

Índice general

1. Introducción	1
1.1. Visión artificial	1
1.2. Autolocalización visual	4
1.2.1. Realidad aumentada	4
1.3. Técnicas de autolocalización visual	5
1.3.1. Structure from Motion (SfM)	5
1.3.2. Visual SLAM	6
Odometría visual	6
2. Objetivos	9
2.1. Descripción del problema	9
2.2. Requisitos	10
2.3. Metodología	10
2.4. Planificación	12
3. Infraestructura	14
3.1. Sensores RGBD	14
3.2. JDeRobot	15
3.2.1. Biblioteca Progeo	16
3.2.2. Biblioteca parallelIce	17
3.2.3. Servidor OpenniServer	17
3.2.4. Herramienta RGBDViewer	17
3.2.5. Pose3D	18
3.3. Biblioteca ICE de comunicaciones	18
3.4. Biblioteca Point Cloud Library (PCL)	18
3.5. Biblioteca OpenCV	18
3.6. Biblioteca Eigen	19
3.7. Biblioteca de interfaz gráfica GTK+	20
3.7.1. Glade	20
3.8. OpenGL	21
4. Desarrollo	23
4.1. Diseño	23
4.2. Análisis 2D	25
4.2.1. Detección de puntos de interés	25
4.2.2. Cálculo de descriptores	26
SIFT	27
SURF	29
4.3. Emparejamiento (<i>matching</i>)	31
4.3.1. Fuerza Bruta	32
4.3.2. FLANN	32
4.3.3. Resolución de errores	33

4.4. Obtención de puntos 3D	35
4.5. Cálculo de movimiento	38
4.5.1. Matriz RT	38
4.5.2. Cálculo RT (SVD)	39
4.5.3. Optimización	41
4.6. Interfaz gráfica	41
5. Experimentos	45
6. Conclusiones	46
6.1. Conclusiones	46
6.2. Trabajos futuros	47

Índice de figuras

1.1.	Embellcimiento de fotos (Meitu T8)	2
1.2.	Aplicaciones de radiografía en medicina	2
1.3.	Visión en la industria	3
1.4.	Ojo de halcón	3
1.5.	Visión para la autoconducción	4
1.6.	Ikea con realidad aumentada	5
1.7.	PhotoTourism	6
1.8.	MonoSLAM	7
1.9.	PTAM	8
2.1.	Ciclo de vida en espiral	11
2.2.	Componenete rectificador	12
2.3.	Esquema de cálculo de matriz RT	13
3.1.	Sensor Kinect	14
3.2.	Sensor Xtion	15
3.3.	Modelo <i>Pinhole</i>	16
3.4.	Captura de RGBDViewer	17
3.5.	Detección y emparejamiento con OpenCV	19
3.6.	Ejemplo con Eigen	20
3.7.	Captura visualizador 3D	22
4.1.	Diagram1	24
4.2.	Diagram2	24
4.3.	FeatureSimple	25
4.4.	ShiDetector	27
4.5.	siftScaleInvariant	27
4.6.	shifDog	28
4.7.	siftLocalExtrema	29
4.8.	Box filter	30
4.9.	Cálculo de orientación con SURF	31
4.10.	sift-detector	33
4.11.	bestPointsSift	34
4.12.	similar-correlation	34
4.13.	diagram-points-3d	36
4.14.	cam-line	36
4.15.	dist-point	37
4.16.	calculate-3d	38
4.17.	matrix-rt	39
4.18.	movement-rt	40
4.19.	supergui	42
4.20.	gui1	43
4.21.	gui2	43

4.22. <i>gui3</i>	44
4.23. <i>gui4</i>	44

Índice de cuadros

3.1. Especificaciones técnicas del Asus Xtion PRO LIVE	15
--	----

Capítulo 1

Introducción

El ser humano no es consciente del proceso neuronal que tiene lugar en nuestro cerebro con el simple hecho de andar o coger un objeto. Se podría decir que tenemos un super ordenador conectado a los órganos sensoriales capaces de recoger muchísima información y procesarla en un tiempo récord.

Desde la antiguedad ya se estuvo pensando en reproducir las habilidades humanas en algún tipo de máquina, la noción de concebir la mente humana como algún tipo de mecanismo no es reciente es referida en célebres filósofos, sin embargo, no es hasta 1950 y con la noción de la computación cuando se introduce la IA (Inteligencia Artificial) por el científico Alan Turing en su artículo *Maquinaria Computacional e Inteligencia* y donde se empieza a coger interés por este campo que será el precursor de una gran cantidad de desarrollos e innovaciones.

1.1. Visión artificial

Dentro del campo de la inteligencia artificial se puede definir visión artificial como la disciplina científica que incluye métodos para adquirir, procesar y analizar imágenes con el fin de producir información que pueda ser tratada por una máquina ofreciendo soluciones a problemas del mundo real.

Una manera simple de comprender este sistema es basarnos en nuestra propia experiencia. Los humanos usamos nuestros sentidos, en este contexto el ojo, para comprender el mundo que nos rodea, y la visión artificial busca producir ese mismo efecto en máquinas.

Cada vez son más los dispositivos electrónicos que llevan incorporada al menos una cámara; *smartphones*, ordenadores portátiles, *tablets*, consolas de videojuegos... Debido a la gran cantidad de información que se puede extraer de las imágenes, el bajo coste, el reducido tamaño de las cámaras y el aumento de capacidad de cómputo de los dispositivos, es un área que ha suscitado el interés por los investigadores, ha crecido enormemente en los últimos años y está cogiendo cada vez más fuerza.

Podemos ver cada vez más como los dispositivos electrónicos disponen de alguna nueva funcionalidad relacionada con el procesamiento de imágenes (Figura 1.1), como puede ser el reconocimiento facial que incorporan algunos smartphones o tablets para desbloquear el dispositivo o procesado automático de fotos realizadas por la cámara como la que incluye el terminal chino **Meitu T8** que incorpora un software llamado *AI Beautification* para embellecer las imágenes.¹

¹<https://www.cnet.com/products/meitu-t8/preview/>



FIGURA 1.1: Embellecimiento de fotos (Meitu T8) (a). Reconocimiento facial (b).

