfaces explícitos ICE o mensajes ROS.

JdeRobot emplea ICE, un *middleware* de comunicaciones desarrollado por la empresa *ZeroC*[2] orientado a objetos distribuidos bajo una doble licencia (GNU GPL y una licencia propietaria). Se utiliza para aplicaciones distribuidas de Internet sin la necesidad de utilizar los protocolos HTTP y es capaz de atravesar cortafuegos, a diferencia de la mayoría de *middleware* de este tipo. Es compatible con diferentes lenguajes de programación como C++, Java, Python y la mayoría de sistemas operativos como Windows, Mac OS, Linux y Solaris.

Las aplicaciones creadas para el presente trabajo están desarrolladas sobre la pataforma JdeRobot 5.5.

4.2.1. openniServer

Dentro del entorno JdeRobot se hizo un gran uso del componente `openniServer`[3]. Se trata de un *driver* para la obtención de las imágenes RGB y de profundidad capturadas por la Kinect. Desarrollado por Francisco Miguel Rivas el componente hace uso de la librería `OpenNI`[4] (empresa adquirida por *Apple* en 2013) para el acceso al hardware del dispositivo.

El componente oferta a través de la interface ICE `rgbd`[5] la imagen RGB y de profundidad obtenida del sensor RGB-D. En el código de la interface que se encuentra más abajo, se puede apreciar como a través del método `getData()` se puede obtener un objeto `rgbData` que contiene la imagen de color y de profundidad capturada por un sensor RGB-D.

[1] `https://github.com/JdeRobot/JdeRobot`
[2] `https://zeroc.com/`
[3] `https://github.com/JdeRobot/JdeRobot/tree/master/src/drivers/openniServer`
[4] `https://github.com/OpenNI/OpenNI`
[5] `https://github.com/JdeRobot/JdeRobot/blob/master/src/interfaces/slice/jderobot/rgbd.ice`

```
        struct rgbData {
                ImageData color;
                ImageData depth;
        };


        interface rgbd{
                idempotent rgbData getData();

        };
};
```

4.3. RTAB-Map

RTAB-Map[6] (Real-Time Appearance-Based Mapping) es un algoritmo de SLAM [36] para sensores RGB-D basado en grafos que ha sido utilizado en este proyecto y del cual se hablará con más detalle en el capítulo estado del arte (ver capítulo 3). Además del algoritmo, los autores pusieron a disposición de la comunidad una herramienta con el mismo nombre del algoritmo para la generación de mapas. Es una herramienta desarrollada en C++ de software libre que soporta el uso de distintos sensores RGB-D así como cámaras estéreo. La herramienta está disponible como un módulo de `ROS`[7], como aplicación individual, como librería e incluso ha sido portada a Android para su uso con *Google Tango*.

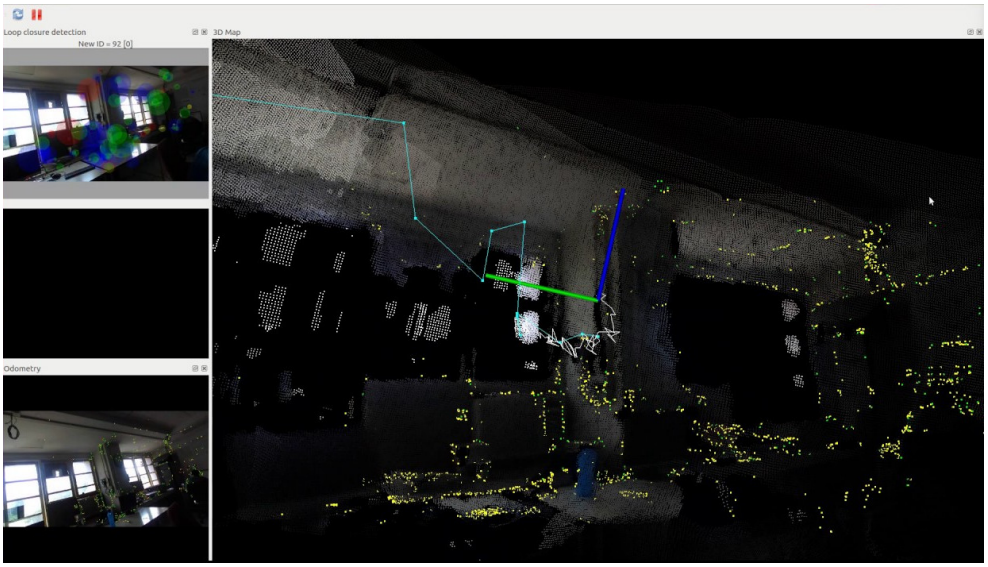


Figura 4.2: Aplicación *RTAB-Map*

Dispone de una interface de usuario muy simple e intuitiva que permite de manera muy cómoda la generación de mapas 3D ofreciendo en todo momento información sobre la calidad de las imágenes obtenidas

[6] `https://introlab.github.io/rtabmap/`
[7] `http://wiki.ros.org/rtabmap_ros`

y su impacto sobre el mapa. Permite la generación de mapas en varias sesiones, así como la fusión de varios mapas. Toda la información es almacenada en una base de datos (*sqllite*) para su posterior procesamiento. Además de la herramienta principal, se proporcionan una gran variedad de utilidades que complementan perfectamente la herramienta principal. Gracias a su arquitectura software, el algoritmo de generación de mapas está totalmente desacoplado de la interface gráfica de usuario y del resto de herramientas, por lo que gracias al API propocionado es posible su uso como una librería en otros proyectos.

Durante el desarrollo del proyecto se utilizó la versión 0.12.5, su uso ha sido clave para el desarrollo de este Trabajo Fin de Máster ya que la estimación de la pose 3D de las cámaras se basa en los mapas 3D generados por dicha herramienta.

4.4. Biblioteca OpenCV

`OpenCV`[8] es una librería de visión artificial desarrollada inicialmente por *Intel* y mantenida actualmente por *Itseez*. Es una librería de código abierto publicada con licencia BSD, lo que permite su libre utilización en propósitos comerciales o de investigación. Existe una gran comunidad alrededor de ella, y de hecho, es considerada como un standard de facto en el desarrollo de aplicaciones de visión artificial.

Figura 4.3: Reconocimiento facial con *OpenCV*

Esta librería proporciona un extenso conjunto de funciones para trabajar con visión y su desarrollo se ha realizado primando la eficiencia. Se han programado utilizando C y C++ optimizados, pudiendo además hacer uso de primitivas de rendimiento integradas en los procesadores *Intel*, que son un conjunto de rutinas de bajo nivel específicas en estos procesadores y que ofrecen un gran rendimiento computacional. Contiene más de 500 funciones que abarcan una gran gama de áreas en el procesamiento de visión (e incluso de la inteligencia artificial), como reconocimiento de objetos (reconocimiento facial), calibración de cámaras, visión estéreo, descriptores visuales o visión robótica. Tiene interfaces para C, C++, Python

[8] `http://opencv.org/`

y Java y soporta la mayoría de sistemas operativos como Windows, Linux, Mac OS, iOS y Android. La documentación detallada de esta librería se puede encontrar en Internet[9].

La versión utilizada en este trabajo ha sido `OpenCV` 3.1.0. Para cuestiones como la transformación del espacio de color de imágenes, la calibración de cámaras RGB, la extracción de características de la imagen y el emparejamiento de descriptores de imagen.

4.5. Biblioteca cvSBA

`cvSBA`[10] es una librería desarrollada por el grupo de Aplicaciones de la Visión Artificial[11] (AVA) de la Universidad de Córdoba[12]. Se trata de un envoltorio para `OpenCV` de la librería de Sparse Bundle Adjustment basado en el algoritmo Levenberg-Marquardt `sba`[13] desarrollada por Manolis Lourakis[14].

En el proyecto se usó la versión 1.0 con algunos errores de compilación resueltos por Di Zeng [15]. Concretamente se utilizó para realizar una optimización final a la estimación de la pose 3D de la cámara.

4.6. Biblioteca PCL

`PCL`[16] (Point Cloud Library) es una librería de software libre de algoritmos para el procesamiento de nubes de puntos y geometría 3D. La librería contiene algoritmos para la estimación de características, reconstrucción de superficies, registro, relleno de modelos y segmentación, además de un visor 3D. Está desarrollada en C++ bajo una licencia BSD. Su desarrollo comenzó en marzo de 2010 en *Willow Warage*[17]. Es una librería multi-plataforma que funciona en la mayoría de sistemas operativos tal como Windows, Mac Os, Linux, Android e iOS.

[9] `http://docs.opencv.org/3.1.0/`
[10] `https://www.uco.es/investiga/grupos/ava/node/39`
[11] `https://www.uco.es/investiga/grupos/ava/`
[12] `http://www.uco.es/`
[13] `http://users.ics.forth.gr/~lourakis/sba/`
[14] `http://users.ics.forth.gr/~lourakis/`
[15] `https://github.com/willdzeng/cvsba`
[16] `http://pointclouds.org/`
[17] `http://www.willowgarage.com/`

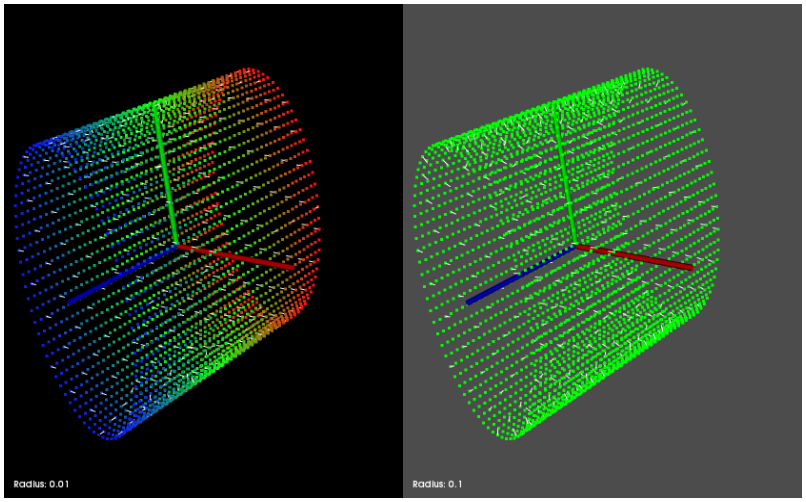


Figura 4.4: Visor 3D de *PCL*

En el proyecto se usó la versión 1.8 de la librería para la visualización de los mapas generados en 3D con la herramienta `RTAB-Map` y de las poses estimadas por el algoritmo implementado en este Trabajo Fin de Máster.

4.7. Biblioteca VTK

`VTK`[18] (Visualization Toolkit) es una librería de software libre para gráficos 3D, procesamiento de imagen y visualización propiedad de la compañía *Kitware*[19]. Está desarrollada en C++ pero posee interfaces para ser utilizada en los lenguajes Tcl/Tk, Java y Python. Soporta una gran variedad de algoritmos de visualización incluidos métodos escalares, vectoriales, de textura y volumétricos. Es una librería multiplataforma disponible para sistemas Windows, Linux, Mac OS y plataformas Unix.

En el proyecto se usó la versión 7.0 de la librería, se utilizó para dar soporte gráfico al visor 3D de `PCL`.

4.8. Biblioteca Qt

Qt[20] es una biblioteca multiplataforma para el desarrollo de interfaces gráficas de usuario, así como para el desarrollo de programas sin interfaz gráfica, proporcionando al desarrollador un conjunto muy amplio de módulos, tipos de datos, manejo de conexiones de red, lectura de ficheros JSON, acceso a base de datos, etc. Qt es software libre a través de *Qt project*, donde participa tanto la comunidad libre como desarrolladores de *Nokia*, *Digia* y otras empresas. Qt es distribuida bajo los términos de la GNU Lesser General Public License. Qt es utilizada en KDE, entorno de escritorio para sistemas como GNU/Linux o FreeBSD, entre otros. Qt está desarrollada en C++ de forma nativa, adicionalmente puede ser utilizada en otros lenguajes de programación como Python a través de *bindings*. Funciona en todas las principales plataformas, y tiene un amplio apoyo de la comunidad.

[18] `http://www.vtk.org/`
[19] `https://www.kitware.com/`
[20] `https://www.qt.io/`

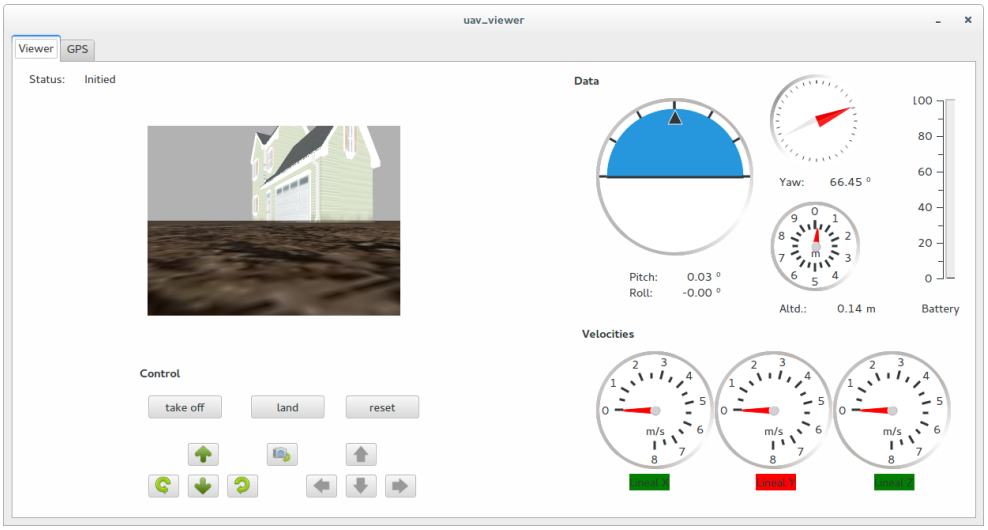


Figura 4.5: Aplicación gráfica desarrollada con *Qt*

En el proyecto se usó la versión 5.0 para el desarrollo de la interface gráfica de usuario de la herramienta desarrollada para la depuración del algoritmo de localización.

4.9. Biblioteca boost

`boost`[21] es una librería que provee de varios módulos de ayuda al programador desarrollada en C++. Es una librería completamente compatible con la librería estándar de C++ `std`. Tiene una licencia Boost que permite el uso comercial de la librería. La librería está incluida en el C++ Standards Comittee's Library Technical Report (TR1) y en el nuevo estándar de C++11. Es multi-plataforma y tiene soporte para la mayoría de sistemas operativos como Windows, Linux y sistemas Unix, es una librería ampliamente utilizada por la comunidad.

En el proyecto se usó la versión 1.58, la librería se utilizó para la gestión de los argumentos de entrada en las diferentes herramientas, la comunicación con el sistema operativo y la gestión de hebras.

4.10. Eigen

`Eigen`[22] es una librería de algebra lineal desarrollada en C++. Es una librería de código abierto licenciada bajo MPL2 desde la versión 3.1.1, las versiones anteriores están licenciadas bajo LGPL3+. La librería permite trabajar eficientemente con matrices, vectores y ofrece métodos de resolución y reducción de sistemas lineales entre otros.

En el proyecto se usó la versión 3.1.2 y se utilizó para la realización de cálculos matriciales.

[21] `http://www.boost.org/`
[22] `http://eigen.tuxfamily.org/index.php?title=Main_Page`

Capítulo 5

Autolocalización visual 3D

Este capítulo trata sobre el algoritmo de autolocalización desarrollado para el Trabajo Fin de Máster, así como de librería que lo implementa y las herramientas desarrolladas. En la sección 5.1 se explica cómo se construye el mapa que espera recibir el algoritmo. La sección 5.4 habla del componente creado para autolocalizar en 3D la cámara, `localizer`, que integra la biblioteca en la que se ha materializado el algoritmo con la captura de imágenes en tiempo real y ofrece la estimación continua de posición a otros componentes JdeRobot. La sección 5.5 describe la herramienta creada, `autoloc_viewer`, que visualiza en 3D y en tiempo real la estimación de posición entregada por el componente `localizer`.

5.1. Generación del mapa 3D

El objetivo del Trabajo Fin de Máster es localizar una cámara dentro de un mapa 3D previamente generado. Concretamente, se quiere obtener la matriz RT, que junto con los parámetros intrínsecos de la cámara proporcionados al algoritmo, nos permitirán calibrar un sensor RGB-D dentro de su entorno de trabajo. Por tanto, en primer lugar necesitamos un mapa 3D del entorno de trabajo de la cámara.

Para la generación del mapa se ha utilizado la herramienta `RTAB-Map`. Se decidió utilizar esta herramienta; porque el algoritmo de generación de mapas que implementa funciona con sensores RGB-D, permite la generación de mapas grandes y realizados en varias sesiones, genera unos mapas de muy buena calidad y además es software libre. Es una herramienta reconocida por la comunidad, que en 2014 ganó el *IROS 2014 Kinect Challenge*[1].

A pesar de la gran calidad de los mapas generados por `RTAB-Map` para localizar nuestra cámara dentro del mapa no necesitamos tanta información. Por este motivo se decidió que para nuestro propósito simplemente eran necesarios aquellos *Keyframes* que el algoritmo de detección de bucle cerrado [36] que implementa la herramienta marcase como *Keyframes* de cierre de bucle cerrado, dado que son éstos *Keyframes* los que aportan la información fundamental del mapa.

[1] `https://github.com/introlab/rtabmap/wiki/IROS-2014-Kinect-Challenge`

Como ya se habló en el capítulo de Infraestructura la herramienta `RTAB-Map` está compuesta a su vez por varias aplicaciones de utilidad. Una de ellas es `Database Viewer`[2]. Esta utilidad permite cargar desde una base de datos *sqlite* todos los datos generados por `RTAB-Map` al generar un mapa. Además, permite generar una nube de puntos 3D para visualizar dicho mapa. Para generar la nube de puntos, la herramienta invoca de nuevo al algoritmo de optimización [20] de `RTAB-Map`, pasándole los *Keyframes* marcados como de cierre de bucle y otros *Keyframes* que el algoritmo de GrapSLAM considera claves.

Fue en ese punto, en la generación de la nube de puntos del mapa, donde se introdujo una modificación en el código fuente de `Database Viewer` para almacenar en disco los *Keyframes* y la pose optimizada de la cámara. De este modo, a través de una ventana emergente de confirmación (ver Figura 5.1), pudimos generar el mapa que sirve de entrada para nuestro algoritmo.

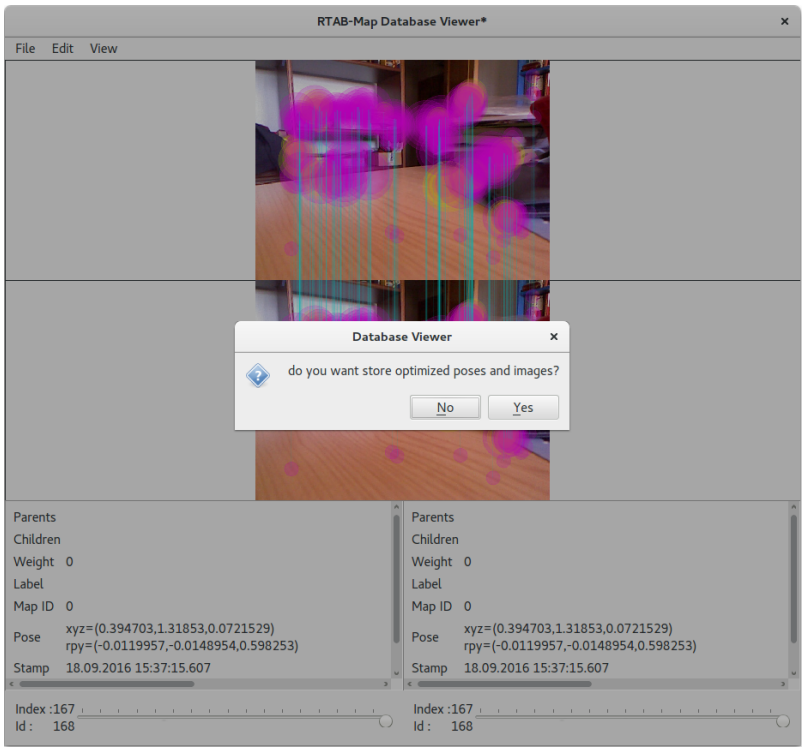


Figura 5.1: Ventana de confirmación para almacenar *Keyframes* en `Database Viewer`

Con la modificación efectuada en `Database Viewer` generamos un mapa compuesto de:

- Un fichero de definición del mapa: un fichero de texto que contiene el identificador de las imágenes que componen el mapa.
- Imágenes RGB: las imágenes de los *Keyframes* extraídos de *Database Viewer*.
- Imágenes de profundidad: las imágenes de profundidad de los *Keyframes* almacenados.
- Pose de las cámaras: un fichero con formato de *OpenCV* (*.yaml*) con la matriz *RT* de la cámara que se usó en la captura de cada *Keyframe*.