Sin ir más lejos, la reciente aplicación que ha desatado el revuelo en las diferentes redes sociales; FaceApp². La aplicación disponible tanto para Android e iOS es capaz de añadir sonrisas a las fotos, cambiar de edad o transformar el género de la persona que ha sido fotografiada.

El procesado de imágenes puede llegar a resultar muy útil en otros ámbitos como el de la medicina. Un ejemplo es la radiografía de la Figura 1.2 que partiendo de una imagen de muy baja calidad se pretende extraer información sobre las manchas blancas que aparecen en la misma.

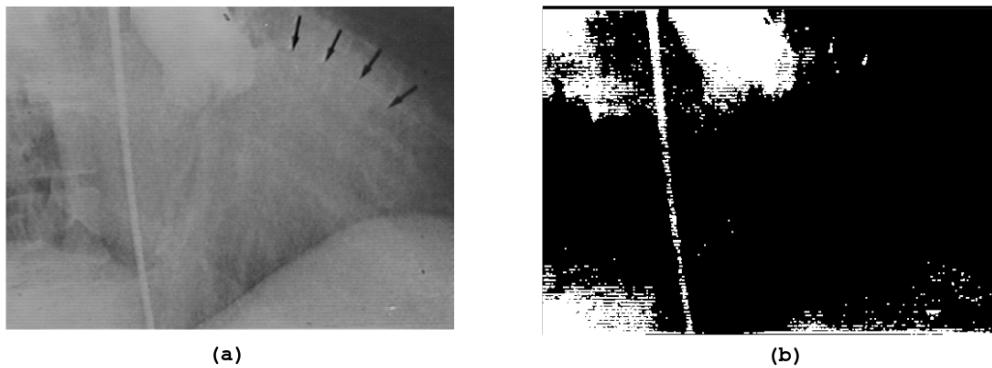


FIGURA 1.2: Radiografía inicial, con los puntos a analizar (a), imagen final procesada (b).

Son numerosas las aplicaciones de visión industrial relacionadas con el entorno de la alimentación. Permiten automatizar el control de calidad para tomar la decisión si un determinado producto cumple el estándar de calidad o no. Un ejemplo es el sistema EggInspector³ por la empresa Moba que se utiliza para clasificar huevos de gallina de forma automática. El sistema está compuesto por 6 cámaras suspendidas por encima de la cinta transportadora que con unos complejos algoritmos no solo pueden comprobar si los huevos están rotos o sucios, sino que son capaces de determinar el tipo de rotura y suciedad, una vez determinada la calidad, los que no corresponden a los estándares mínimos, son separados de la línea por un robot.

²<https://www.faceapp.com/>

³<http://www.moba.net/page/es/Grading/Moba-Grader-Options/Detection-Systems/Egg-inspector>

Siguiendo en la línea de la industria la inspección de embalajes se ha incrementado enormemente con la automatización del proceso y la visión artificial facilitando tareas como la detección del correcto nivel de llenado, verificación de tapones, control de calidad de sellado, lectura de óptica de caracteres (OCR), códigos de barras, verificación de posición, calidad de impresión de las etiquetas, conteo de productos en cajas o *palets*. Algunas de las aplicaciones típicas de la industria del *packaging* están representadas en la Figura 1.3.



FIGURA 1.3: Presencia, aplicación e integridad de etiquetas (a), códigos 2D (b), códigos de barras (c), validación de lote, fecha y código (d), orientación de piezas montadas (e), correcto sellado (f), calidad de impresión (g), presencia y cierre de tapones (h).

En los deportes quizás la aplicación más conocida sea el Ojo de Halcón (Figura 1.4), que se utiliza en los torneos de tenis de alto nivel para determinar la trayectoria de la pelota y saber si entró o no en el campo contrario, pero la visión artificial se usa en numerosos deportes sobretodo en estudios estadísticos post-partido para averiguar el tiempo de posesión del balón en los partidos de fútbol o los kilómetros hechos por cada jugador en el terreno de juego, entre muchos otros.

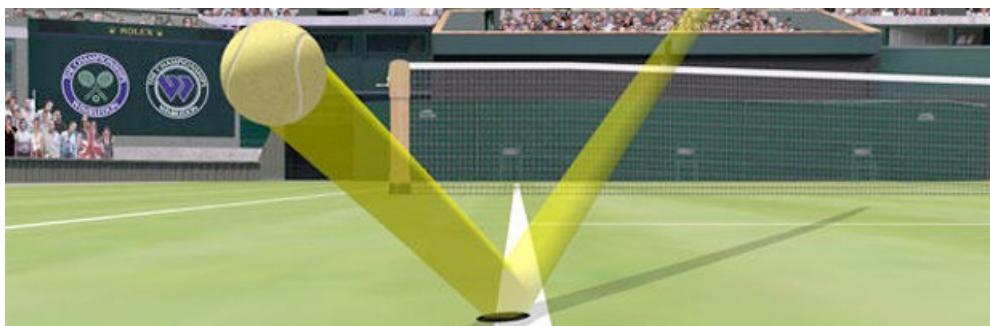


FIGURA 1.4: Ojo de Halcón en tenis

1.2. Autolocalización visual

La autolocalización visual consiste en conocer la localización de la cámara en todo momento simplemente con las imágenes capturadas sin disponer de ninguna información extra. Debido al gran abanico de posibilidades que propone este sistema, es uno de los retos más importantes dentro del campo de la robótica.

La autolocalización visual se plantea en los sistemas de navegación automáticos náuticos, terrestres y aéreos. Actualmente numerosas empresas están invirtiendo en este tipo de sistemas en el que apuestan por una navegación total o parcial.