[2] `https://github.com/introlab/rtabmap/wiki/Tools`

5.2. Diseño del algoritmo de autolocalización

La Figura 5.2 muestra las distintas etapas que componen el algoritmo de autolocalización visual 3D propuesto en este trabajo. El algoritmo recibe como entrada un mapa y una imagen RGB de la cual se quiere conocer la posición 3D de la cámara que la tomó. A continuación, una vez el mapa ha sido leído, se extraen las características de las imágenes RGB que componen el mapa. El algoritmo trata de estimar la posición 3D de la cámara que tomó la imagen de entrada, buscando en el mapa aquella imagen que más se le parezca. Para ello, en primer lugar se extraen las características a la imagen RGB de entrada para después, mediante un proceso de emparejamiento (ver sección 5.3.3), determinar que imagen del mapa tiene más características en común con la imagen de entrada. En la etapa de emparejamiento, también se aplican distintos filtros que nos permiten incrementar la calidad de los éstos. De este modo, dicha etapa trata de encontrar la imagen del mapa que más se parece a la imagen de entrada (se encuentra en el mapa). Llegados a este punto, el algoritmo conoce la posición 3D de la cámara de la imagen que más se parece a la imagen de entrada, por lo que, mediante la aplicación de técnicas de autolocalización visual 3D se realiza una primera estimación de la posición 3D de la cámara que tomó la imagen de entrada. Con la primera estimación, se realiza un refinamiento de ésta para terminar de ajustar dicha pose mediante algoritmos de optimización.

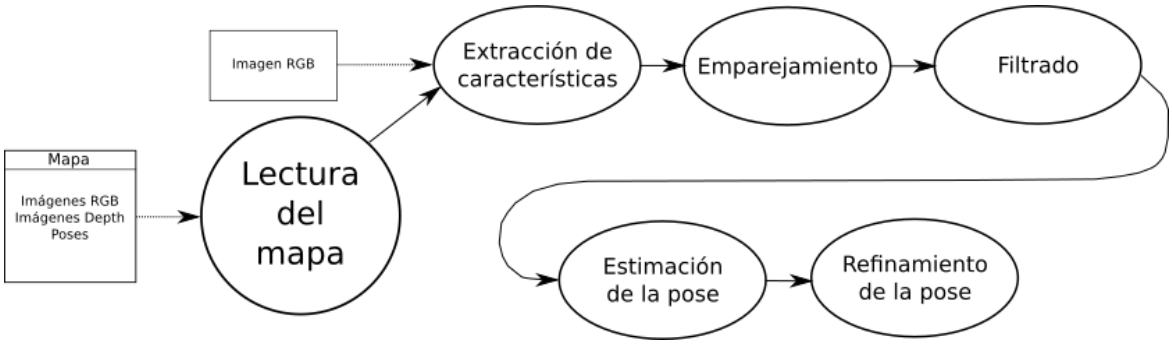


Figura 5.2: Diseño del algoritmo de autolocalización

5.3. La librería `mapper`

`mapper` es el nombre de la librería que se ha desarrollado para implementar el algoritmo de auto localización propuesto en este Trabajo Fin de Máster. Está desarrollada en C++ y dividida en varios módulos; la lectura del mapa, la extracción de las características de las imágenes, el emparejamiento de dichas características, la estimación de la pose y por último, una optimización de grano fino de la pose estimada. La librería además implementa su propio modelo de cámara para realizar proyecciones y reproyecciones, permite la generación de nubes de puntos 3D y funciona con formatos de ficheros estándar.

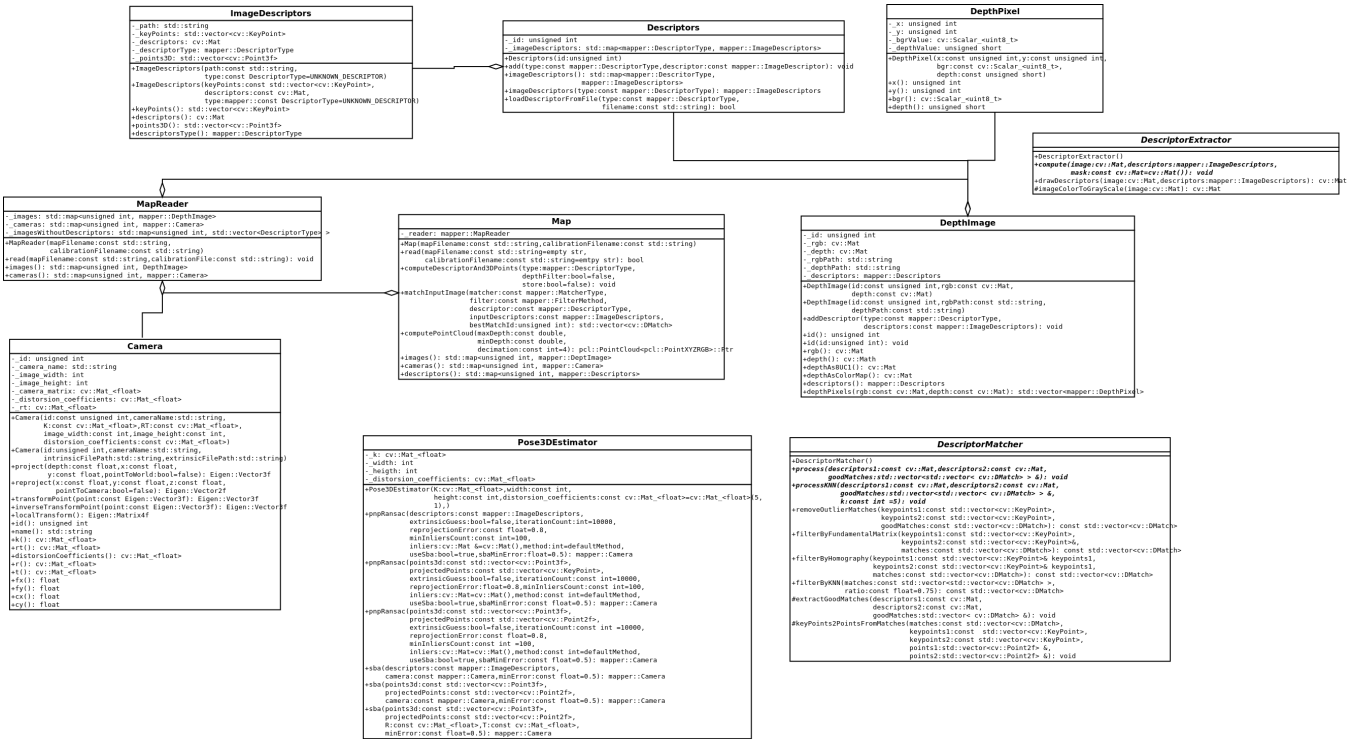


Figura 5.3: Diagrama UML simplificado de `mapper`

5.3.1. Lectura del mapa

La lectura del mapa generado en la sección anterior se realiza con la clase `MapReader` (ver Figura 5.3). Esta clase, que recibe en su constructor una ruta al fichero del mapa y al fichero que contiene los parámetros intrínsecos de la cámara, lee de disco todas las imágenes (tanto RGB como de profundidad) y las poses 3D de las cámaras generadas con `Database Viewer`. Una instancia de esta clase se encuentra como un miembro privado de la clase `Map`, la cual se encarga de gestionar todos los aspectos del mapa. Entre ellos, se encuentra la opción de la extracción de características de las imágenes del mapa, lo cual se realiza cada vez que `MapReader` lee un mapa a no ser que dichas características se hayan almacenado en disco previamente, en ese caso simplemente las carga desde disco a memoria.

Aunque se explicará con detalle más adelante, la clase `Map`, además de extraer características de las imágenes, también calcula los puntos 3D de dichas características que son almacenadas en disco junto con los descriptores de las imágenes.

5.3.2. Extracción de características

Antes de poder estimar la posición 3D de la cámara es necesario situar en qué región del mapa se encuentra. El algoritmo recibe como entrada una imagen que ha sido capturada en el área de trabajo, del cual tenemos un conjunto de imágenes que conforman el mapa. La hipótesis es sencilla, si la imagen de entrada se parece a alguna de las imágenes que contiene el mapa, entonces la cámara se encuentra el algún lugar desde donde puede ver la misma escena que la cámara que tomó la imagen contenida en el mapa. El

primer problema para la librería `mapper` es implementar un método que permita determinar qué imagen del mapa se parece más a la imagen de entrada.

Este es un problema sobradamente conocido dentro de la Visión Artifical y se puede abordar encontrando características discriminantes en las imágenes y emparejándolas con las de la imagen actual. Existen muchos métodos para detectar píxeles caracterísitos en una imagen, quizás dos de los más conocidos sean el método de Canny [4] para detectar bordes o el detector de esquinas de Harris [25]. Si bien estos métodos han demostrado su correcto funcionamiento a lo largo de los años, tienen algunas deficiencias que no los hacen aptos para determinadas aplicaciones. Por ejemplo, las esquinas tienen una forma muy característica fácil de detectar en una imagen, pero esta forma deja de ser tan fácilmente identificable a media que se cambia la escala de la imagen. Las condiciones de luminosidad en la escena también pueden alterar la detección de características con estos métodos, por lo que para resolver este problema se eligió un método de extracción de características más robusto que los anteriores que se explicará en detalle más adelante.

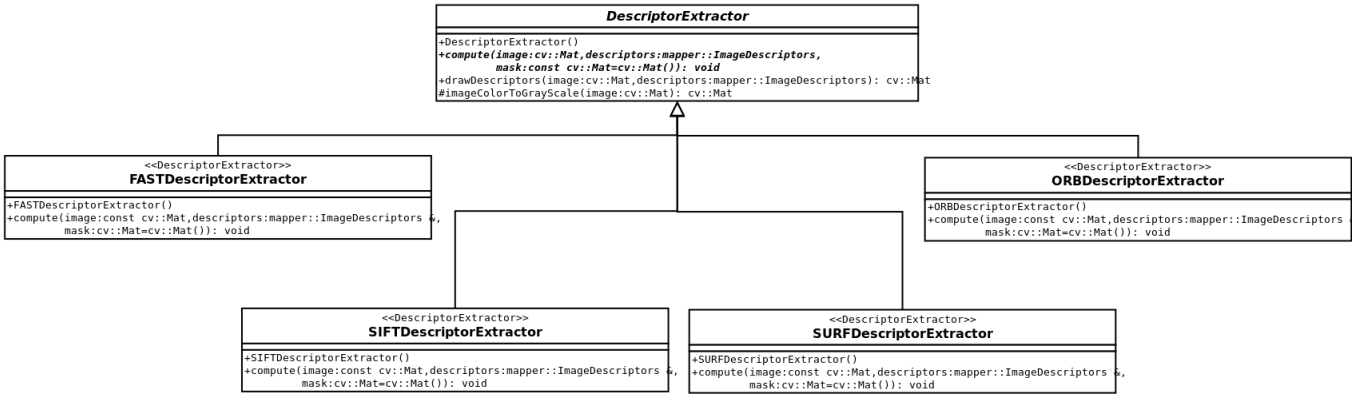


Figura 5.4: Diagrama UML del sistema de detección de características de `mapper`

La Figura 5.3 muestra el diagrama UML de clases del sistema de extracción de características de la librería `mapper`. Aunque finalmente se decidió utilizar un único método de extracción de características, la librería está preparada para usar varios de ellos. Debido a su diseño es fácil implementar más métodos si fuera necesario. El sistema cuenta con la clase base `DescriptorExtractor` que contiene el método abstractor `compute()`, de este modo una clase que implemente un método de extracción de características puede heredar de la clase base e implementar su método abstracto consiguiendo así que la librería pueda añadir un método más a su colección de métodos de una manera simple.

Existen dos puntos donde se realiza la extracción de caracterísitas; el primero es cuando se lee el mapa, si los descriptores de la imagen no se han extraído antes entonces se calculan y se almacenan en disco. El segundo lugar es cuando se le pasa al algoritmo una nueva imagen de entrada, en este punto la librería extraerá los puntos característicos de la imagen de entrada que posteriormente serán utilizados para el emparejamiento. El cálculo de descriptores se realiza con el método `computeDescriptors()` de la clase `Map`. Este método recibe como argumentos una imagen RGB, una imagen de profundidad (si se tiene), el tipo de descriptor que se quiere calcular y un booleano que indica si se quiere realizar un filtrado con la imagen de profundidad. El método devuelve un objeto de la clase *ImageDescriptors* que contiene los

descriptores de la imagen y los puntos claves (*KeyPoints*) de ésta. Si al método se le pasa una imagen de profundidad, permite obtener sólo aquellos descriptores que se encuentren entre una distancia máxima y una mínima, realizando así un filtrado sobre las características de la imagen en función de la profundidad de los puntos característicos. Además, para las imágenes del mapa de las cuales conocemos los parámetros intrínsecos y extrínsecos de la cámara con la cual fueron tomadas, se calculan los puntos 3D de cada punto característico en la imagen, teniendo así una correspondencia entre puntos 2D (*KeyPoints*) y sus puntos 3D en el espacio.

5.3.2.1. SIFT

El descriptor elegido para extraer características fue SIFT (*Scale Invariant Feature Transform*) un método propuesto por David G. Lowe en 2004 [40]. Una de las principales características de SIFT es que es invariante ante cambios de escala. SIFT utiliza un filtrado en diferentes escalas basado en Laplacianas de Gaussianas (LoG) que son utilizadas como detector de regiones (*blob*). Lo interesante de este método es que el valor σ actúa como un factor de escala, así que, modificando dicho valor se pueden encontrar regiones en diferentes escalas de la imágenes. El problema de las LoG es que son muy costosas computacionalmente por lo que SIFT usa una aproximación a las LoG conocida como el algoritmo de Diferencia de Gaussianas. La Diferencia de Gaussianas se calcula como la diferencia de un filtro Gaussiano con dos valores de σ, este proceso se realiza en las diferentes octavas obtenidas al calcular una pirámide [16] de la imagen. La Figura 5.5 ilustra este proceso:

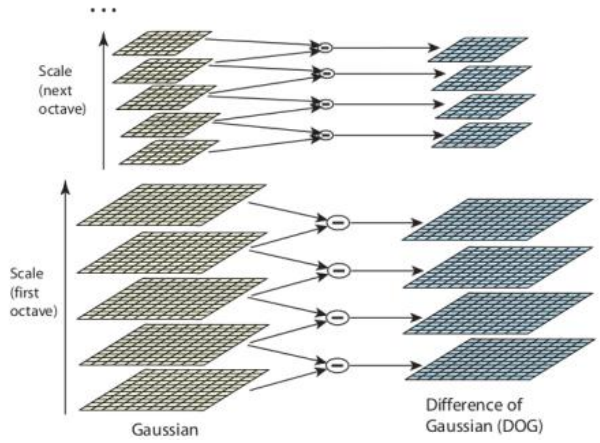


Figura 5.5: Diferencia de Gaussianas aplicado a las octavillas obtenidas en el proceso del cálculo de una pirámide de imágenes

Con las DoG calculadas se buscan en ellas un máximo local comparando un píxel con sus n vecinos. Este proceso se realiza en las distintas DoG (distintas escalas de la imagen). El máximo de todos estos píxeles es candidato a punto característico.

SIFT también es invariable a rotación y translación. Para ello de nuevo, y en función de la escala, se selecciona un vecindario de píxeles alrededor del punto característico. Para esa región se calcula la magnitud y dirección del gradiente.

Finalmente el descriptor SIFT se calcula de la siguiente manera, se coge una región de 16x16 alrededor del punto característico. Esta región se divide a su vez en 16 sub-bloques de 4x4, por cada sub-bloque se calcula un histograma de 8 direcciones. A esta información se le añaden varias medidas sobre la iluminación para incrementar su robustez. De este modo se obtienen descriptores de la imagen invariantes ante escala, rotación y translación, además de ser robustos frente a cambios de iluminación en la escena.

5.3.3. Emparejamiento de características

Como ya hemos visto en la sección anterior, del proceso de extracción de características se obtienen descriptores de la imagen. Cada uno de estos descriptores, que en realidad son pequeñas regiones de la imagen, contiene un punto 2D que indica la localización del pixel característico en la imagen. Este punto es conocido como *KeyPoint*, y el método que tenemos para decidir si dos imágenes son parecidas es encontrar el mayor número de correspondencias de *KeyPoints* entre una región y otra. Este proceso, que se conoce como emparejamiento (*matching*), permite hacernos una idea de cuánto una imagen se parece a la otra. A mayor número de emparejamientos podemos asumir que las imágenes se parecerán más. Aunque como veremos más adelante, esto no siempre se cumple, ya que a veces se cometen errores en el emparejamiento y uno o varios *KeyPoints* son emparejados de manera incorrecta.