Actualmente se pueden encontrar sistemas de asistencia y seguridad en los vehículos más modernos como; sistemas de frenado automático de emergencia, asistente de mantenimiento de carril o aparcado automático. Aunque estos sistemas se entienden como asistentes o ayudas a la conducción, el conductor sigue tomando la gran responsabilidad de la navegación.

La empresa israelí **Mobileye**⁴ presentará su primer modelo de vehículo completamente autónomo, junto a Intel y BMW, en 2021. El cerebro de la máquina se basa en un sensor, que identifica lo que ocurre a su alrededor al instante: los carriles, las señales de tráfico, otros automóviles, motos, bicicletas e incluso a los peatones. En la Figura 1.5 se puede ver una captura de la vista del coche antes de parar en un semáforo.



FIGURA 1.5: Vista del coche antes de parar en un semáforo.

1.2.1. Realidad aumentada

La realidad aumentada es el término que se usa para definir una visión directa o indirecta de un entorno físico del mundo real, cuyos elementos se combinan con elementos virtuales generados por ordenador para la creación de una realidad mixta en tiempo real.

⁴<https://www.mobileye.com/>

Aunque se ha popularizado con el juego de **Pokémon Go**⁵, cada vez son más los gigantes tecnológicos que se interesan por ella. La empresa sueca Ikea ya cuenta con una aplicación móvil que permite ver su catálogo en realidad aumentada (Figura 1.6).



FIGURA 1.6: Catálogo Ikea con realidad aumentada.

Puesto que la realidad virtual es una experiencia ficticia, tiene gran potencial en el mundo de los videojuegos. Pero no es el único. También puede tener aplicaciones en medicina, la industria del cine, la moda, los deportes o la publicidad.

1.3. Técnicas de autolocalización visual

Las técnicas de autolocalización ha suscitado gran interés por los investigadores en los últimos años. El problema ha sido abordado por dos comunidades distintas, por un lado la de visión artificial que denominó al problema como **structure from motion (SfM)**, donde la información es procesada por lotes, capaz de representar un objeto 2D a 3D con solo unas cuantas imágenes desde diferentes puntos de vista. Y por otro lado la comunidad robótica denominó al problema **SLAM (Simultaneous Localization and Mapping)** que trata de resolver el problema de una manera más compleja adaptando el funcionamiento de los sistemas en tiempo real.

1.3.1. Structure from Motion (SfM)

Las técnicas SfM se analizan generalmente de forma offline, las escenas se graban a través de un conjunto de imágenes y luego se procesa, lo que permite realizar optimizaciones para el cálculo de la trayectoria, como por ejemplo el llamado ajuste de haces.

Existen aplicaciones comerciales que utilizan estas técnicas como es el caso de la aplicación PhotoTourism (Noah Snavely y Szeliski, 2006) desarrollada por Microsoft. Que consiste en el cálculo de la posición 3D en la que fueron captadas las imágenes,

⁵<http://www.pokemongo.com/es-es/>

por ejemplo de un monumento, para después extraer el modelo 3D con el que el usuario puede interactuar libremente (Figura 1.7).

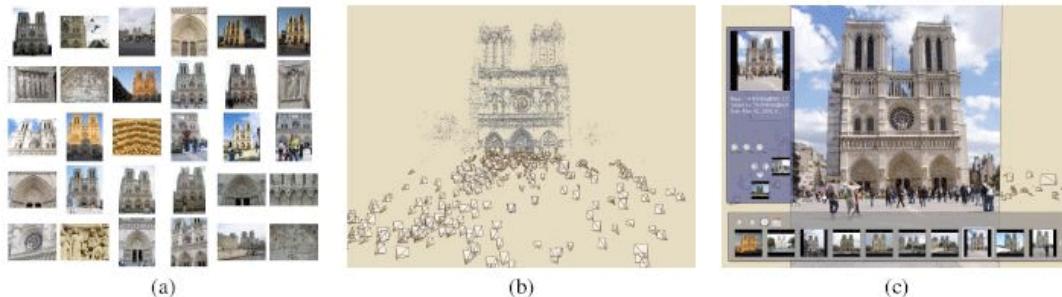


FIGURA 1.7: PhotoTourism: Se recogen una gran colección de imágenes (a), se reconstruyen los puntos 3D y los puntos de vista (b), por último la interfaz permite al usuario interactuar moviéndose a través del espacio 3D mediante la transición entre fotografías.

1.3.2. Visual SLAM

En el problema conocido como *Simultaneous Localization and Mapping* (SLAM) busca resolver los problemas que plantea colocar un robot móvil en un entorno y una posición desconocidas, y que él mismo se encuentre capaz de construir incrementalmente un mapa de su entorno consistente y a la vez utilizar dicho mapa para determinar su propia localización.

La solución a este problema conseguiría hacer sistemas de robots completamente autónomos que junto con un mecanismo de navegación el sistema se encontrará con la capacidad para saber a dónde desplazarse, ser capaz de encontrar obstáculos y reaccionar ante ellos de manera inteligente.

La resolución al problema SLAM ha suscitado un gran interés en el campo de la robótica y ha sido resuelto teóricamente de diversas formas como es el caso del artículo (Durrant-Whyte y Bailey, 2006). Y aunque algunas de ellas han obtenido buenos resultados en la práctica siguen surgiendo problemas a la hora de buscar el método más rápido o el que genere un mejor resultado con menos índice de fallo. La búsqueda de algoritmos y métodos que resuelvan estos problemas sigue siendo una tarea pendiente.

Odometría visual

Dentro de las familias de técnicas pertenecientes a las de Visual SLAM se encuentra la de odometría visual, que es la que abordaremos en este trabajo. Consiste en la estimación del movimiento de la cámara en tiempo real. Es decir, el cálculo de la rotación y traslación de la cámara a partir de dos imágenes simultáneas. Se trata de una técnica incremental ya que se basa en la posición anterior para calcular la nueva.