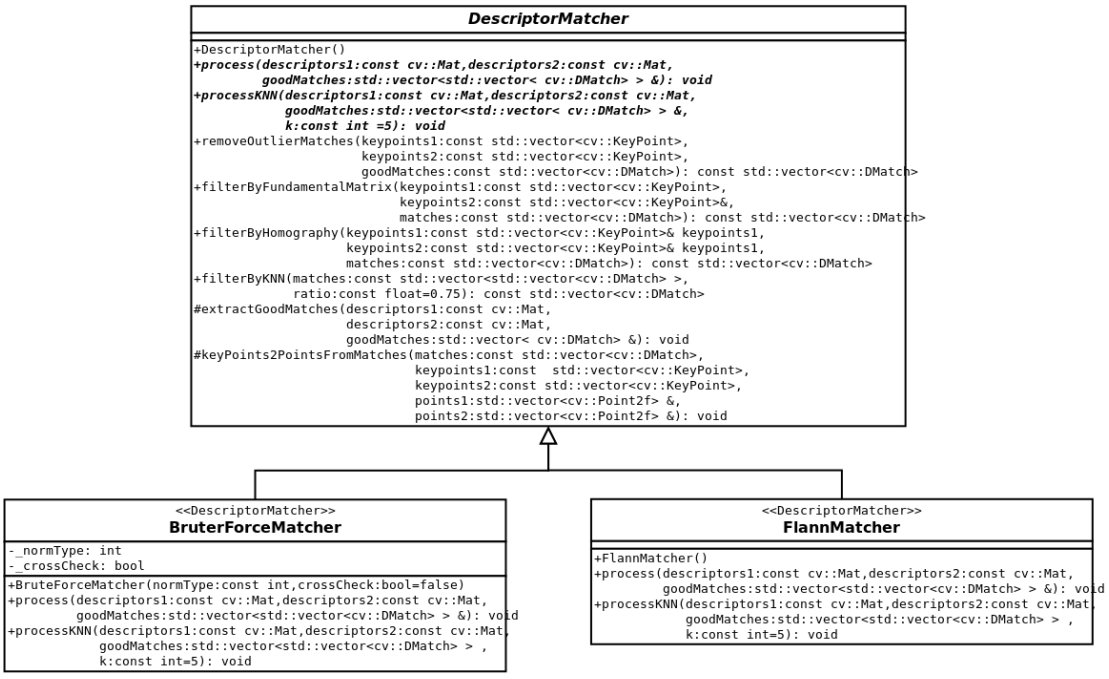


Figura 5.6: Diagrama de clases del sistema de emparejamiento de `mapper`

La librería `mapper` implementa el sistema de emparejamiento descrito en la Figura 5.6. Utiliza un enfoque de herencia entre clases que permite de manera muy simple añadir nuevos emparejadores (`matchers`). *OpenCV* proporciona dos emparejadores, el de fuerza bruta (`BFMatcher`[3])

[3] `http://docs.opencv.org/2.4/modules/features2d/doc/common_interfaces_of_descriptor_matchers.html?highlight=flannbasedmatcher#bfmatcher`

y `FlannBasedMatcher`. En la librería se utiliza este último que está basado en FLANN[4] [46] debido a que es más rápido que el emparejador por fuerza bruta. Este emparejador se basa en algoritmos de búsqueda del vecino más próximo (*Nearest Neighbor*) para encontrar el mejor emparejamiento entre las distintos *KeyPoints* candidatos. Estos algoritmos se basan en un método para clasificar casos basándose en su parecido a otros casos, parten de la idea de que los casos parecidos están próximos entre sí, mientras que los casos que no se parecen permanecen alejados. Los casos próximos se denominan vecinos, así cuando se presenta un nuevo caso se calcula la distancia con respecto de los vecinos y se clasifica con aquellos vecinos más cercanos. A diferencia del emparejador por fuerza bruta, este método no devuelve el mejor emparejamiento por cada descriptor, sino que devuelve un vector con los N emparejamientos que obtuvo para un descriptor después de aplicar el algoritmo KNN.

En `mapper` cada vez que se pasa una nueva imagen al algoritmo se extraen sus características y, a continuación, se busca el mejor emparejamiento entre todas las imágenes del mapa. La imagen del mapa que obtenga más emparejamientos, es elegida como la imagen más parecida a la imagen de entrada y será con la pose asociada a dicha imagen con la que se estimará la pose de la nueva imagen tal y como veremos más adelante.

El proceso de emparejamiento se realiza a través del método `matchInputImage()` de la clase `Map` que recibe como argumentos el emparejador que se va a utilizar, el tipo de descriptor y los descriptores de la imagen de entrada y de la otra imagen con la que se quiera comparar. El método nos da una distancia entre el *KeyPoint* que queremos emparejar a los *n KeyPoints* más parecidos en la segunda imagen. La implementación de *OpenCV* devuelve un vector de vectores (un vector por cada descriptor de la primera imagen). En cada posición de dicho vector se encuentra otro vector de objetos `DMatch`[5] que nos indica la distancia del *KeyPoint* de la primera imagen con los *n KeyPoints* encontrados en la segunda imagen.

5.3.4. Filtrado de emparejamientos

El proceso de emparejamiento no está exento de errores ya sea por la calidad de la imagen, por las características de la iluminación, por la textura (o ausencia de ella) de los objetos en la escena, etc. es posible que el calculo de descriptores de una imagen no sea como se espera. En ocasiones estos métodos sólo encuentran descriptores en una región de la imagen (a veces incluso en un único punto), lo que provoca que el proceso de emparejamiento no pueda hacer otra cosa más que intentar emparejar los descriptores que tiene para trabajar. Es un problema con el que hay que convivir e intentar mitigar sus efectos lo máximo posible.

Para nuestro algoritmo estos errores de emparejamiento pueden provocar que la estimación de la posición 3D de la cámara sea completamente errónea. Por este motivo, se añadió a la librería `mapper` una etapa de filtrado que permitiera asegurar, en la medida de lo posible, que los emparejamientos de los que disponemos sean de buena calidad. Podemos dividir los filtros implementados en dos grupos; el

[4] `http://docs.opencv.org/2.4/modules/features2d/doc/common_interfaces_of_descriptor_matchers.html?highlight=flannbasedmatcher#flannbasedmatcher`

[5] `http://docs.opencv.org/2.4/modules/features2d/doc/common_interfaces_of_descriptor_matchers.html?highlight=flannbasedmatcher#dmatch`

primero es un filtrado en apariencia sin información geométrica de la cámara, sólamente se trabaja con los emparejamientos calculados. El segundo grupo tiene información de la cámara y consiste en añadir restricciones geométricas a los emparejamientos con el fin de eliminar aquellos que no cumplan dichas restricciones.

5.3.4.1. Filtrado de sobre-saliencia

Del método de extracción de características visto en la sección 5.3.3 se obtiene un vector de vectores de emparejamiento con las distancias del *KeyPoint* de la primera imagen que se quiere empajerar a los n *KeyPoints* detectados en la segunda imagen. Para el caso de los descriptores SIFT, la medida de distancia es la distancia euclídea. En este caso esta medida indica cómo de similar son los descriptores entre sí, por tanto, la distancia ideal sería 0 y a medida que esta distancia aumenta, la similitud entre ambos descriptores disminuye.

En la sección 5.3.3 también se explicó que la librería `mapper` utiliza un emparejador basado en algoritmos del vecino más próximo (*Nearest Neighbor*), concretamente la implementación de *OpenCV* utiliza el algoritmo KNN (*K Nearest Neighbor*). Para nuestro algoritmo esto puede suponer un problema ya que, con este método de emparejamiento obtenemos una lista de posibles emparejamientos para nuestro *KeyPoint*. Con este funcionamiento se pueden dar casos como el de la Figura 5.7. En ella se aprecia que el *KeyPoint* de la imagen izquierda ha obtenido dos posibles emparejamientos en la imagen de la derecha. A simple vista podemos observar que el emparejamiento que se encuentra más abajo es erróneo, mientras que el correcto es el superior. Sin embargo, las esquinas del libro son prácticamente iguales por lo que en este caso la medida de distancia que nos proporcionan los emparejamientos serán muy bajas, lo que significa que ambos *KeyPoints* son muy similares. Para nuestro algoritmo no queremos que un *KeyPoint* se empareje con dos *KeyPoints* de la otra imagen porque no podemos asegurar que elegimos el *KeyPoint* correcto, por tanto, el filtro de sobre-saliencia descarta ambos emparejamientos prefiriendo así tener menos emparejamientos pero asegurando que no se produzcan situaciones como la de la recién expuesta.

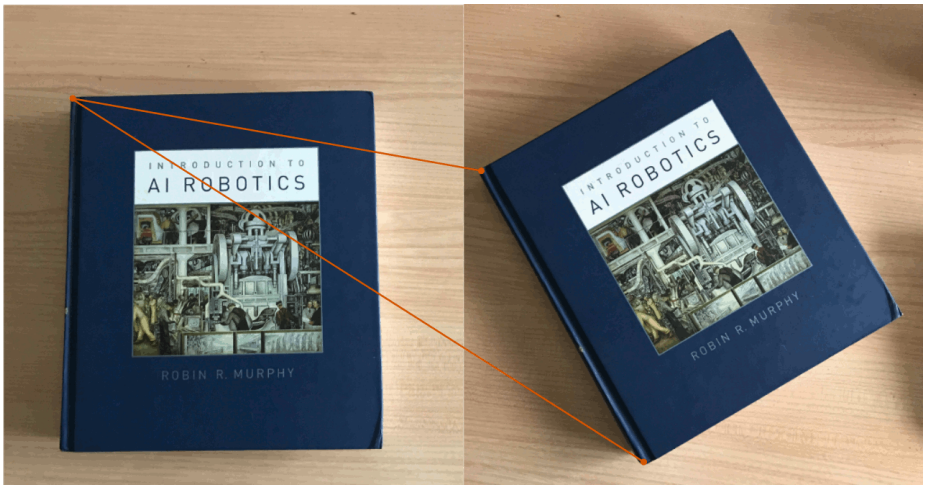


Figura 5.7: Un *KeyPoint* con varios emparejamientos

El filtrado de sobre-saliencia implementado en `mapper` recorre los emparejamientos para cada descriptor.

Por cada vector de emparejamientos de un descriptor se cogen los dos primeros (los emparejamientos están ordenados por la similitud entre los descriptores). Con estos dos, se evalúa como de distintos son. Para ello se divide la distancia de ambos emparejamientos entre sí, si el resultado es mayor a un valor determinado experimentalmente (0,75), se descartan todos los emparejamientos para dicho descriptor. Que el resultado de la división sera superior a 0,75 nos indica que los emparejamientos son muy parecidos entre sí, ya que la medida de distancia de los emparejamientos es una medida de similitud, luego si dos emparejamientos tienen una distancia muy parecida podemos decir que son iguales. En caso de que el resultado sea menor al ratio establecido, se guarda el primer emparejamiento, que sobresale lo suficiente del segundo.

5.3.4.2. Filtrado por matriz fundamental

La matriz fundamental (F) es la representación algebraica de la geometría epipolar [26]. La geometría epipolar entre dos vistas es la geometría de la intersección de los planos de dos imágenes y el plano que tiene como eje la línea que une los centros de las cámaras (*baseline*). Esta geometría pretende buscar la correspondencia de puntos en el emparejamiento de varías vistas.

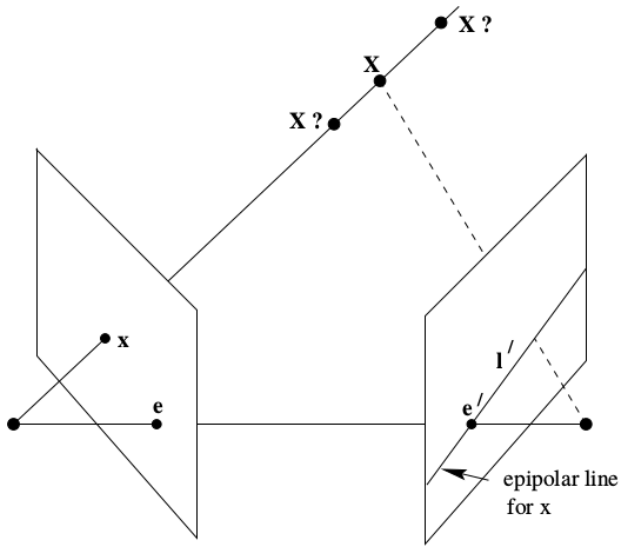


Figura 5.8: Geometría epipolar

Si tuviéramos un punto 3D (X) y su proyección 2D en el plano de imagen (x), la geometría epipolar dice que proyectando en el plano de la otra imagen el rayo formado entre X y x, el punto 2D x' (la correspondencia de x en la otra imagen) se encontrará en algún lugar de la línea $I^{'}$ (ver Figura 5.8). Esta geometría es comúnmente utilizada en la reconstrucción 3D ya que permite encontrar correspondencias de los puntos a reconstruir en la otra imagen limitando su búsqueda a una línea.

En nuestro caso el problema es distinto ya que tenemos un punto en una imagen (*KeyPoint*) y un conjunto de posibles correspondencias en la otra imagen obtenidos en la etapa de emparejamiento. Si pudiésemos encontrar una restricción epipolar que asociase a nuestro *KeyPoint* un solo punto en la otra imagen, filtraríamos los emparejamientos asegurándonos que el punto resultante sea correspondencia del *KeyPoint*.

Podemos definir la matriz fundamental F con la siguiente ecuación:

$$x'^{T} F x = 0 \tag{5.1}$$

donde x y x' son correspondencias de puntos 2D entre dos imágenes, así si tenemos al menos siete correspondencias [26] podemos calcular la matriz fundamental entre las dos vistas. La función `findFundamentalMat()`[6] de *OpenCV* nos permite encontrar la matriz fundamental dado un conjunto de correspondencias 2D. La función implementa los algoritmos descritos por Richard Hartley y Andrew Zissermarn [26] en su libro *Multiple View Geometry in Computer Vision*. En `mapper` se invoca a dicha función para que encuentre F con el método RANSAC[19], el cual devuelve una lista con las correspondencias que fueron marcadas como *inliers*. De este modo podemos filtrar los emparejamientos asegurando que cumplen la restricción epipolar.

5.3.4.3. Filtrado por homografía

Una homografia es una transformación que relaciona puntos en una imagen con sus correspondientes puntos en otra. Como muestra la Figura 5.9 el rayo correspondiente al punto x intersectará con el plano π en x_π, este punto se proyecta sobre la otra imagen en el punto x'. La relación entre los puntos x y x' es la homografía inducida por el plano π. Esta es una relación de la geometría proyectiva que sólo depende de la intersección de planos con líneas. La homografía relaciona puntos entre dos vistas si los puntos se encuentran en el mismo plano.

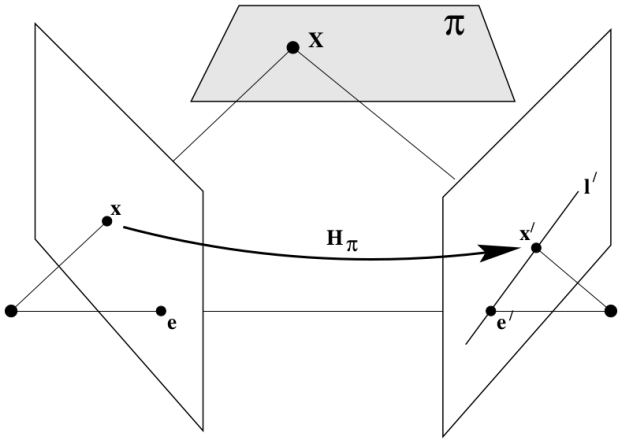


Figura 5.9: Homografía inducida por un plano

Existen dos relaciones entre dos vistas; la primera es la geometria epipolar (ver sección 5.3.4.2) donde un punto en una vista determina una línea en la otra vista donde se encontrará la correspondencia del punto. La segunda relación es la homografía, donde un punto en una vista determina un punto en la otra que es la intersección de un rayo con un plano.

[6] `http://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#findfundamentalmat`

De un modo similar al visto en el cálculo de la matriz fundamental, podemos definir la homografía con la siguiente ecuación:

$$x' = H_\pi x \tag{5.2}$$

donde x y x' son correspondencias de puntos 2D entre dos vistas. Por tanto si encontrásemos una homografía entre los puntos obtenidos en la etapa de emparejamiento podríamos filtrar aquellos puntos que se encuentran en el mismo plano y eliminar aquellos que queden fuera. La librería `mapper` utiliza la función `findHomography()`[7] de *OpenCV*, que recibe como parámetros la correspondencia de puntos 2D entre dos imágenes y devuelve una máscara con los índices de los emparejamientos que se usaron para encontrar la homografía. De este modo conseguimos filtrar los emparejamientos y asegurar que los que queden son puntos que se encuentran en el mismo plano.

5.3.5. Estimación de la pose 3D

Llegados a este punto el algoritmo ya ha determinado qué imagen del mapa es la más parecida a la imagen de entrada. Conviene recordar que por cada imagen del mapa se tiene la posición 3D (una matrix *RT*) de la cámara que tomó la imagen, los puntos característicos (*KeyPoints*) emparejados y filtrados y la reproyección 3D de dichos puntos. Para la imagen de entrada el algoritmo dispone de una lista de puntos 2D, que son los puntos característicos de la imagen de entrada que han sido emparejados y filtrados con los puntos característicos de la imagen del mapa. Como los puntos característicos han sido emparejados y la reproyección 3D solo se realizó con los *KeyPoints* filtrados de la imagen del mapa, existe una correspondencia entre los puntos 3D y los *KeyPoints* de la imagen de entrada.

Con lo descrito en el párrafo anterior tenemos todo lo necesario para realizar una estimación de la posición 3D de la cámara a través del algoritmo PnP. Este algoritmo espera una colección de puntos 3D y la proyección de dichos puntos sobre el plano de la imagen. La librería `mapper` utiliza la función `solvePnPRansac()`[8] de *OpenCV* para realizar la estimación de la pose. La función recibe como parámetro los puntos 3D y sus proyecciones 2D, la matriz de la cámara K y los coeficientes de distorsión (proporcionados como entrada al algoritmo), y algunos parámetros para el algoritmo RANSAC, como el número máximo de iteraciones o el error máximo permitido. La función estima la posición 3D de la cámara minimizando el error de reproyección en un proceso de optimización de mínimos cuadrados. Como resultado se devuelve un vector de translación y un vector de rotación que será convertido a una matriz de rotación $3x3$ con la función `Rodrigues()`[9] de *OpenCV*.

Todo este proceso se realiza en la clase `Pose3DEstimator` (ver Figura 5.3) a través del método `pnpRansac()` que devuelve un objeto de la clase `Camera` con la estimación de la posición 3D de la cámara que tomó la imagen de entrada.

[7] `http://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#findhomography`

[8] `http://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#solvepnpransac`

[9] `http://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#rodrigues`

5.3.6. Ajuste de grano fino de la pose

Este es el último paso del algoritmo y consiste en refinar la estimación de la pose obtenida con `PnP` (ver sección 5.3.5) con el algoritmo SBA [3] (*Sparse Bundle Adjustment*). En el problema del SfM (*Structure from Motion*) descrito en la sección 3.3 es muy típico el uso de SBA en la parte final del proceso para refinar las estimaciones de las poses. Nuestro problema es distinto ya que no tenemos múltiples vistas que queramos optimizar pero aún así, y gracias a la librería `cvSBA`, podemos realizar este último ajuste de nuestra estimación únicamente con una colección de puntos 3D, sus proyecciones 2D y la estimación inicial de la cámara.

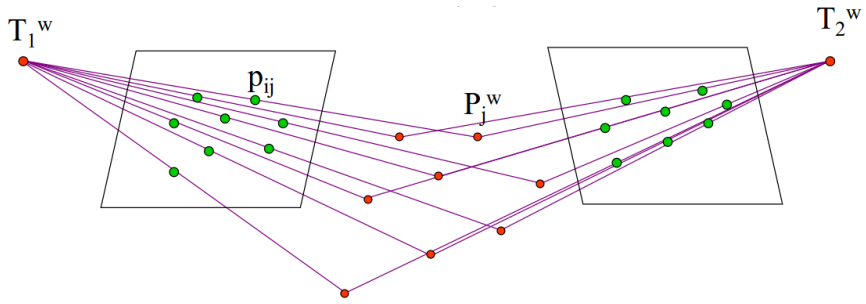


Figura 5.10: Ajustes de haces de luz

El ajuste de haces (*Bundle Adjustment*) trata de refinar la reconstrucción de una escena en base a su estructura 3D y los parámetros de las vistas que ven dicha escena. El término *bundle* se refiere a los "haces" producidos al unir mediante una línea el centro de cada cámara con los puntos 3D de la escena (ver Figura 5.10). Este refinamiento se consigue realizando una minimización por mínimos cuadrados del error de reproyección. La librería `cvSBA` permite varios modos de funcionamiento, y en función del elegido se realiza la optimización en distintos parámetros de la escena.

`-MOTION`: con esta opción la optimización se realiza únicamente sobre los parámetros extrínsecos de la cámara.

`-STRUCTURE`: si se selecciona esta opción se optimizan los puntos 3D de la escena para que cuadren con las cámaras.

`-MOTIONSTRUCTURE`: es la combinación de las dos anteriores donde se optimizan tanto los puntos 3D de la escena como las posiciones de las cámaras.