Este tipo de algoritmos se suelen utilizar técnicas de extracción de puntos de interés, cálculos de descriptores y algoritmos para el emparejamiento. Normalmente el proceso es el mismo, una vez calculados los puntos emparejados se calcula la matriz fundamental o esencial y descomponerlas mediante SVD para obtener la matriz

de rotación y translación (RT) (Scaramuzza y Fraundorfer, 2011. Fraundorfer y Scaramuzza, 2012).

Uno de los trabajos más importantes en el ámbito es el de monoSLAM de Davison⁶ (Andrew J. Davison y Stasse, 2007) que propone resolver este problema con una única cámara RGB como sensor y realizar el mapeado y la localización simultáneamente. El algoritmo propuesto por Davison utiliza un filtro extendido de Kalman para estimar la posición y la orientación de la cámara, así como la posición de una serie de puntos en el espacio 3D. Para determinar la posición inicial de la cámara es necesario a priori dotar de información con la posición 3D de por lo menos 3 puntos. Después el algoritmo es capaz de situar la cámara en el espacio tridimensional y de generar nuevos puntos para crear el mapa y servir como apoyo a la propia localización de la cámara. En la Figura 1.8 se pueden ver unas capturas de pantalla sobre uno de los experimentos realizados.

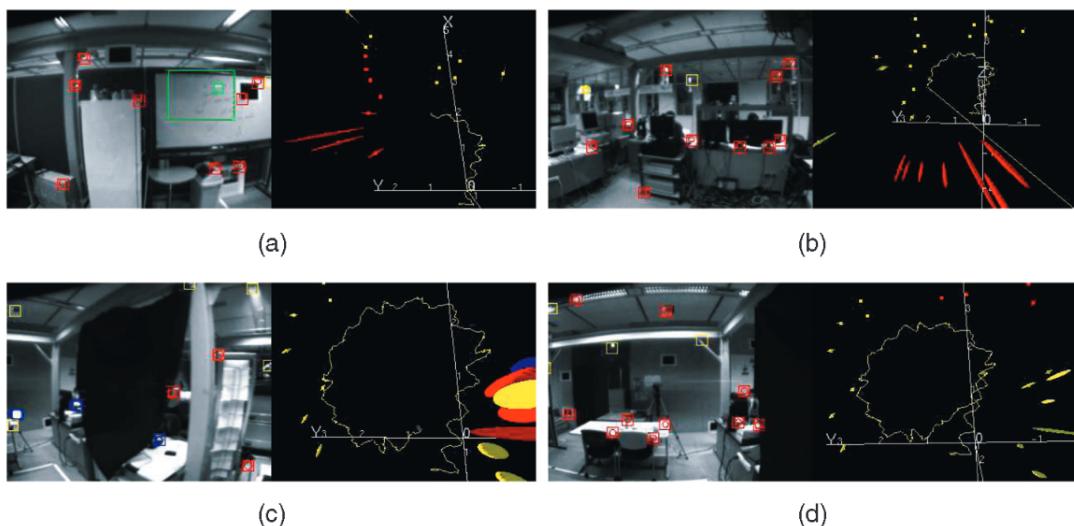


FIGURA 1.8: MonoSLAM: Un robot humanoide camina en una trayectoria circular de radio 0.75m. La estela amarilla muestra la trayectoria estimada del robot, y las elipses muestran los errores de localización.

Es importante destacar también la trascendencia que ha tenido el trabajo PTAM (Klein y Murray, 2007) que viene a solucionar uno de los principales problemas que tienen los algoritmos monoSLAM; el tiempo de cómputo, ya que aumenta exponencialmente con el número de puntos (Figura 1.9). Para ello se aborda el problema separando el mapeado de la localización, de tal modo que solo la localización deba funcionar en tiempo real, dejando así que el mapeado trabaje de una manera asíncrona. Este algoritmo parte de la idea de que solo la localización es necesaria que funcione en tiempo real. PTAM hace uso de *keyframes*, es decir, fotogramas clave que se utilizan tanto para la localización como para el mapeado y también de una técnica de optimización mediante ajuste de haces, como en SfM.

⁶<http://www.doc.ic.ac.uk/~ajd/>

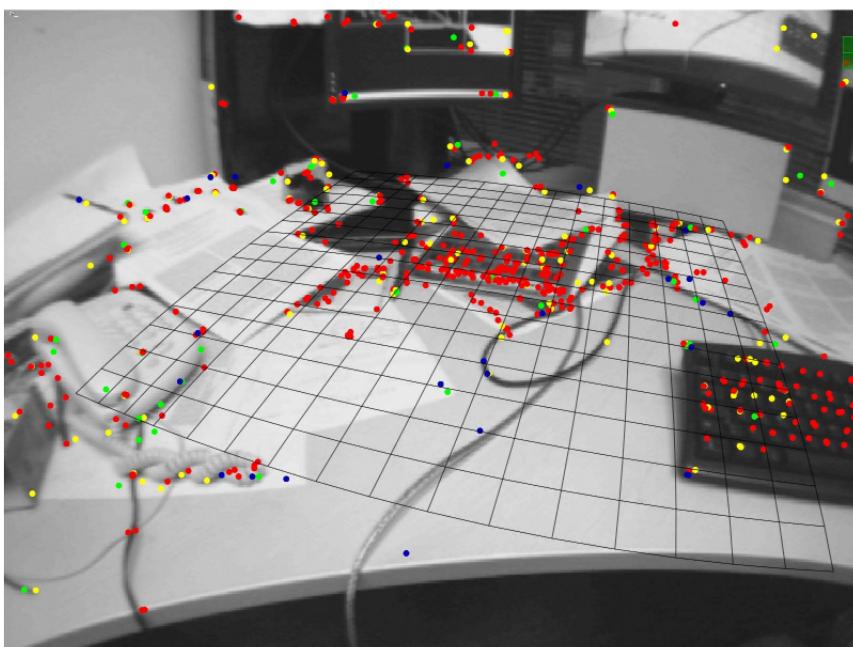


FIGURA 1.9: PTAM: Funcionamiento típico del sistema sobre un escritorio.

Capítulo 2

Objetivos

Una vez presentado el contexto, en este capítulo se abordarán los objetivos que se pretenden alcanzar. Tras una descripción del problema y unos requisitos obligatorios, se detallará la metodología y la planificación que se ha llevado a cabo en la elaboración de este trabajo.

2.1. Descripción del problema

El objetivo principal de este trabajo es desarrollar una solución al problema SLAM.

A través de técnicas de odometría visual incrementales, se ha construido un sistema capaz de averiguar la posición y orientación de un sensor RGBD que se mueve libremente por el espacio y que va alimentando al sistema con las imágenes obtenidas en tiempo real.

A fin de abordar el objetivo principal de la manera más simple se han propuesto los siguientes subobjetivos:

1. Actualización de los datos del sensor. Creación de un módulo encargado de recoger de manera dinámica las imágenes RGB y DEPTH del sensor, y actualizarlas en una memoria compartida.
2. Detección de puntos de interés. Creación de un componente capaz de recoger a través de diferentes técnicas puntos de interés de una imagen con la opción de desarrollar algún filtro para añadir robustez a la detección.
3. Emparejamiento de puntos. Desarrollo de una funcionalidad encargada de relacionar los puntos de interés previamente obtenidos, a través de técnicas como Fuerza bruta o FLANN.
4. Estimación del movimiento. Implementación de la matemática necesaria para una vez tenidos los puntos emparejados calcular la matriz de rotación y traslación (RT) que se necesitará para calcular en ese instante el movimiento de la cámara.
5. Pruebas y experimentos. Ejecución de la aplicación con objetivo de ajustar, pulir y validar las funciones y algoritmos realizados.

2.2. Requisitos

A parte de los requisitos mencionados en el apartado anterior. La implementación y solución final del proyecto debe satisfacer además los siguientes puntos:

- Desarrollo del proyecto haciendo uso de la plataforma **JDeRobot 5.4.0**, que ha resultado de mucha utilidad y ha permitido un ahorro significante de tiempo, ya que permite abstraerse de algunas de las funcionalidades de más bajo nivel como puede ser la captura de información del sensor o el protocolo de comunicaciones. Permite llevar un desarrollo modular y el aprovechamiento de componentes ya implementados en la plataforma. Tanto las ventajas como el uso de JDeRobot se tratarán en detalle en el siguiente capítulo (3).
- Como la mayoría de componentes están en C++, el trabajo también se ha desarrollado utilizando el mismo lenguaje de programación.
- Funcional bajo sistema operativo linux, en este caso Ubuntu 14.04 LTS.
- Uso exclusivo de un único sensor RGBD.
- Implementación de una interfaz de usuario clara donde se pueda apreciar el proceso de estimación de posición. Incluyendo un entorno visual 3D donde se refleje la posición y movimiento de la cámara en tiempo real.

2.3. Metodología

Para abordar un proyecto de tal envergadura es necesaria una metodología de desarrollo para ir progresando de una manera ordenada y efectiva. En este caso se ha optado por el modelo en espiral basado en prototipos propuesto por B. Boehm en 1986 (Boehm, 1986), ya que permite el desarrollo de una manera progresiva e incremental.

Esta metodología permite el desarrollo de implementaciones parciales que van siendo probadas a medida que después de cada ciclo se va generando un prototipo más completo de lo que incorpora el ciclo anterior. Por lo tanto, en cada ciclo o iteración se va añadiendo complejidad a la vez que se van generando funcionalidades nuevas.

Este modelo, utilizado en el trabajo, ha servido de gran ayuda, ya que permite ir avanzando de menos a más, con unos requisitos dependientes de los anteriores y a la vez diferentes para cada iteración. Además permite ir evaluando y adaptando la evolución del desarrollo a nuestros intereses, algo que suele ocurrir normalmente en los proyectos de investigación.

En la Figura 2.1 se puede observar el ciclo completo de desarrollo de software en el modelo en espiral. Cada etapa o ciclo completo está compuesto por cuatro fases:

- Identificación de objetivos. En esta primera fase se deciden y se planifican los objetivos a alcanzar en la siguiente iteración partiendo de lo realizado en el ciclo anterior. En caso de la primera iteración se definen los objetivos iniciales.
- Evaluación alternativa. Aquí se definen requisitos y se estudian las distintas maneras de abordar los objetivos marcados de la etapa anterior. Se estudian los riesgos y se evalúan de manera que se puedan reducir lo máximo posible. Se debe tener un prototipo antes de la siguiente etapa.

- Desarrollo del producto. En esta fase se diseña y se implementa el producto en base a lo planteado en las anteriores fases. Por último, se verifica y se prueba.
- Planificación de la siguiente fase. Considerando el resultado de la fase anterior, se planifica la siguiente considerando los errores cometidos y los resultados esperados, comenzando así una nueva iteración.