En `mapper` se invoca a `cvSBA` con la opción `MOTION` en la clase `Pose3DEstimator` (ver Figura 5.3), pasando un *flag* al método `pnpRansac()` para que ajuste la estimación de la cámara. Si se activa la opción de refinamiento, la posición final devuelta por este método es la estimación de PnP más el ajuste de SBA.

5.4. El componente autolocalizador `localizer`

El componente `localizer` es un recubrimiento de la librería `mapper` para su integración en el entorno JdeRobot (ver sección 4.2). Está desarrollado en C++ y, mediante las interfaces ICE que oferta, permite

que otros componentes se conecten al él para obtener los resultados de la estimación continua de la pose 3D en tiempo real.

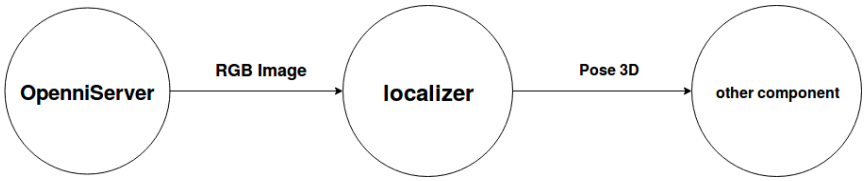


Figura 5.11: Interfaces de `localizer`

Este componente se conecta (ver Figura 5.11), mediante interfaces `ICE`, con el componente `OpenniServer` encargado de obtener y publicar las imágenes, tanto RGB como de profundidad, de sensores RGB-D. Para nuestro problema sólamente son necesarias las imágenes en color por lo que `localizer` sólamente se conecta a la interface que ofrece imágenes RGB.

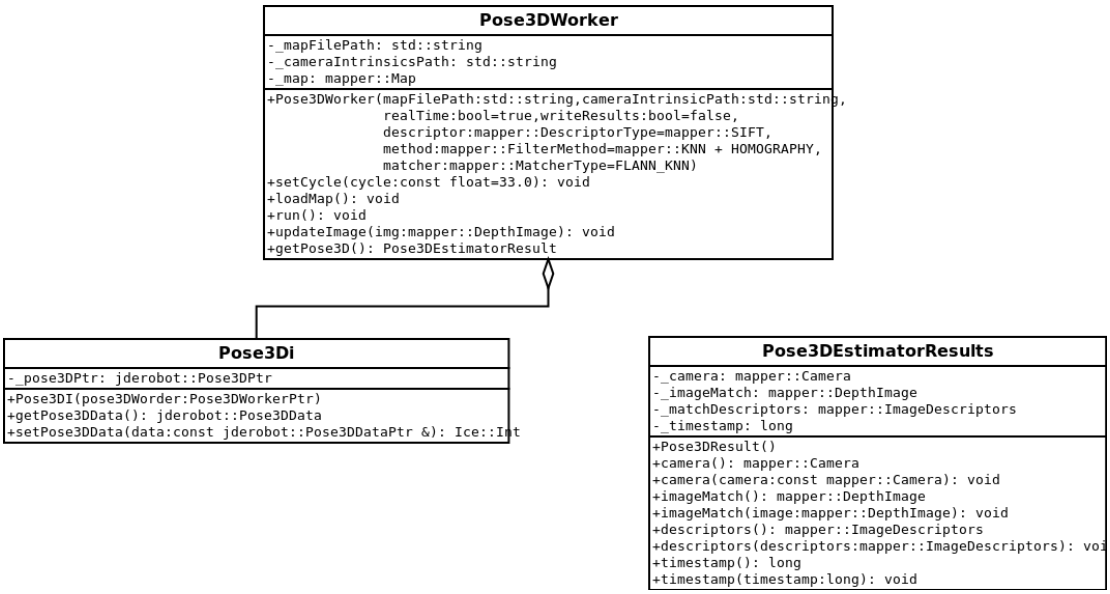


Figura 5.12: Diagrama UML de clases de `localizer`

`localizer` va encolando en una cola todas las imágenes recibidas desde `OpenniServer` para que un hilo distinto del principal estime las poses y las encole en una cola de salida. La librería `mapper` devuelve sus estimaciones como una matriz RT, sin embargo, en JdeRobot la manera de representar posiciones 3D es mediante la interface `Pose3D` que está definida del siguiente modo:

```
/**
* Pose3D data information
*/
class Pose3DData
{
        float x;
        float y;
        float z;
```

```
        float h;
        float q0;
        float q1;
        float q2;
        float q3;
        Time timestamp;
};

/**
* Interface to the Pose3D.
*/
interface Pose3D
{
        idempotent Pose3DData getPose3DData();
        int setPose3DData(Pose3DData data);
};
```

Una posición 3D en JdeRobot está definida mediante un punto y un cuaternión que indica la orientación. Además, la interface añade una marca de tiempo, de este modo un cliente que se conecte a la interface `Pose3D` ofertada por `localizer` podrá saber el orden de las posiciones 3D estimadas.

Como `localizer` hacer uso de `mapper` es necesario indicarle, mediante la lista de argumentos, la ruta al fichero de definición del mapa y la ruta al fichero que contiene la matriz K de la cámara utilizada para crear el mapa. A parte de publicar posiciones 3D de las imágenes que recibe, `localizer` tiene un modo de funcionamiento *offline* donde estima la posición de una lista de imágenes pasadas en la lista de argumentos y las guarda en disco con el formato de *OpenCV* .yaml.

5.5. El componente visualizador `autoloc_viewer`

`autoloc_viewer` es una aplicación gráfica para la depuración de la librería `mapper` y el componente `localizer`. Desarrollada en C++ cuenta con una interface gráfica de usuario escrita en *Qt5*, que permite la visualización de las nubes de puntos generadas por el mapa, las poses estimadas y la trayectoria tridimensional de la cámara.

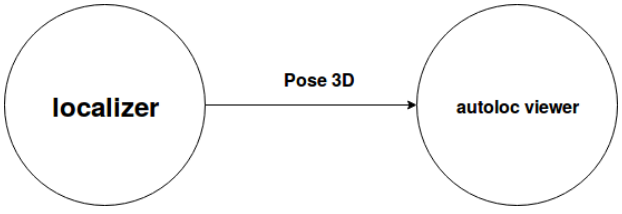


Figura 5.13: Interfaces de `autoloc_viewer`

La herramienta también está integrada en JdeRobot y se conecta al componente `localizer` mediante

la interface ICE `Pose3D`. Por ella recibe la poses de las cámaras estimadas por `mapper` y las muestra en pantalla, si tiene más de una pose, también dibujará la trayectoria que ha seguido la cámara.

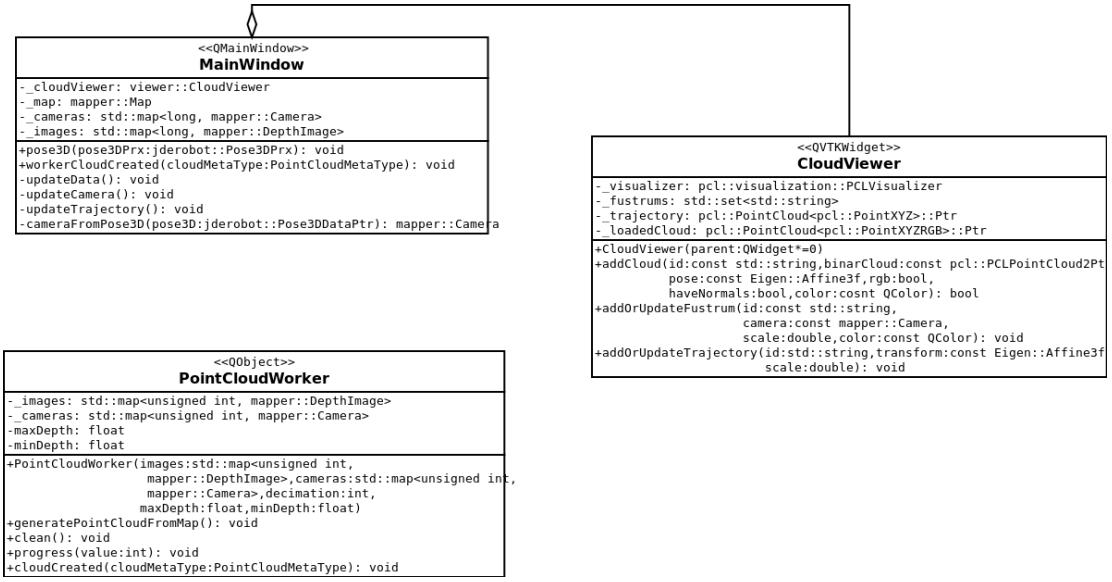


Figura 5.14: Diagrama UML de clases de `autoloc_viewer`

Como se puede apreciar en la Figura 5.14 la clase `CloudViewer` (que hereda de `QTVTKWidget`) tiene como miembro de clase un visualizador 3D de la librería *PCL*. Este visualizador sólamente es capaz de visualizar nubes de puntos, por lo que la trayectoria y el polígono que representa a las cámaras, han de ser transformados al tipo de dato `pcl::PointCloud` para ser mostrados.

La clase `PointCloudWorker`, que hereda de `QObject`, se ejecuta en un hilo separado cada vez que se quiera generar una nube de puntos a partir de un mapa. La clase calcula los puntos 3D de la nube de puntos invocando a una función de utilidad (`computePointCloud()`) de la librería `mapper`. Al heredar de la clase `QObject` es posible reimplementar el método `progress` para indicar el progreso de la generación de la nube de puntos al hilo encargado de la gestión de eventos de aplicación.

Al igual que el componente `localizer`, la aplicación `autoloc_viewer` también dispone de un método de funcionamiento *offline* donde es posible cargar un conjunto de cámaras (ficheros *.yaml*) de disco para poder visualizar la trayectoria y la última cámara añadida, así como el mapa donde se realizó la captura.

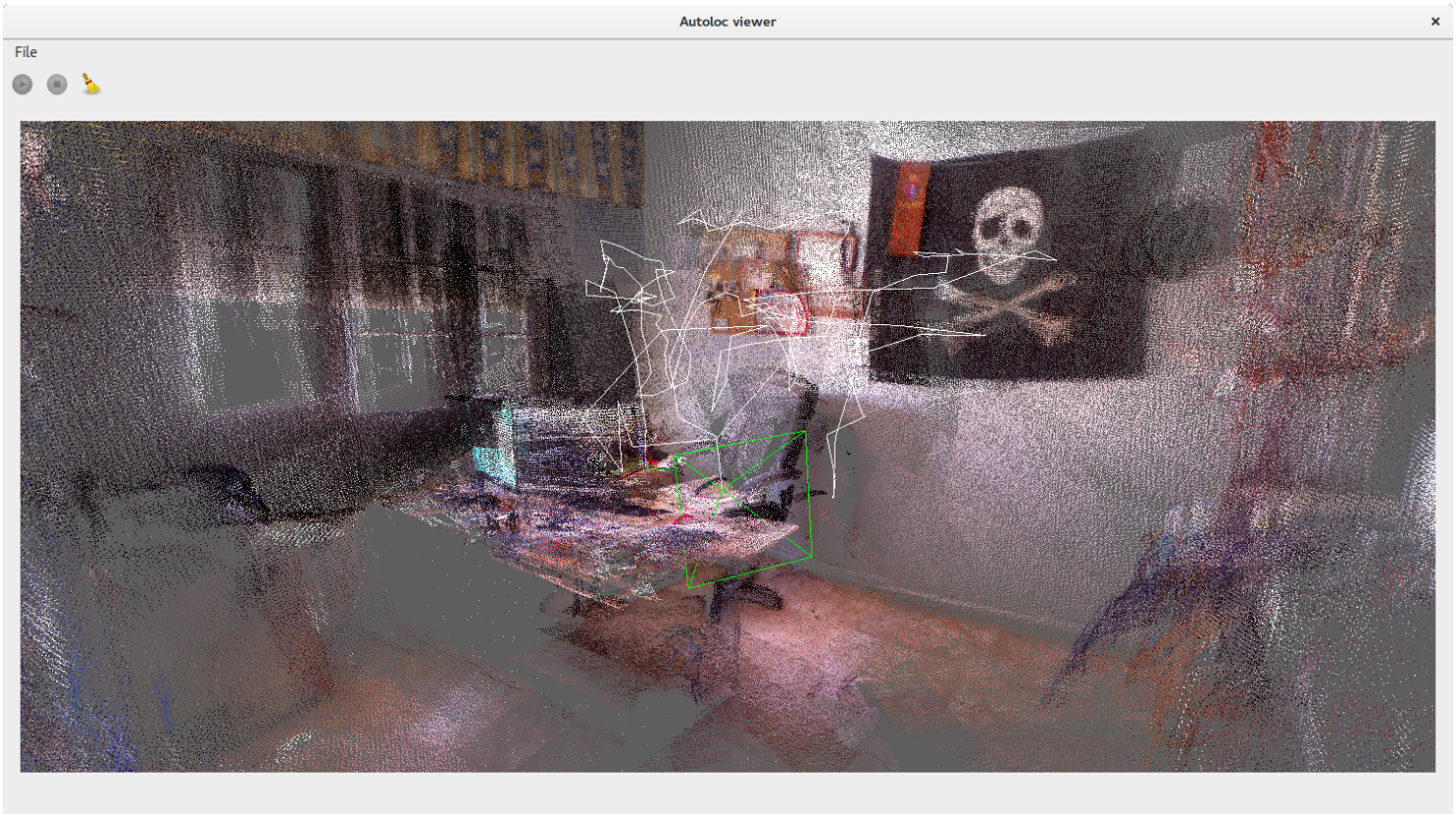


Figura 5.15: Vista del mapa, trayectoria y cámara en `autoloc_viewer`

En la Figura 5.15 se puede apreciar la nube de puntos de un mapa, la trayectoria que siguió la cámara (en gris) y la última pose de cámara (en verde).

Capítulo 6

Experimentos

En este capítulo se exponen los resultados obtenidos en la ejecución del algoritmo de autolocalización descrito en el capítulo anterior. Primero se presentan los resultados individualizados para cuatro escenarios de ejecución distintos, a continuación se muestra una comparativa en modo tabla con todos los resultados obtenidos en los distintos escenarios y por último se muestra de manera gráfica los distintos filtrados de emparejamientos implementados en el proyecto.

6.1. Estimación de la pose 3D

La Figura 6.1 muestra una representación de un mapa generado con RTAB-Map. La línea negra más gruesa, representa el grafo optimizado que contiene el menor número de vistas. Los triángulos representan las poses de las cámaras que capturaron los *Keyframes*, representados por los rectángulos, del mapa.

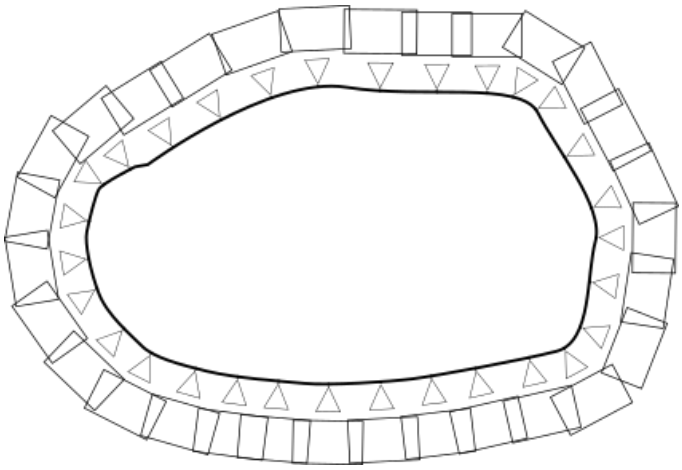


Figura 6.1: Representación esquemática de un mapa 3D

Para evaluar la calidad del algoritmo implementado se ha ejecutado en cuatro escenarios distintos. En todos ellos se ha realizado el mismo procedimiento. Primero se genera un mapa del entorno, que como se explicó en la sección 2.2, está compuesto por un conjunto de imágenes (de color y de profundidad) y las poses de las cámaras que tomaron dichas imágenes. Segundo, se van cogiendo las imágenes RGB del mapa

una a una, como si fueran la entrada del algoritmo sacándolas (ver Figura 6.2) del conjunto de imágenes que componen el mapa. De este modo, si el mapa tuviese N imágenes, se realizan N evaluaciones sobre un mapa que contiene $N-1$ imágenes. Al no estar en el mapa ni la imagen ni la cámara que se está tratando de estimar, el algoritmo deberá encontrar una pose 3D cercana al lugar donde debería estar la cámara. Como conocemos esa cámara la usamos como verdad absoluta para calcular el error de reproyección con respecto de la cámara estimada en el proceso, de este modo tenemos una medida que nos permite caracterizar el error que se comete en la estimación.

Este proceso se ha realizado con las distintas combinaciones de filtrado de emparejamientos descritos en la sección 5.3.4. Cabe resaltar que al eliminar del mapa una de las poses con sus correspondientes imágenes (RGB y de profundidad) se produce un vacío en el grafo. Si la imagen seleccionada no tiene suficiente solapamiento con las imágenes adyacentes, pueden provocarse errores en el emparejamiento e incluso no encontrar el emparejamiento correcto al no disponer de suficiente información. Si durante el proceso de estimación se obtiene un error de reproyección superior a 100 píxeles, esa estimación se trata como un error del algoritmo y se toma en cuenta igual que otros errores que se puedan surgir durante el proceso como la insuficiencia de *KeyPoints* excepciones en los algoritmos de estimación u optimización, etc.

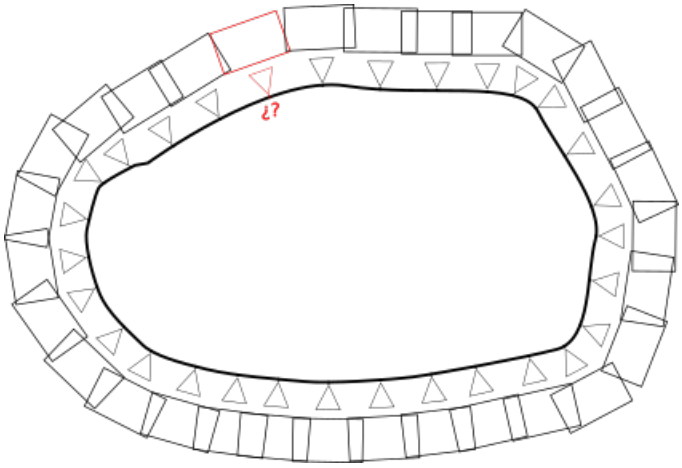


Figura 6.2: Procedimiento para la evaluación del algoritmo

En la sección 6.1.1 se muestran los resultados para el primer experimento sobre el laboratorio de Robótica de la URJC en el campus de Fuenlabrada, la sección 6.1.2 sobre un despacho casero, en la sección 6.1.3 el experimento se realizó sobre un despacho de la URJC y finalmente, la sección 6.1.4 muestra los resultados obtenidos para un despacho multipuesto de la biblioteca de la URJC del campus de Fuenlabrada.

6.1.1. Escenario 1

Este laboratorio alberga un campo reglamentario de fútbol de la competición *RoboCup*, además de varias mesas, estanterías y puestos de trabajo. En el escenario sólamente se disponía de iluminación artifical obtenida a través de unos fluorescentes en el techo. Se realizó un mapa tal como se describe en la sección 5.1. Después de extraer las imágenes marcadas como *Keyframe* de RTAB-Map, el mapa resultante contiene 274 imágenes con sus correspondientes poses e imágenes de profundidad.