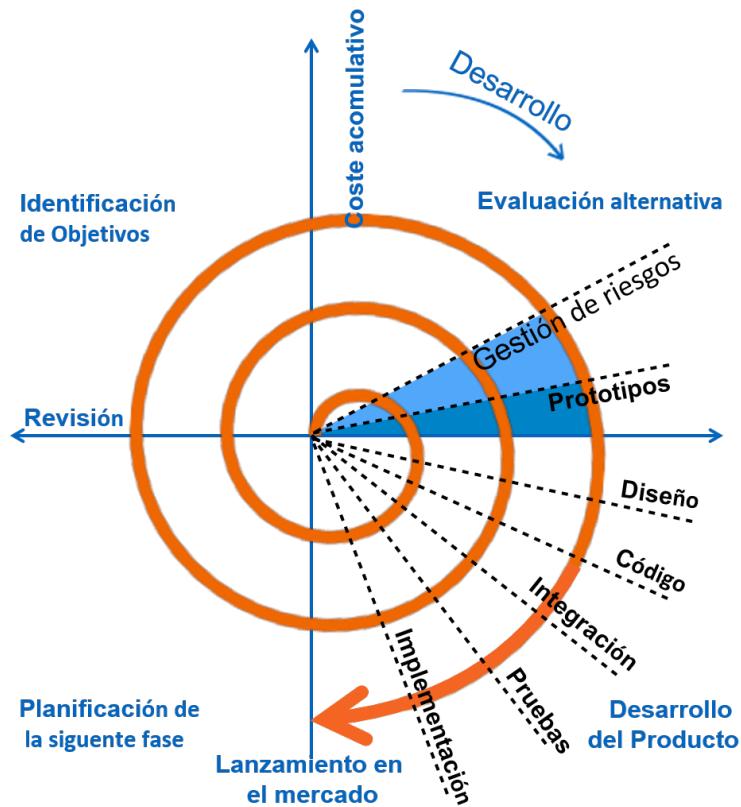


FIGURA 2.1: Ciclo de vida del desarrollo del *software* en el modelo espiral.

En la siguiente sección se detallarán los ciclos que han seguido a lo largo de este proyecto. Para las fases de planificación y análisis se han mantenido reuniones semanales con el tutor, con intención de revisar, resolver problemas y encarar los nuevos objetivos establecidos.

A fin de documentar y guardar los hitos realizados en el desarrollo del proyecto, así como los errores cometidos y su posible solución, se ha llevado un seguimiento en mediawiki¹ con los detalles de las diferentes iteraciones, ayudadas a veces por imágenes y/o videos.

Para la gestión de código se han usado herramientas *software* de control de versiones, primeramente con Subversion (SVN)² y finalmente con GIT en un repositorio de GitHub³.

¹<http://jderobot.org/J.benitod.tfg>

²<http://svn.jderobot.org/users/j.benitod/pfc>

³<https://github.com/RoboticsURJC-students/2014-pfc-Javier-Benito>

2.4. Planificación

A lo largo del trabajo se han ido proponiendo etapas asesoradas y con supervisión del tutor. Las más importantes caben destacar:

1. Familiarización de la herramienta JDeRobot.

Esta etapa consistió en la instalación y el estudio de la plataforma, profundizando en el uso de algún componente con un objetivo muy concreto y sencillo.

Después, y para entender el funcionamiento de algunos de los componentes a bajo nivel más importantes para el trabajo, se propuso el desarrollo de algunos de ellos en otros lenguajes de programación tales como Java o Python.

2. Aprendizaje de las herramientas específicas.

Aquí, a través de prácticas muy concretas se entendió el funcionamiento de algunas de las librerías esenciales para la práctica final.

- Se realizó un componente utilizando **PCL** para el cálculo de planos desde nubes de puntos.
- Se utilizó **Eigen** para otra práctica en el cálculo de sistemas sobredimensionados de ecuaciones, con descomposición QR y en valores singulares (SVD) resolviendo en diferentes casos una recta de regresión para distintos escenarios.
- **GSL**, para el cálculo de un componente rectificador de imágenes (Figura 2.1).

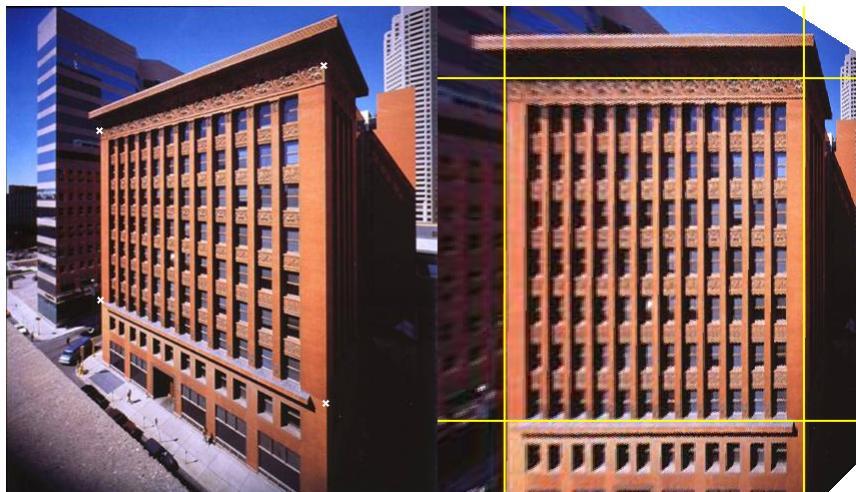


FIGURA 2.2: Componente rectificador de imágenes.

3. Implementación de un algoritmo para el cálculo de la matriz RT.

En esta parte se propuso una práctica que sirvió de base para la parte final en la que desde una nube de puntos y otra multiplicada por una matriz RT inventada, se sacaría a través de SVD dicha matriz a partir de las nubes de puntos iniciales y con la opción también de añadir ruido gaussiano a una de ellas. En la Figura 2.3 se encuentra el esquema realizado.

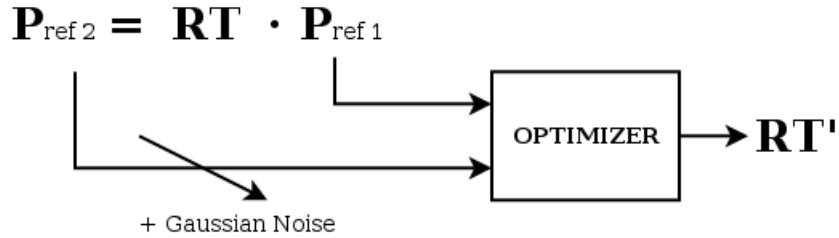


FIGURA 2.3: Esquema de la práctica de cálculo de matriz RT.

4. Creación de un componente para la extracción de puntos de interés y emparejamiento.

Aquí se empezó a desarrollar la práctica final, en donde se comenzó por la extracción de puntos de interés con SIFT de las imágenes y los diferentes algoritmos para los emparejamientos. Como esta parte corresponde al desarrollo de la práctica final será detallada en el capítulo 4.

5. Integración y pruebas experimentales.

Esta fue la etapa más larga y costosa, ya que el objetivo era dar una solución funcional al problema propuesto. Surgieron multitud de errores que se tuvieron que ir depurando con cada iteración, proponiendo soluciones y alternativas. Fueron necesarias numerosas pruebas para conseguir un desarrollo capaz de aportar una solución estable. Esta etapa también será detallada en los siguientes capítulos, en concreto en los capítulos 4 y 5.

Capítulo 3

Infraestructura

En este capítulo se detallarán las herramientas base empleadas en la realización de este trabajo.

3.1. Sensores RGBD

Los sensores RGBD son capaces de captar a parte de las componentes roja, verde y azul de la luz, información de profundidad (o "D" depth en inglés). Es decir, por cada píxel asocia la información de color con su correspondiente componente de profundidad. Esta tecnología fue desarrollada por la empresa israelí **PrimeSense**. El sensor Kinect dispone también de un micrófono multiarray con el cual puede predecir de dónde proviene el sonido.

En el 2010 Microsoft sacó al mercado el sensor Kinect (Figura 3.2) para la consola de juegos Xbox 360 y Xbox One. Pronto se convirtió en uno de los dispositivos electrónicos más vendidos en todo el mundo después de su lanzamiento.

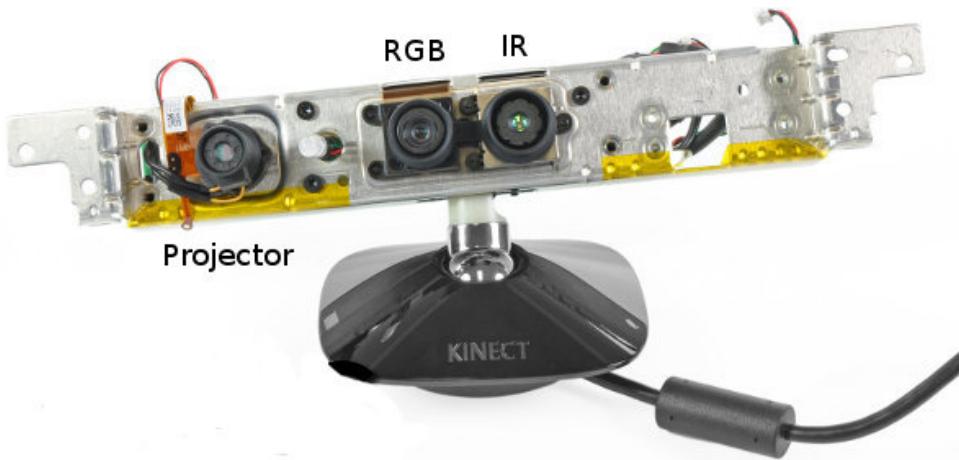


FIGURA 3.1: Sensor Microsoft Kinect

Este sensor salió al mercado a un precio mucho más reducido que algunos que existían antes que él por lo que el interés por este tipo de sensores se disparó y comenzaron a aparecer en diferentes áreas de la tecnología, como interfaces naturales de usuario (en inglés natural user interface, NUI), reconstrucción y realidad virtual o cartografía 3D.

CUADRO 3.1: Especificaciones técnicas del Asus Xtion PRO LIVE

Campo de visión:	58° H, 45° V, 70° D
Distancia de uso:	Entre 0.8m y 3.5m
Tamaño de la imagen de profundidad:	VGA (640x480) : 30 fps QVGA (320x240): 60 fps
Resolución:	SXGA (1280*1024)

El sensor utilizado en este trabajo es el **Asus Xtion PRO LIVE** que dispone de la misma tecnología comercializado por Asus, que proporciona profundidad, color y audio (utilizando un micrófono multiarray como el sensor Kinect). ¹



FIGURA 3.2: Asus Xtion PRO LIVE

Las especificaciones técnicas de este sensor se encuentran recogidas en la tabla 3.1

3.2. JDeRobot

JDeRobot es un proyecto desarrollado por el grupo de robótica de la Universidad Rey Juan Carlos ². Consiste en una plataforma de desarrollo de aplicaciones robóticas y de visión artificial. Está en su mayoría escrito en C++, donde disponen de una colección de componentes capaces de comunicarse a través de ICE middleware ³, los componentes pueden ejecutarse en diferentes ordenadores y pueden ser programados en diferentes lenguajes.

JdeRobot incluye numerosas herramientas, drivers, interfaces, librerías y tipos. Es software libre con licencia GPL y LGPL. También utiliza software de terceros como Gazebo, ROS, OpenGL, GTK y Eigen entre otros.

La versión de JdeRobot empleada ha sido la versión 5.4.0. A continuación se detallarán los componentes de JdeRobot que han sido de utilidad para la realización de este proyecto.

¹https://www.asus.com/3D-Sensor/Xtion_PRO_LIVE/

²<http://jderobot.org>

³<https://zeroc.com/products/ice>

3.2.1. Biblioteca Progeo

Es una biblioteca de geometría proyectiva incluida en JdeRobot, que proporciona funciones muy útiles que relacionan puntos en dos y tres dimensiones.

Ha sido realmente útil en este trabajo para que a partir de puntos en dos dimensiones (pixeles) y su correspondiente información de profundidad (distancia), sacar los puntos relativos de la cámara en tres dimensiones.

Progeo usa el modelo de cámara **Pinhole**, en la Figura 3.3 se puede observar la representación geométrica de la retroproyección y la proyección. Este modelo es definido por unos parámetros intrínsecos y extrínsecos que definen la composición de todos los parámetros iniciales de configuración de la cámara. Los parámetros extrínsecos que establecen la posición 3D, foco de atención (foa) y roll, mientras que los parámetros intrínsecos determinan la distancia focal y el centro óptico o píxel central.