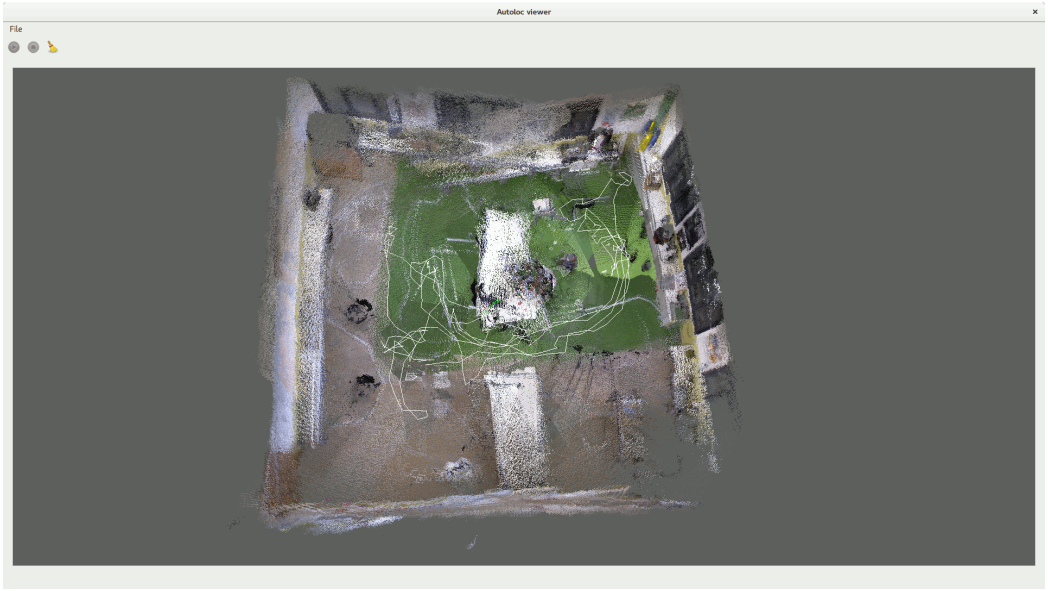


Figura 6.3: Vista superior del mapa generado en el Laboratorio de Robótica de la URJC

Las Figuras 6.3 y 6.4 muestran distintas vistas del laboratorio. En gris se puede apreciar la trayectoria de la cámara al generar el mapa y, el polígono verde se corresponde con la última posición de la cámara.

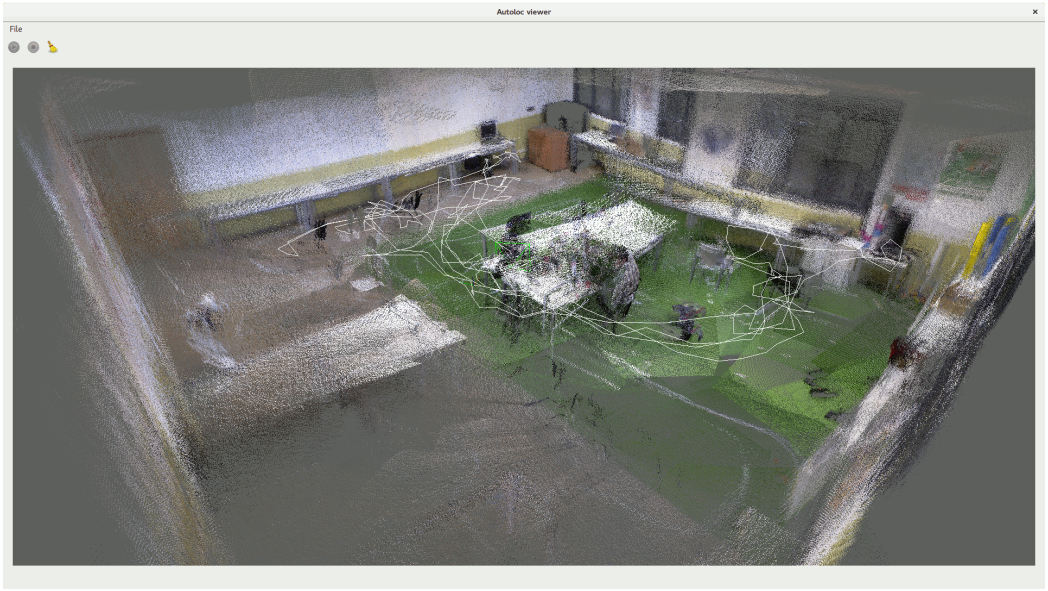


Figura 6.4: Vista lateral del mapa generado en el Laboratorio de Robótica de la URJC

Como muestra la trayectoria de la cámara, el mapa se realizó realizando varias pasadas por el mismo lugar sin pasar por todos los recovecos del laboratorio. El realizar varias pasadas ayuda a *RTAB-Map* a determinar con mayor exactitud el cierre de bucle. A pesar de ser un mapa de muy buena calidad, se observan errores como en la Figura 6.3 donde se puede ver cómo una de las paredes no es totalmente recta, probablemente debido a que durante la generación del mapa no sé pasó cerca de esa región y el algoritmo de *GraphSLAM* no pudo añadir más restricciones al grafo. Sin embargo, la última posición de la cámara y la trayectoria de ésta son coherentes con la estructura el mapa.

6.1.1.1. Sin filtrado

En este experimento se trata de estimar la posición 3D de la cámara sin realizar ningún tipo de filtrado en la etapa de emparejamiento. Por tanto los puntos 3D y las proyecciones 2D que llegan a la etapa de estimación de la pose no han sido tratados, son los obtenidos con los métodos habituales.

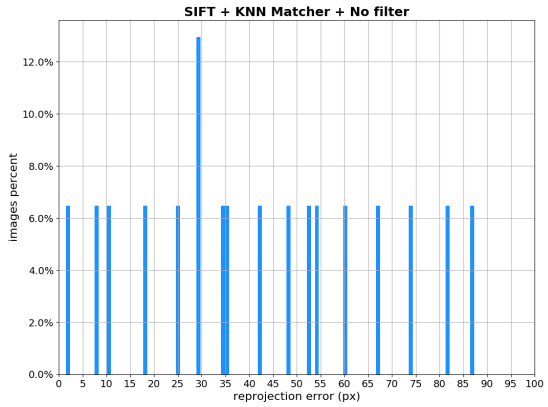


Figura 6.5: Histograma del error

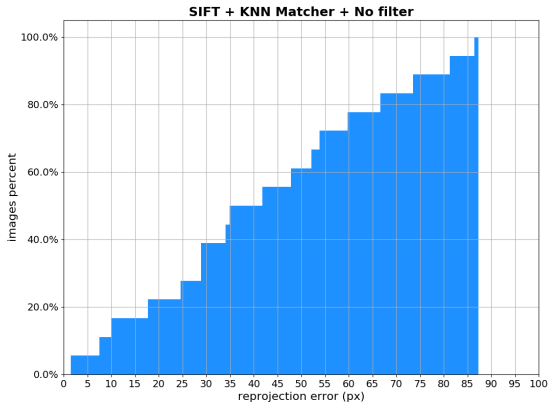


Figura 6.6: Histograma acumulado del error

Con este método se han conseguido estimar correctamente el 6 % de las poses, el restante 94 % o son estimaciones inválidas debidas a algún error en el proceso o son estimaciones con un error de reproyección superior a 100 píxeles. Del 6 % de las estimaciones correctas, se ha obtenido un error de reproyección menor a 5 píxeles en tan solo un 7 % (ver Figura 6.5) de las estimaciones.

6.1.1.2. Filtro de sobre-saliencia

En este segundo experimento se añade el filtrado de sobre-saliencia, explicado en la sección 5.3.4.1.

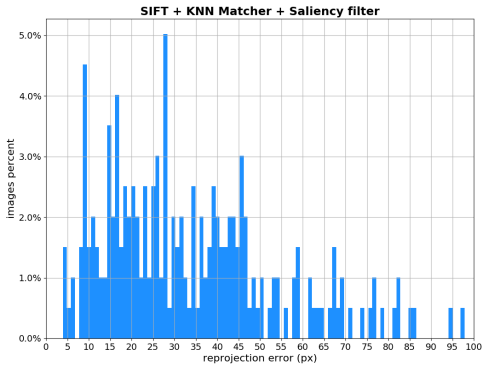


Figura 6.7: Histograma del error

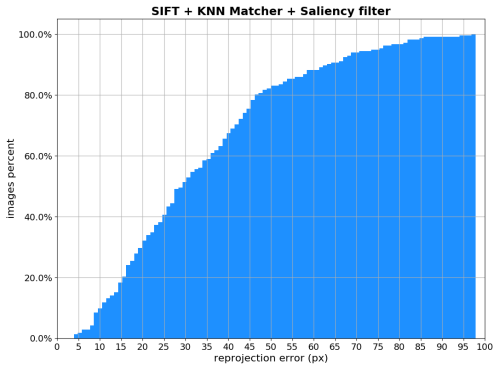


Figura 6.8: Histograma acumulado del error

Aplicando el filtro de sobre-saliencia a los emparejamientos se ha conseguido estimar correctamente un 77 % de las poses, sin embargo el número de estimaciones con un error de reproyección inferior a 5 píxeles se ha reducido hasta apenas un 2 %. Esto pone de manifiesto el funcionamiento del filtro de sobre-saliencia,

es probable que algunos de los emparejamientos eliminados por este filtrado aportasen la información necesaria para realizar estimaciones de mayor calidad, pero como no tenemos una manera segura de comprobar qué emparejamientos mejoran la estimación se descartan antes de introducir emparejamientos erróneos al sistema.

6.1.1.3. Filtro de sobre-saliencia y matriz fundamental

Para este experimento se introducen restricciones geométricas en el filtrado de emparejamientos, además del filtro de sobre-saliencia.

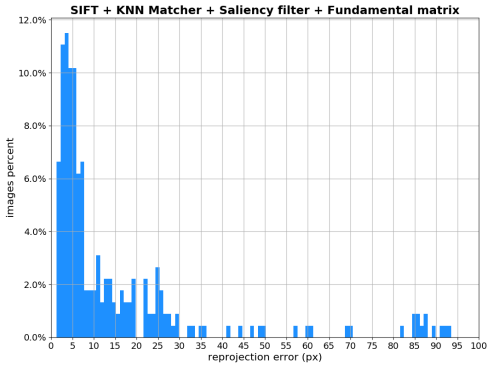


Figura 6.9: Histograma del error

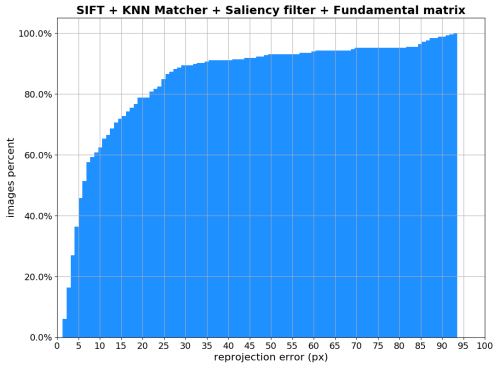


Figura 6.10: Histograma acumulado del error

El uso de restricciones geométricas mejora considerablemente las estimaciones del algoritmo, con este método el algoritmo es capaz de estimar correctamente un 89 % de las estimaciones. De ese 89 % se han realizado estimaciones con un error de reproyección inferior a 5 píxeles en un 35 % de los casos (ver Figura 6.10).

6.1.1.4. Filtro de sobre-saliencia y homografía

Para el último experimento los emparejamientos se filtran con un filtro de sobre-saliencia y el filtrado por homografía.

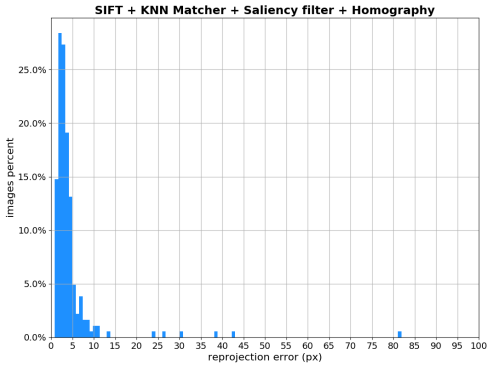


Figura 6.11: Histograma del error

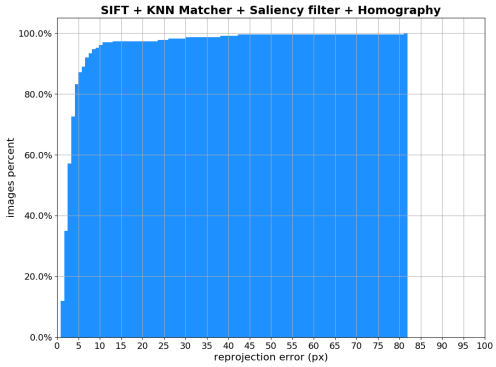


Figura 6.12: Histograma acumulado del error

Añadiendo el filtrado por homografía se consiguen los mejores resultados para este escenario. Aunque el porcentaje de estimaciones correctas es inferior (82 %) con respecto del obtenido con el filtrado con matriz fundamental (89 %), las estimaciones con un error de reproyección menor de 5 píxeles aumentan de un 35 % hasta casi un 85 %.

6.1.2. Escenario 2

El escenario para este experimento consiste en un despacho que contiene una mesa de escritorio, distintas estanterías y un ventana por donde entra luz natural. Además, durante la realización del mapa se añadió una fuente de luz incandescente. El mapa resultante, que sirve como entrada al algoritmo, contiene 140 imágenes.

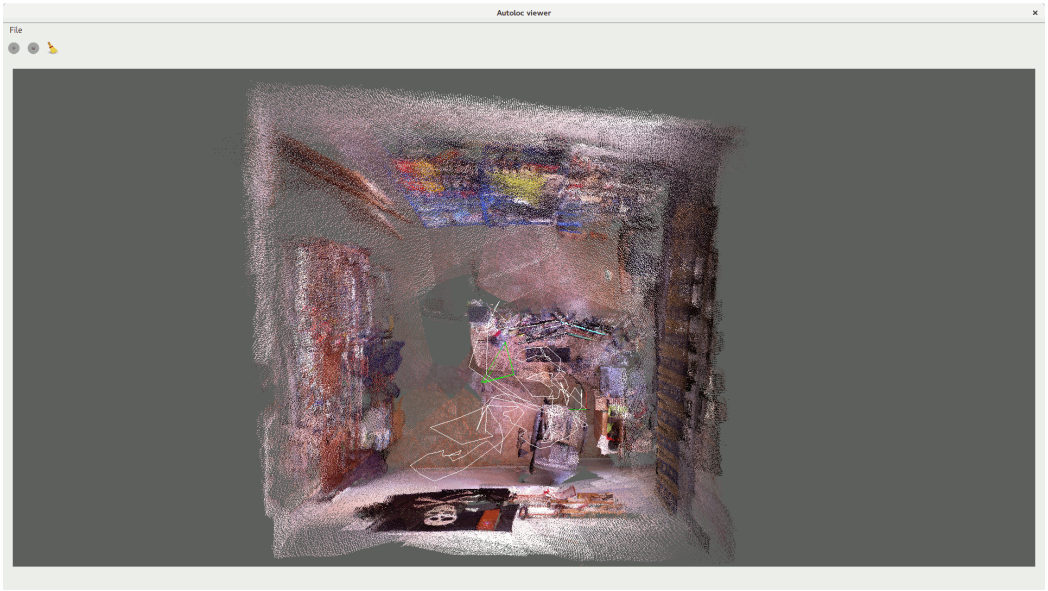


Figura 6.13: Vista superior del mapa de un despacho

De nuevo, la trayectoria de la cámara (en gris) y la última posición de la cámara (polígono verde) indican la manera en la que se realizó el mapa. Desde la vista superior del despacho (ver Figura 6.13) se aprecia cómo la equina superior izquierda no está totalmente alineada con el resto. El mapa se realizó de manera rápida, sin recorrer todos los espacios del despacho realizando algunas pasadas para asegurar el cierre de bucle. De la Figura 6.13 llama la atención, cómo con tan sólo unas pocas pasadas, *RTAB-Map* es capaz de generar mapas con tanto nivel de detalle, justo a la derecha de la bandera se puede apreciar una cartulina sobre un corcho, la deformación de la cartulina no es debido a ningún error en la generación del mapa, la cartulina se encuentra así.

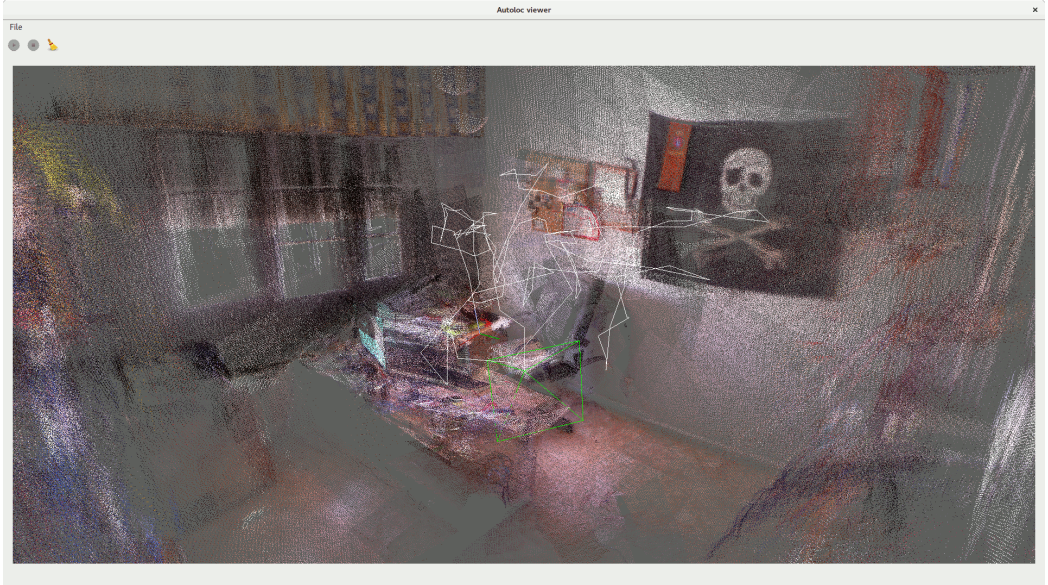


Figura 6.14: Vista lateral del mapa de un despacho

6.1.2.1. Sin filtrado

Con las estimaciones realizadas con los emparejamientos sin filtrar se obtienen los resultados mostrados por las Figuras 6.15 y 6.16. De un modo similar al Escenario 1, los resultados de las estimaciones no son satisfactorios son muy pobres.

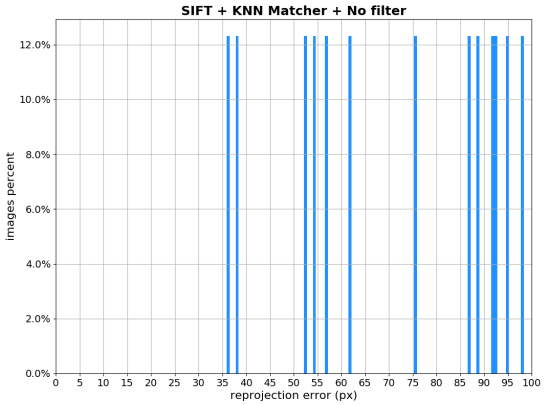


Figura 6.15: Histograma del error

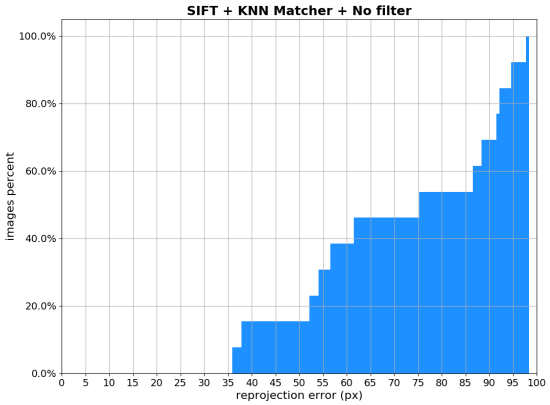


Figura 6.16: Histograma acumulado del error

En este caso se han realizado correctamente un 9 % de las estimaciones. De ellas no se ha conseguido realizar ninguna estimación con un error de reproyección inferior a 5 píxeles. Para realizar una estimación de buena calidad (error de reproyección inferior a 5 píxeles) es fundamental seleccionar correctamente la imagen más parecida a la imagen de entrada. El algoritmo determina cuál es la imagen más parecida en base al número de emparejamientos obtenidos entre dos imágenes. Al no existir ningún tipo de filtrado no podemos determinar la calidad de los emparejamientos. Normalmente esto provoca que se escoja una

imagen que en realidad no se parece a la de entrada, lo que provoca a su vez que la estimación acabe fallando o se haga una estimación de mala calidad.

6.1.2.2. Filtro de sobre-saliencia

Con el filtro de sobre-saliencia añadimos restricciones sobre la apariencia de los descriptores, eliminando emparejamientos que puedan generar incertidumbre en la estimación de la pose.

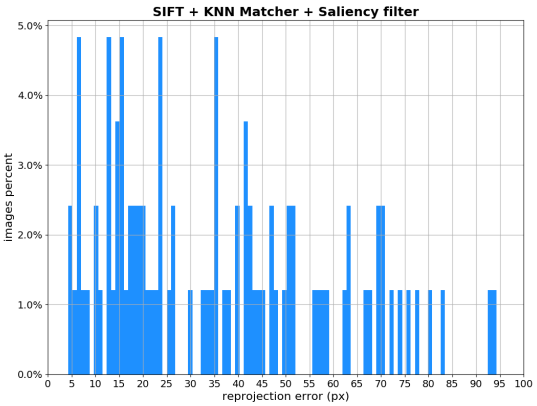


Figura 6.17: Histograma del error

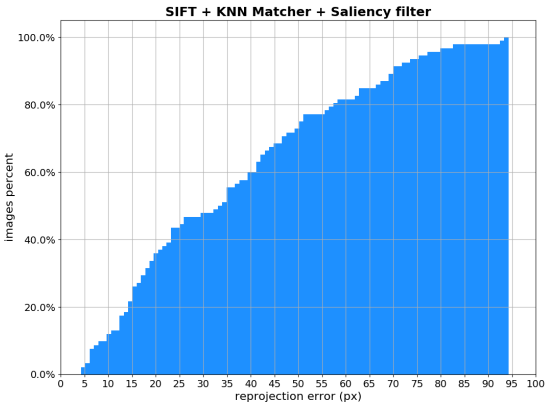


Figura 6.18: Histograma acumulado del error

Para este escenario y con este filtro se consigue estimar correctamente un 65 % de las estimaciones. De esas estimaciones alrededor de un 2 % se realiza con un error de reproyección inferior a 5 píxeles.

6.1.2.3. Filtro de sobre-saliencia y matriz fundamental

A continuación se muestran los resultados obtenidos añadiendo el filtrado de la matriz fundamental.

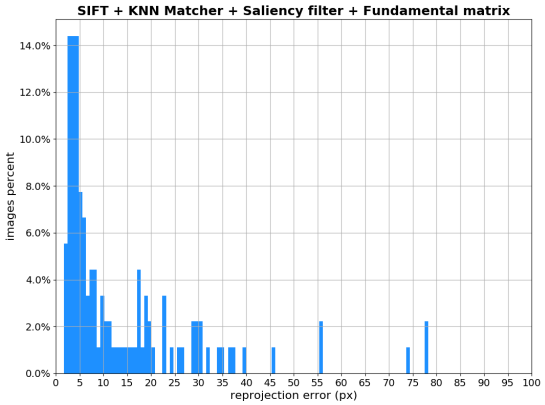


Figura 6.19: Histograma del error

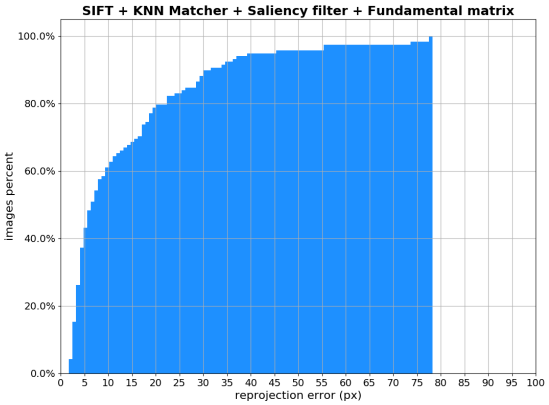


Figura 6.20: Histograma acumulado del error

Con las restricciones epipolares nos aseguramos que los emparejamientos cumplen con dicha propiedad geométrica y eso se traduce en una mejora de los resultados. Ahora somos capaces de estimar correctamente

el 84 % de las estimaciones, y somos capaces de realizar estimaciones con un error de reproyección por debajo de 5 píxeles alrededor de un 38 % de los casos.

6.1.2.4. Filtro de sobre-saliencia y homografía

De nuevo, los mejores resultados se obtienen añadiendo un filtrado por homografía. Para este caso el algoritmo ha sido capaz de estimar correctamente el 77 % de las estimaciones. De ellas, alrededor de un 88 % han sido con errores de reproyección por debajo de 5 píxeles. Estos resultados son muy parecidos a los obtenidos en el escenario anterior.

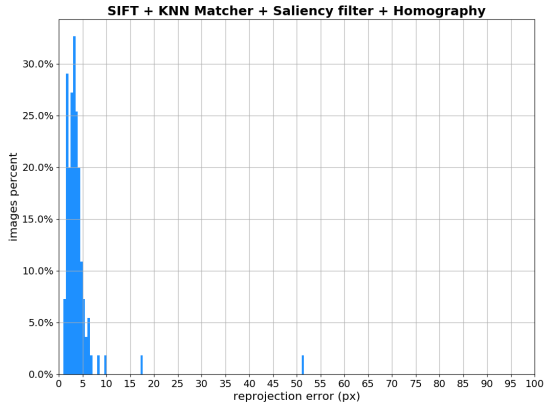


Figura 6.21: Histograma del error

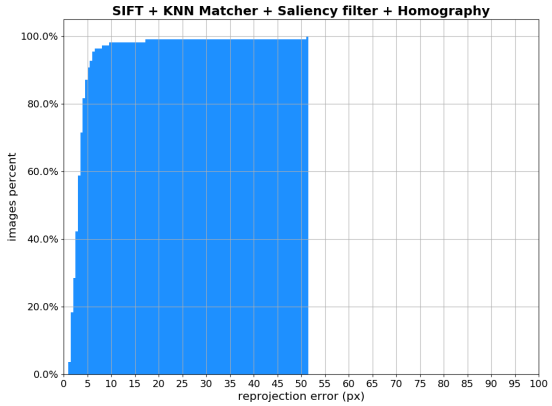


Figura 6.22: Histograma acumulado del error

6.1.3. Escenario 3

Para este escenario se realizó un mapa 3D de un despacho de la URJC en el campus de Fuenlabrada, el despacho cuenta con dos mesas de escritorio, diversas estanterías y pizarras, y una mesa.

Figura 6.23: Vista superior de un despacho de la URJC del campús de Fuenlabrada

Dispone de un ventanal por donde entra luz natural y fluorescentes como fuente de luz artificial. El mapa contiene 295 imágenes. Como muestra la Figura 6.23, durante la generación del mapa también se capturó el techo y se realizó de una manera más minuciosa capturando casi todos los rincones del despacho. Esto se aprecia en las líneas rectas que forman las paredes, que aún cometiendo algún error, como la parte del techo que queda fuera de la estructura principal, ha sido posible realizar un mapa tan preciso como el expuesto en la Figura 6.24.

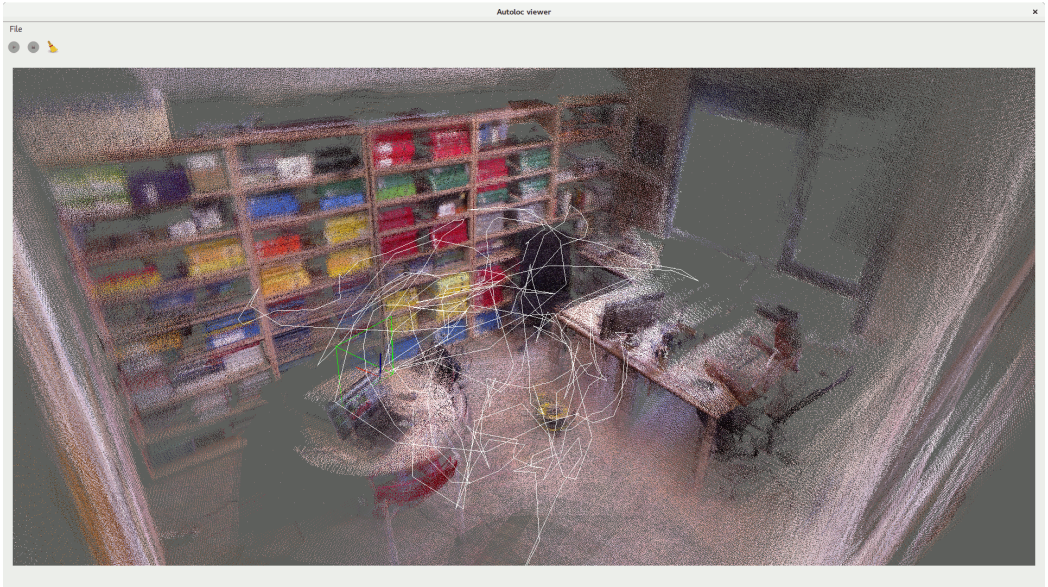


Figura 6.24: Vista lateral de un despacho de la URJC del campus de Fuenlabrada

6.1.3.1. Sin filtrado

Una vez más, para este método las estimaciones son muy poco precisas, tal como muestran las Figuras 6.25 y 6.26. Se han estimado correctamente un 20 % de las estimaciones, esto supone un incremento considerable respecto al mismo método en los anteriores escenarios. Probablemente esto se deba a la realización del mapa, en este caso se realizó de una manera más exhaustiva, llegando a todos los rincones de la estancia realizando varias pasadas. Esto ayuda a que el grafo que forma el mapa contenga más vistas y que éstas sean de mayor calidad (proporcionen más información) al haber más solapamiento entre ellas.

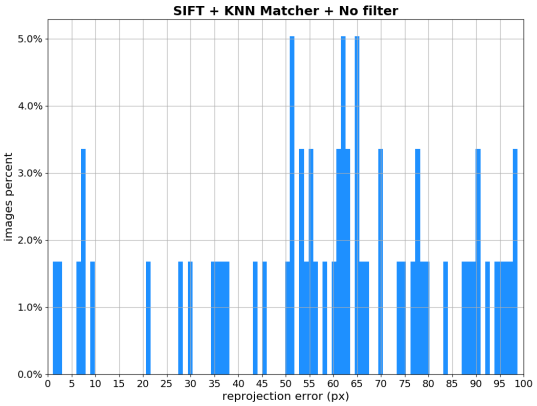


Figura 6.25: Histograma del error

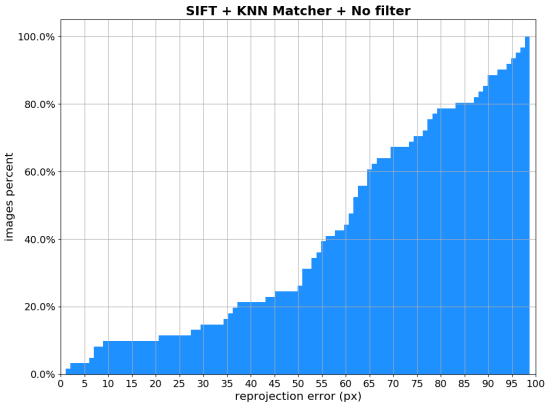


Figura 6.26: Histograma acumulado del error

6.1.3.2. Filtro de sobre-saliencia

Con este filtro podemos apreciar (ver Figura 6.27) cómo ahora se consiguen realizar algunas estimaciones con errores de reproyección más bajos. Para este escenario se estimaron correctamente un 63 % de las estimaciones, obteniendo alrededor de un 5 % con un error de reproyección inferior a 5 píxeles.

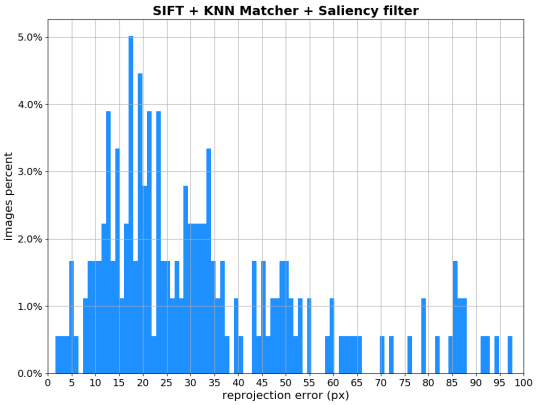


Figura 6.27: Histograma del error

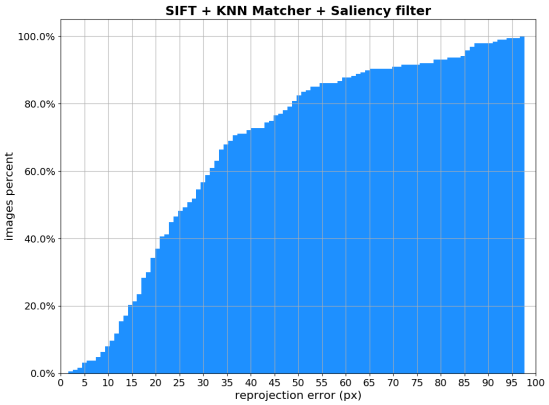


Figura 6.28: Histograma acumulado del error

6.1.3.3. Filtro de sobre-saliencia y matriz fundamental

Tener la posibilidad de poder buscar la correspondencia de un descriptor en una única línea de la otra imagen nos permite, además de ahorrar tiempo de cómputo, agregar restricciones a nuestro algoritmo.

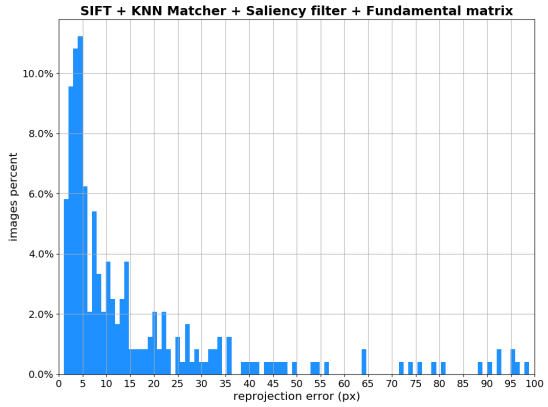


Figura 6.29: Histograma del error

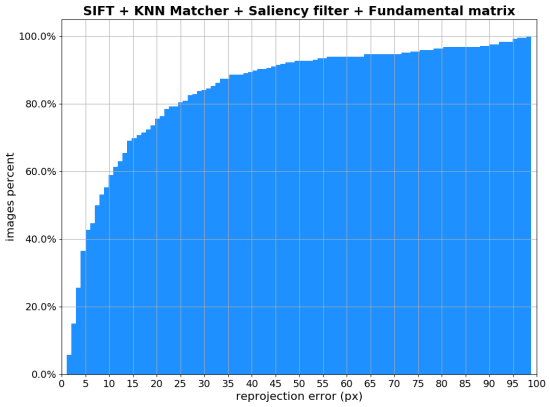


Figura 6.30: Histograma acumulado del error

En este escenario, aplicando las restricciones epipolares, conseguimos estimar correctamente un 83 % de las estimaciones. De ese 83 % somos capaces de estimar alrededor de un 39 % de las estimaciones con un error de reproyección inferior a 5 píxeles.

6.1.3.4. Filtro de sobre-saliencia y homografía

Para el último experimento sobre este escenario se añade al filtro de sobre-saliencia un filtrado por homografía sobre los emparejamientos. Obteniendo los resultados que se pueden apreciar en las Figuras 6.31 y 6.32.

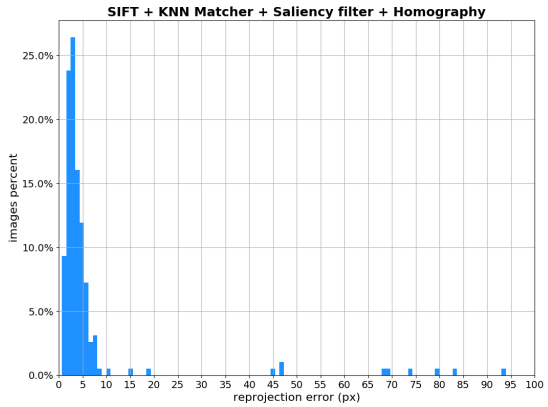


Figura 6.31: Histograma del error

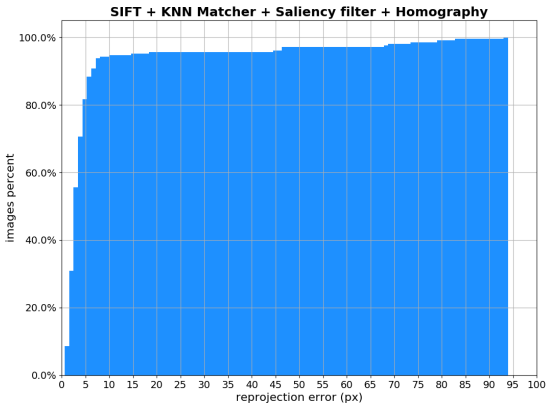


Figura 6.32: Histograma acumulado del error

Durante los anteriores experimentos el filtrado por homografía ha demostrado su buen funcionamiento a la hora de filtrar emparejamientos. Para este caso, el número de estimaciones correctas es de un 70 %, obteniendo estimaciones con un error de reproyección inferior a 5 píxeles en un 82 % de los casos. De nuevo el número de estimaciones correctas disminuye con respecto del filtrado con matriz fundamental, pero las estimadas son más precisas con la homografía.

6.1.4. Escenario 4

Para este último escenario se eligió un despacho multipuesto de la biblioteca del campus de Fuenlabrada. En él tenemos diversos puestos de trabajo y un ventanal protegido con cortinas por donde entra escasamente la luz natural, todo el despacho está iluminado con fluorescentes. El mapa generado contiene 213 imágenes, además de sus poses e imágenes de profundidad.

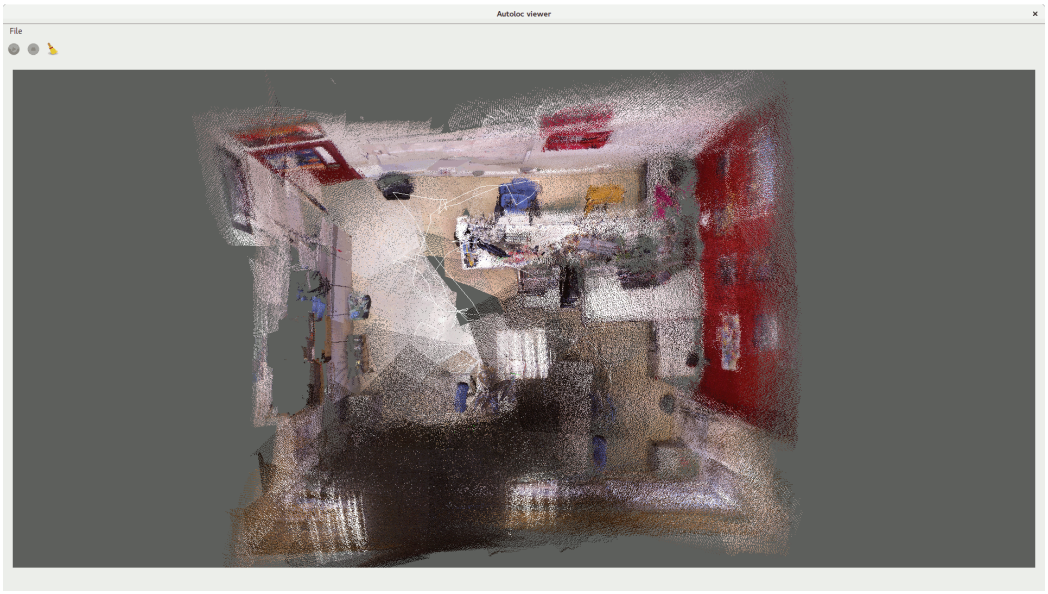


Figura 6.33: Vista superior de un despacho de la biblioteca del campus de Fuenlabrada

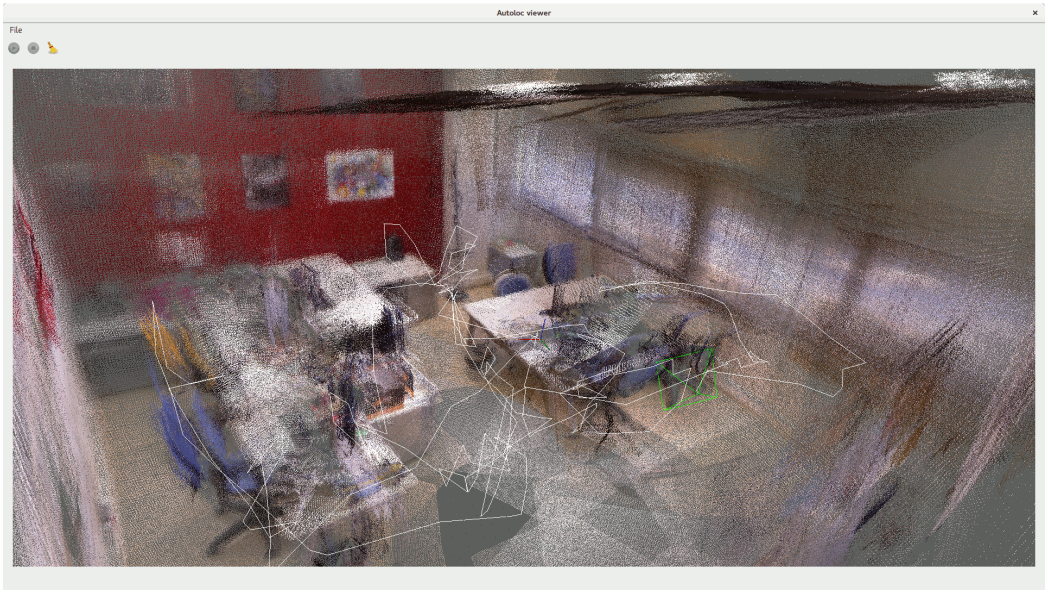


Figura 6.34: Vista lateral de un despacho de la biblioteca del campus de Fuenlabrada

Mientras se realizaba el mapa se intentó capturar el techo del despacho, pero después de varios intentos tuvimos que desistir ya que *RTAB-Map* no conseguía añadir al mapa los fotogramas capturados. Aunque en apariencia el mapa ha sido correctamente generado, la posición de la última cámara no está situada

correctamente. Una vez se hubo acabado de realizar el mapa, el sensor RGB-D se posó sobre la mesa. Sin embargo, el polígono verde que representa la cámara (ver Figura 6.34) se encuentra por debajo de la mesa y apuntando hacia el suelo. Se decidió añadir este mapa a los experimentos para comprobar el rendimiento real del algoritmo en un mapa que contiene errores.

6.1.4.1. Sin filtrado

En cuanto a los resultados de las estimaciones sin hacer uso de ningún filtrado de emparejamientos, los resultados son similares a los obtenidos en el resto de escenarios.

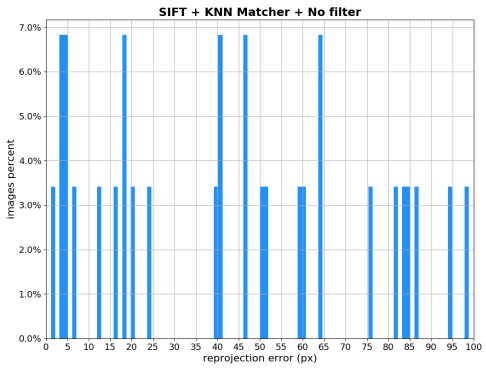


Figura 6.35: Histograma del error

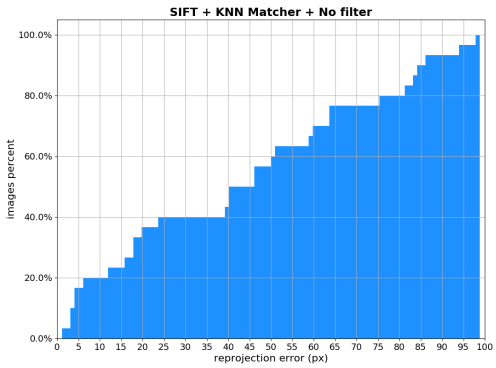


Figura 6.36: Histograma acumulado del error

6.1.4.2. Filtro de sobre-saliencia

El filtro de sobre-saliencia consigue una vez más aumentar el número de estimaciones correctas pasando de un 14 % a un 53 %.

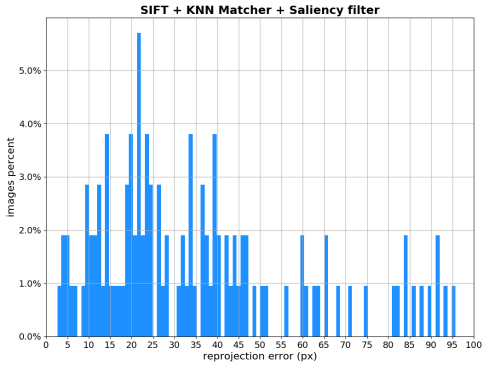


Figura 6.37: Histograma del error

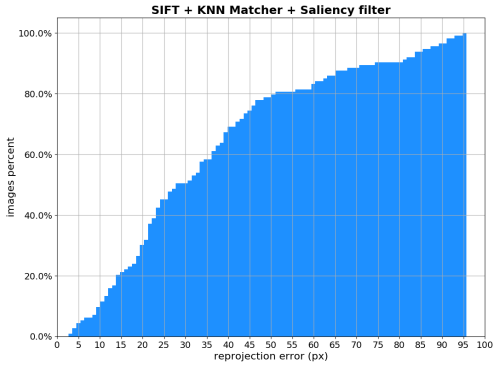


Figura 6.38: Histograma acumulado del error

6.1.4.3. Filtro de sobre-saliencia y matriz fundamental

Añadiendo restricciones geométricas observamos cómo mejora la precisión de las estimaciones. Es algo que ya habíamos observado en los anteriores escenarios.

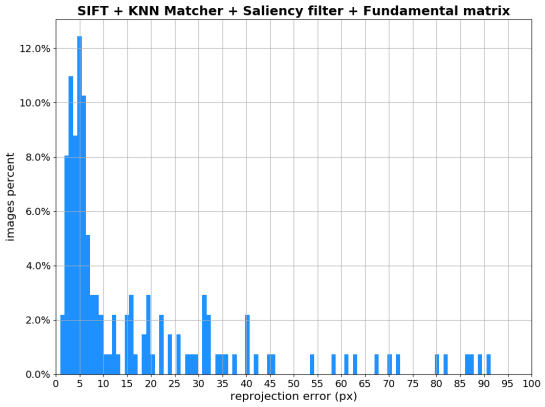


Figura 6.39: Histograma del error

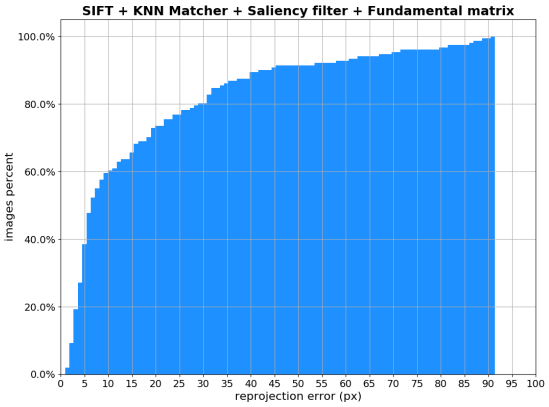


Figura 6.40: Histograma acumulado del error

Vemos cómo no sólo se incrementa precisión de las estimaciones, si no que además, conseguimos más estimaciones correctas (70 %) reduciendo el número de estimaciones inválidas. De nuevo, es algo que venimos observando en los escenarios anteriores.

6.1.4.4. Filtro de sobre-saliencia y homografía

Por último tenemos los resultados de las estimaciones aplicando un filtrado por homografía sobre este escenario.

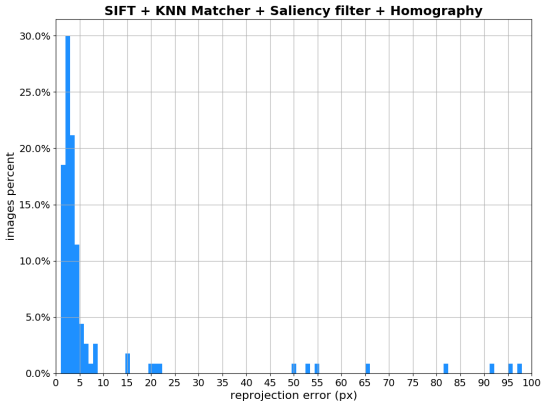


Figura 6.41: Histograma del error

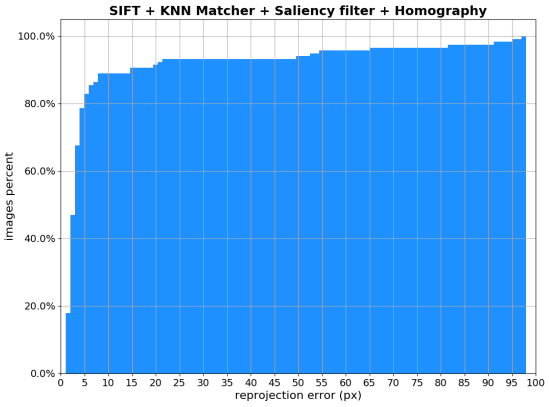


Figura 6.42: Histograma acumulado del error

El número de estimaciones correctas es de 54 %, algo inusual con respecto los anteriores métodos. De nuevo la calidad del mapa generado juega un papel fundamental. El mapa en este escenario se realizó de una manera menos exhaustiva que los anteriores, de hecho tal y como se comentó al inicio de esta sección, el mapa contiene errores. Probablemente el algoritmo de cierre de bucle no fue capaz de determinar con exactitud los *Keyframes* del mapa, lo que provocó que la optimización del grafo no pudiera ajustar correctamente las poses de las cámaras. Aún así, de las estimaciones realizadas correctamente alrededor de un 78 % se ha realizado con un error de reproyección inferior a 5 píxeles.

6.1.5. Resumen del efecto de los filtros de emparejamientos

A continuación se muestran, a modo de resumen, los datos obtenidos en todos los escenarios donde se ha realizado el experimiento.

	% estimaciones correctas	% estimaciones incorrectas	% error de reproyección < 5 px
Sin filtro	6,57	91,43	6
Sobre-saliencia	77,37	22,63	2
Matriz fundamental	89,42	10,58	35
Homografía	82,48	17,52	85

Cuadro 6.1: Escenario 1

	% estimaciones correctas	% estimaciones incorrectas	% error de reproyección < 5 px
Sin filtro	9,29	90,71	0
Sobre-saliencia	65,71	34,29	2
Matriz fundamental	84,29	15,71	38
Homografía	77,86	22,14	88

Cuadro 6.2: Escenario 2

	% estimaciones correctas	% estimaciones incorrectas	% error de reproyección < 5 px
Sin filtro	20,68	79,32	5
Sobre-saliencia	63,39	36,61	63
Matriz fundamental	83, 39	16,61	39
Homografía	70,17	29,83	82

Cuadro 6.3: Escenario 3

	% estimaciones correctas	% estimaciones incorrectas	% error de reproyección < 5 px
Sin filtro	14,08	85,92	18
Sobre-saliencia	53,05	46,95	5
Matriz fundamental	70, 90	29,10	38
Homografía	54,93	45,07	78

Cuadro 6.4: Escenario 4

De este resumen se pueden extraer varias conclusiones. Primero, el algoritmo de autolocalización desarrollado ha sido validado experimentalmente, pues en su mejor configuración proporciona unas estimaciones de posición con muy poco error de reproyección. Segundo, la mejor configuración es la que filtra los posibles emparejamientos combinando la sobre-saliencia con la restricción geométrica de la homografía. Tercero, la calidad del mapa influye en la precisión del algoritmo de autolocalización.

6.2. Comparación directa de los métodos de filtrado de emparejamientos

En esta sección se muestran de manera gráfica los resultados de los distintos tipos de filtrados aplicados en la etapa de emparejamientos. En las Figuras, la imagen izquierda es la imagen del mapa que más se parece a la imagen de entrada (la de la derecha) según el método de filtrado aplicado. Las líneas naranjas unen los *KeyPoints* emparejados en ambas imágenes, los puntos rojos son la reproyección del punto 3D del *KeyPoint* calculado con la cámara que utilizamos como verdad absoluta (que se encuentra en el mapa inicial), el punto azul es la reproyección del punto 3D del *KeyPoint* calculado con la cámara estimada y la línea verde muestra el error de reproyección cometido.

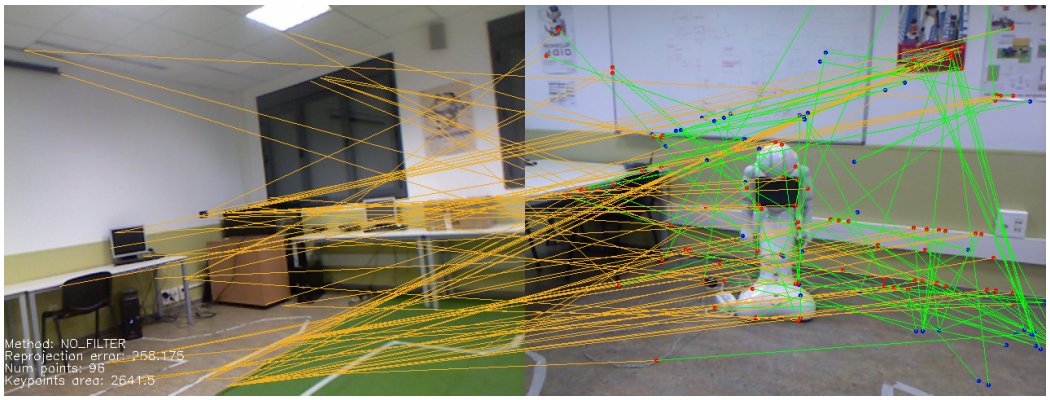


Figura 6.43: Emparejamientos sin filtrado

En la Figura 6.43 podemos ver los emparejamientos sin aplicar ningún filtro en ellos. Salta a la vista como la imagen con más emparejamientos no se parece en nada a la imagen de entrada (imagen de la derecha). La cámara estimada comete un error de reproyección de 258 píxeles, por lo que, es tratada como un error del algoritmo.

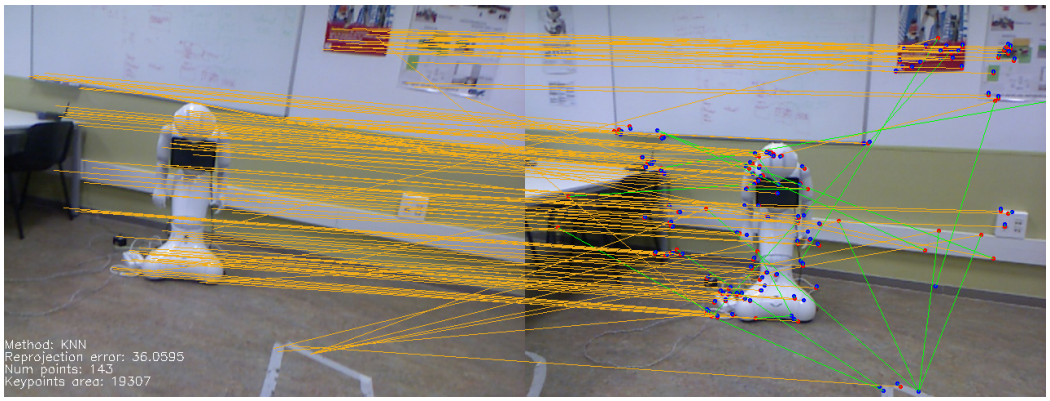


Figura 6.44: Emparejamientos con filtro de sobre-saliencia

Aplicando el filtro de sobre-saliencia se encuentra correctamente la imagen más parecida a la imagen de entrada (ver Figura 6.44). Sin embargo, se comete un error de proyección alto (36 píxeles) provocado por el mal emparejamiento de algunos de los *KeyPoints*.

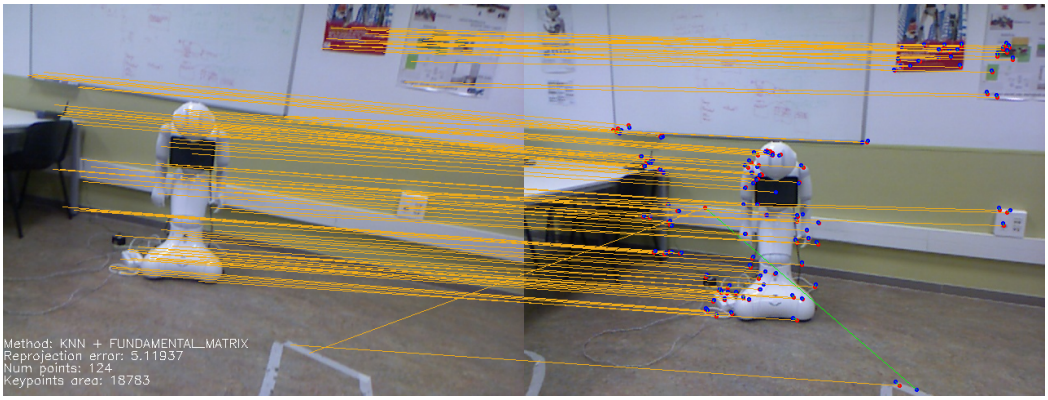


Figura 6.45: Emparejamientos con filtro de sobre-saliencia y matriz fundamental

Añadiendo el filtrado con la matriz fundamental conseguimos reducir el error de reproyección (se realiza una mejor estimación de la pose 3D). En este caso se ha cometido un error de 5 píxeles en un total de 174 emparejamientos.

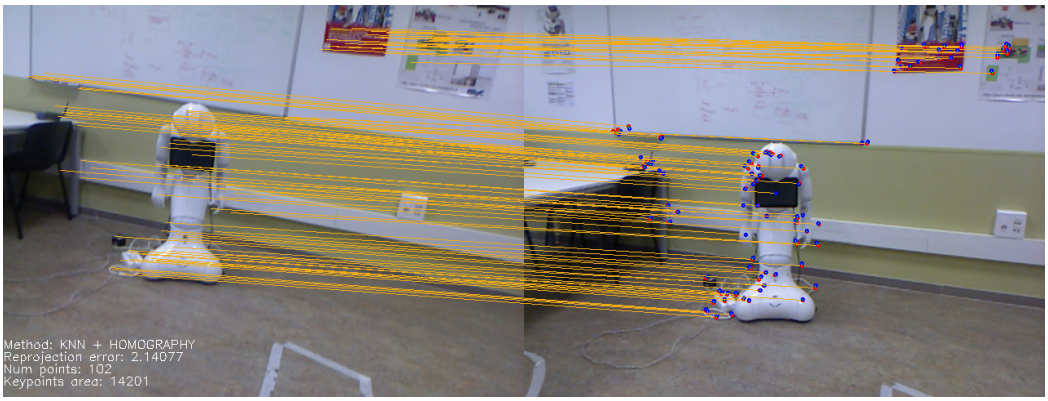


Figura 6.46: Emparejamientos con filtro de sobre-saliencia y homografía

Finalmente, y tal como muestran los experimentos realizados, el mejor resultado se obtiene añadiendo el filtrado por homografía. Este método ha cometido un error de reproyección de 2 píxeles en 102 emparejamientos. El descenso de emparejamientos demuestra que el filtrado por homografía elimina más emparejamientos incorrectos que el cálculo de la matriz fundamental.

Capítulo 7

Conclusiones

Una vez descritos los objetivos y las soluciones alcanzadas, en este capítulo se resumen las conclusiones principales de este trabajo y se proponen varias líneas por las que se puede extender.

7.1. Aportes

Se ha propuesto y programado un algoritmo de autolocalización visual 3D con sensores RGB-D que resuelve el problema de la calibración de este tipo de sensores en entornos de trabajo conocidos.

Durante la realización de este Trabajo Fin de Máster se han estudiado y comprendido distintas técnicas de autolocalización visual como las expuestas en el capítulo 3. Se evaluó y se estudió en profundidad los trabajos de Mathieu Labbé y François Michaud (*Online Global Loop Closure Detection for Large-Scale Multi-Session Graph-Based SLAM*[36] y *Memory Management for Real-Time Appearance-Based Loop Closure Detection*[34]) de manera que se obtuvieron los conocimientos suficientes como para, modificando el código fuente de la aplicación creada por los autores (`RTAB-Map`), extraer los elementos necesarios que sirvieron de apoyo fundamental en la consecución de este proyecto. En concreto para construir mapas 3D del entorno de trabajo utilizando un sensor RGB-D.

Se desarrolló una librería (`mapper` ver sección 5.3) que implementa el algoritmo propuesto. En esta librería se implementó su propio modelo de cámara que permite realizar operaciones como la proyección y reproyección de puntos o transformaciones geométricas, poniendo en práctica los conocimientos obtenidos durante el Máster. La librería hace uso de los estándares de la industria, como el sistema de extracción de características o el de emparejamiento, haciendo uso de tecnologías actuales y con un gran respaldo por parte de la comunidad. Además, se implementaron métodos de filtrado para mejorar la precisión del emparejamiento de características haciendo uso de algoritmos basados en apariencia y algoritmos basados en restricciones geométricas. Se usaron algoritmos para calcular la posición 3D de la cámara y se refinaron los resultados mediante técnicas de optimización.

Siguiendo los estándares de la industria se realizo un conjunto de test unitarios que demuestran el correcto funcionamiento de la solución implementada. Gracias a su enfoque basado en la programación

orientada a objetos, es una librería fácilmente ampliable, mantenible y escalable, de una manera simple es posible añadir los últimos métodos de extracción de características o los últimos algoritmos de emparejamiento. Al trabajar con los estándares de la comunidad, es fácil exportar los resultados a otra plataforma, o incluso de importar mapas construidos con otras soluciones. Su diseño en forma de librería permite utilizar esta solución en otras aplicaciones, ya sea haciendo uso de la propia librería o implementado un envoltorio de ella.

También se diseñaron y Se desarrollaron las herramientas de apoyo `localizer` y `autloc_viewer`. La primera de ellas, `localizer`, para la integración de la librería dentro del entorno JdeRobot. Gracias a este componente, cualquier usuario de JdeRobot, podrá hacer uso de la solución propuesta simplemente realizando peticiones a interfaces ICE o, si prefiere hacerlo *offline*, haciendo uso de la terminal de comandos. La segunda herramienta, `autoloc_viewer`, es una herramienta de depuración que permite visualizar el mapa donde trabaja el sensor RGB-D, las poses estimadas y la trayectoria de la cámara dentro del entorno. Para su desarrollo se han utilizado, de nuevo, estándares de la industria y las librerías más utilizadas por la comunidad. Permite su uso con interfaces ICE o mediante la línea de comandos.

El algoritmo propuesto fue validado experimentalmente mediante la realización de cuatro experimentos sobre escenarios distintos. Se implementaron distintas métricas de error para medir la calidad de las estimaciones realizadas por el algoritmo y se desarrolló un sistema de evaluación del algoritmo mediante test unitarios en la librería `mapper`. De las pruebas realizadas se desprende que el algoritmo, en su mejor combinación, proporciona unas estimaciones de posición con muy poco error de reproyección (menor a 5 píxeles). La mejor configuración es la que filtra los posibles emparejamientos combinando el filtro de sobre-saliencia con la restricción geométrica de la homografía. Además, la calidad del mapa influye en la precisión del algoritmo.

Todo el software desarrollado durante el proyecto será liberado en el repositorio *GitHub* de JdeRobot para que cualquier persona pueda hacer uso de él, pueda modificarlo o mejorarlo, o simplemente estudiarlo para comprender los detalles del algoritmo.

Por todo lo expuesto en los párrafos anteriores, se puede decir que se alcanzaron los objetivos propuestos (ver sección 2.2) en el Trabajo Fin de Máster. Se propuso una solución al problema planteado. Todo ello obteniendo unos resultados razonables tanto en calidad de las estimaciones como en tiempo de ejecución. Además, se aportó a la comunidad un nuevo mecanismo para la calibración de sensores RGB-D.

7.2. Líneas futuras

El algoritmo tiene margen de mejora, sobre todo en lo relacionado con el rendimiento computacional. Algunas partes pueden ser paralelizables e incluso es posible que algunas de ellas puedan correr sobre GPU para reducir los tiempos de computo. La búsqueda de imágenes dentro del mapa puede ser optimizada para no tener que recorrer todo el mapa, dividendo el mapa en regiones que permitan mantener un tiempo de búsqueda razonable.

También se pueden explorar el uso de otros descriptores distintos a SIFT para comprobar su eficacia frente a los resultados obtenidos con este descriptor.

No se descarta tampoco el uso de otra técnica para la estimación de la pose, que pueda mejorar los resultados actuales.

En cuanto al emparejamiento de descriptores, se podría explorar la vía de determinar cuál es el número mínimo de *KeyPoints* necesarios para hacer una estimación de calidad. Esto reduciría el tiempo de cómputo al realizar las estimaciones. Actualmente el algoritmo usa únicamente dos fotogramas para estimar la posición de la cámara, la imagen de entrada y la imagen más parecida dentro del mapa. Se podría añadir una tercera imagen (la segunda más parecida en el mapa) para estimar la posición. En este caso *SBA* podría realizar un aporte mayor al realizado únicamente con dos imágenes.

Bibliografía

[1] Manolis IA Lourakis Antonis A Argyros. Sba: A software package for generic sparse bundle adjustment. *ACM Transactions on Mathematical Software (TOMS)*, 2009. `http://dl.acm.org/citation.cfm?id=1486527`.

[2] Karsten Ottenberg Bert M. Haralick, Chung-Nan Lee and Michael Nölle. Review and analysis of solutions of the three point perspective pose estimation problem. *International journal of computer vision*, 1994. `http://haralick-org.torahcode.us/journals/three_point_perspective.pdf`.

[3] Richard I. Hartley Bill Triggs, Philip F. McLauchlan and Andrew W. Fitzgibbon. Bundle adjustment—a modern synthesis. *International workshop on vision algorithms*, 1999.

[4] John Canny. A computational approach to edge detection. *IEEE Transactions On Pattern Analysis and Machine Intelligence*, 1986.

[5] Matia Pizzoli Christian Forster and Davide Scaramuzza. Svo: Fast semi-direct monocular visual odometry. *Robotics and Automation*, 2014.

[6] Mark Cummins Paul M. Newman Christopher Mei, Gabe Sibley and Ian D. Reid. A constant-time efficient stereo slam system. *BMVC*, 2009.

[7] H. Schoppers Clark F. Olson, Larry H. Matthies and Mark W. Maimone. Robust stereo ego-motion for long distance navigation. *Computer Vision and Pattern Recognition*, 2000.

[8] Alejandro Hernández Cordero. Autolocalización visual aplicada a la realidad aumentada. *Trabajo fin de Máster*, 2013. `https://gsyc.urjc.es/jmplaza/students/tfm-visualslam-alex-2014.pdf`.

[9] Yazmin Lucy Cumberbich. 3d augmented reality system using jderobot. *Proyecto fin de carrera*, 2014. `http://svn.jderobot.org/users/ycumberbirch/pfc/trunk/memoria/thesis.pdf`.

[10] Adnan Darwiche. Modeling and reasoning with bayesian networks. *Cambridge University Press*, 2009.

[11] Oleg Naroditsky David Nistér and James Bergen. Visual odometry. *Computer Vision and Patter Recognition*, 2004.

[12] Andre Davison. 15 years of visual slam. *diapositivas*, 2015. `http://wp.doc.ic.ac.uk/thefutureofslam/wp-content/uploads/sites/93/2015/12/slides_ajd.pdf`.

[13] Andrew J. Davison. Slam with a single camera. *Workshop on Concurrent Mapping and Localization for Autonomous Mobile Robots*, 2002.

[14] Andrew J. Davison. Real-time simultaneous localisation and mapping with a single camera. *ICCV*, 2003.

[15] Dieter Fox Dirk Hahnel, Wolfram Burgard and Sebastian Thrun. An efficient fastslam algorithm for generating maps of large-scale cyclic environments from raw laser range measurements. *Intelligent Robots and Systems*, 2003.

[16] J. R Bergen P. J. Burt E. H. Adelson, C. H. Anderson and J.M Ogden. Pyramid methods in image processing. *RCA Engineer*, 1984.

[17] Kurt Konolige Ethan Rublee, Vincent Rabaud and Gary Bradski. Orb: An efficient alternative to sift or surf. *Computer Vision (ICCV)*, 2011.

[18] Michel Dhome Fabien Dekeyser Etienne Mouragnon, Maxime Lhuillier and Patrick Sayd. Real time localization and 3d reconstruction. *Computer Vision and Pattern Recognition*, 2006. `http://maxime.lhuillier.free.fr/pCvpr06.pdf`.

[19] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 1981. `https://www.sri.com/sites/default/files/publications/ransac-publication.pdf`.

[20] C. Stachniss P. Pfaff G. Grisetti, S. Grzonka and W. Burgard. Efficient estimation of accurate maximum likelihood maps in 3d. *EEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2007.

[21] Davide Scaramuzza Gabriel Nützi, Stephan Weiss and Roland Siegwart. Fusion of imu and vision for absolute scale estimation in monocular slam. *Journal of intelligent and robotic systems*, 2011.

[22] Eduardo Perdices García. Técnicas para la localización visual robusta de robots en tiempo real con y sin maps. *Tésis doctoral*, 2017. `https://gsyc.urjc.es/jmplaza/students/phd-eduardo_perdices-2017.pdf`.

[23] Javier García and Zeev Zalevsky. Range mapping using speckle decorrelation.

[24] Dorian Gálvez-López and Juan D. Tardos. Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 2012.

[25] Christopher G. Harris and JM Pike. 3d positional integration from image sequences. *Image and Vision Computing*, 1988.

[26] Richard Hartley and Andrew Zisserman. Multiple view geometry in computer vision. *Cambridge university press*, 2003.

[27] Christopher Engels Henrik Stewenius and David Nistér. Recent developments on direct relative orientation. *ISPRS Journal of Photogrammetry and Remote Sensing*, 2006.

[28] Tinne Tuytelaars Herbert Bay and Luc Van Gool. Surf: Speeded up robust features. *Computer Vision and Image Understanding*, 2006.

[29] Joel A. Hesch and Stergios I. Roumeliotis. A direct least-squares (dls) method for pnp. *Computer Vision (ICCV), 2011 IEEE International Conference*, 2011. `http://www-users.cs.umn.edu/~stergios/papers/ICCV-11-DLS-PnP.pdf`.

[30] Yanis Pavlidis Jean-Philippe Tardif and Kostas Daniilidis. Monocular visual odometry in urban environments using an omnidirectional camera. *Intelligent Robots and Systems*, 2008. `https://pdfs.semanticscholar.org/10a5/f5bd0f35a80f0c49b4ed812aec5af86bf955.pdf`.

[31] Carl T. Kelley. Iterative methods for optimization. *SIAM*, 1999.

[32] Georg Klein and David Murray. Parallel tracking and mapping for small ar workspaces. *Mixed and Augmented Reality*, 2007.

[33] D. Koller and N. Friedman. Probabilistic graphical models: Principles and techniques. *MIT Press*, 2009.

[34] Mathieu Labbé and François Michaud. Memory management for real-time appearance-based loop closure detection. *Intelligent Robots and Systems (IROS 2011)*, 2011.

[35] Mathieu Labbé and François Michaud. Appearance-based loop closure detection for online large-scale and long-term operation. *IEEE Transactions on Robotics*, 2013.

[36] Mathieu Labbé and François Michaud. Online global loop closure detection for large-scale multi-session graph-based slam. *Intelligent Robots and Systems (IROS 2014)*, 2014.

[37] Kuen-Han Lin and Chieh-Chih Wang. Stereo-based simultaneous localization, mapping and moving object tracking. *Intelligent Robots and Systems (IROS)*, 2010.

[38] H. Christopher Longuet-Higgins. A computer algorithm for reconstructing a scene from two projections. *Readings in Computer Vision: Issues, Problems, Principles, and Paradigms*, 1987.

[39] Steven Lovegrove and Andrew J. Davison. Real-time spherical mosaicing using whole image alignment. *ECCV*, 2010.

[40] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 2004.

[41] F. Lu and E. Milios. Globally consistent range scan alignment for envrionment mapping. *paper*, 1987.

[42] Alberto López-Cerón. Autolocalización visual robusta basada en marcadores. *Trabajo fin de Máster*, 2015. `https://svn.jderobot.org/users/alopezceron/tfm/trunk/memoria/memoria.pdf`.

[43] Larry Matthies and STEVENA Shafer. Error modeling in stereo navigation. *IEEE Journal on Robotics and Automation*, 1987. `http://www.ri.cmu.edu/pub_files/pub3/matthies_l_1987_1/matthies_l_1987_1.pdf`.

[44] Shawn McCann. 3d reconstruction from multiple images. 2015.

[45] Michael J. Milford and Gordon F. Wyeth. Single camera vision-only slam on a suburban road network. *Robotics and Automation*, 2008. `http://ftp.itam.mx/pub/alfredo/ROBOTICS/RatSLAM/icra2008_milford.pdf`.

[46] Marius Muja and David G. Lowe. Fast approximate nearest neighbots with automatic algorithm configuration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2009.

[47] Raul Mur-Artal and Juan D. Tardos. Orb-slam2: an open-source slam system for monocular, stereo and rgb-d cameras. *arXiv preprint*, 2016.

[48] Daniel Martín Organista. Odometría visual con sensores rgbd. *Proyecto fin de carrera*, 2013. `https://svn.jderobot.org/users/dmartino/pfc/trunk/MEMORIA/pfc.pdf`.

[49] John J. Leonard Paul M. Newman and Richard J. Rikoski. Towards constant-time slam on an autonomous underwater vehicle using synthetic aperture sonar. *Robotics Research*, 2003.

[50] Daniel Azuara Pérez. Realidad aumentada con interacción física desde una cámara móvil usando jderobot. *Proyecto fin de carrera*, 2014. `http://svn.jderobot.org/users/dazuara/pfc/trunk/MEMORIA/memoria.pdf`.

[51] Olivier Leboulleux Radu Horaud, Bernard Conio and Bernard Lacolle. An analytic solution for the perspective 4-point problem. *Computer Vision, Graphics, and Image Processing*, 1989. `https://hal.archives-ouvertes.fr/inria-00589992/document`.

[52] Luis Miguel López Ramos. Autolocalización en tiempo real mediante seguimiento visual monocular. *Proyecto fin de carrera*, 2009. `https://gsyc.urjc.es/jmplaza/students/pfc-monoslam-2010.pdf`.

[53] Jose María Martínez Montiel Raul Mur-Artal and Juan D. Tardos. Orb-slam: a versatile and accurate monocular slam system. *EEE Transactions on Robotics*, 2015.

[54] Steven Lovegrove Andrew J. Davison Richard A. Newcombe. Dtam: Dense tracking and mapping in real-time. *ICCV*, 2011.

[55] Georg Klein Robert O. Castle and David W. Murray. Combining monoslam with object recognition for scene augmentation using a wearable camera. *Image and Vision Computing*, 2010.

[56] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. *Computer vision–ECCV*, 2006.

[57] Radu Bogdan Rusu and Steve Cousins. 3d is here: Point cloud library (pcl). *IEEE International Conference on Robotics and Automation (ICRA)*, 2011.

[58] Davide Scaramuzza and Roland Siegwart. Appearance-guided monocular omnidirectional visual odometry for outdoor ground vehicles. *IEEE transactions on robotics*, 2008. `https://www.ifi.uzh.ch/dam/jcr:8633224b-0bf4-4f5b-b721-7101e4ef8810/IEEE_TransRobotics_scaramuzza.pdf`.

[59] Jianbo Shi and Carlo Tomasi. Good features to track. *IEEE Conference on Computer Vision and Patter Recognition*, 1994.

[60] Chi Xu Shiqi Li and Ming Xie. A robust o (n) solution to the perspective-n-point problem. *IEEE transactions on pattern analysis and machine intelligence*, 2012. `https://pdfs.semanticscholar.org/e44b/46fd91fa1688eafe5419f36e76d9b9c016b7.pdf`.

[61] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. *Proc. 9th Int. Conf. on Computer Vision. Nice, France*, 2003.

[62] Andrew J. Davison Steven Lovegrove and Javier Ibanez-Guzmán. Accurate visual odometry from a rear parking camera. *Intelligent Vehicles Symposium (IV)*, 2011. https://www.doc.ic.ac.uk/~ajd/Publications/lovegrove_etal_iv2011.pdf.

[63] Sebastian Thrun and Michael Montemerlo. The graphslam algorithm with applications to large-scale map of urban structures. *International Journal on Robotics Research*, 2005.

[64] Microsoft Kinect v1. Especificaciones kinect v1. *web*, 2010. https://msdn.microsoft.com/en-us/library/jj131033.aspx.

[65] Francesc Moreno-Noguer Vincent Lepetit and Pascal Fua. Epnp: Anaccurate o (n) solution to the pnp problem. *International journal of computer vision*, 2009. http://cvlabwww.epfl.ch/~lepetit/papers/lepetit_ijcv08.pdf.

[66] Austin Weber. Nikola Tesla: Father of Unmanned Vehicle Technology. http://www.assemblymag.com/articles/87689, 2010.

[67] Alberto López-Cerón y José María Cañas. Accuracy analysis of marker-based 3d visual localization. *XXXVII Jornadas de Automática*, 2016. https://gsyc.urjc.es/jmplaza/papers/jornadasautomatica2016-pnp.pdf.

[68] Li Zhang, Brian Curless, and Steven. M. Seitz. Rapid shape acquisition using color structured light and multi-pass dynamic programming. *3DPVT*, 2002. http://grail.cs.washington.edu/projects/moscan/paper.pdf.

[69] N.L. Zhang and Poole D. A simple approach to bayesian network computations. *7th Canadian Conference on Artificial Intelligence*, 1994.