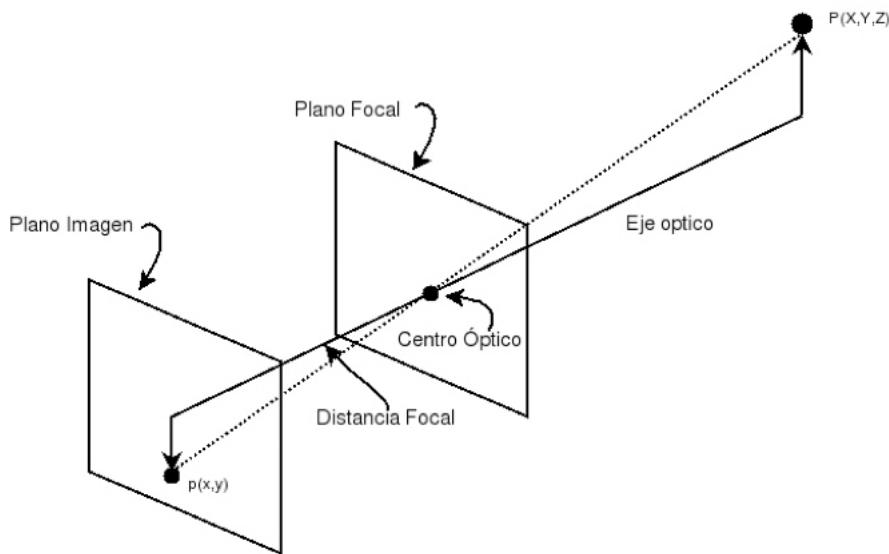


FIGURA 3.3: Modelo de cámara Pinhole

Las funciones que proporciona esta biblioteca son las siguientes:

- **Project:** Esta función permite proyectar un punto 3D del mundo al correspondiente pixel en 2D de la imagen de la cámara.
- **Backproject:** Esta función es capaz de a partir de las coordenadas de un píxel en 2D, obtener la línea de proyección que conecta la cámara y el foco con el rayo 3D que proyecta dicho píxel en el plano imagen. Con esto y conociendo la distancia real del punto 3D a calcular, se calcula las coordenadas reales del punto 3D.
- **DisplayLine:** Esta función permite conocer si una línea definida por dos puntos en 2D es visible dentro del plano imagen.
- **Display_info:** Esta función muestra toda la información sobre la cámara utilizada.

3.2.2. Biblioteca parallelIce

Es otra librería incluída en JdeRobot, que soluciona el problema de latencia de información proveniente de los diferentes drivers, evitando la espera y proviniendo de un acceso asíncrono a una copia en local de las interfaces con muy bajo tiempo de procesado.

3.2.3. Servidor OpenniServer

OpenniServer es un driver que se comporta como un servidor y es capaz de proporcionar con un sensor RGBD (Kinect o Xtion), imágenes de color, de profundidad o nubes de puntos que son enviados a través de la interfaz ICE a un puerto específico, donde se pueden escuchar los datos. Este driver es el que se necesita para el funcionamiento de este trabajo ya que es desde donde se recogen tanto las imágenes de color (RGB) como las de profundidad (Depth) para su posterior procesado.

3.2.4. Herramienta RGBDViewer

Es una herramienta que permite enseñar la información proveniente de los sensores RGBD con openniServer como forma de visualización de los datos; imágenes RGB, DEPTH o nubes de puntos.

El funcionamiento corresponde a un hilo de ejecución llamado Control que se encarga de recolectar las imágenes provenientes del driver, una clase Shared para guardar y recoger los datos, y por último una clase Gui que se encargará de coger los datos (imágenes y nubes de puntos) guardados en Shared y mostrarlas.

Esta herramienta a servido como referencia para la realización de este trabajo. En la Figura 3.4 podemos ver una captura de pantalla con las diferentes visualizaciones.

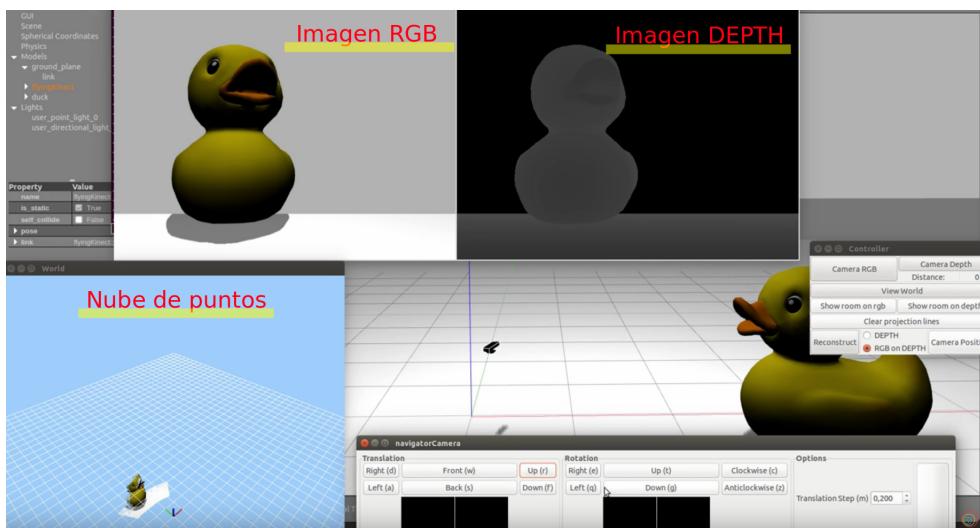


FIGURA 3.4: RGBDViewer: Captura de pantalla con las tres vistas de los diferentes datos; imagen de color, de profundidad y nube de puntos.

3.2.5. Pose3D

Es una interfaz que define una posición en tres dimensiones (x, y, z, h) y una orientación con un cuaternión (q_0, q_1, q_2, q_3).

3.3. Biblioteca ICE de comunicaciones

ICE (Internet Communications Engine) es un RPC framework desarrollado por Zeroc con soporte en C++, C#, Java, JavaScript y Python entre otros. Se encuentra bajo doble licencia GNU GPL y código cerrado. Actúa como plataforma de comunicaciones y funciona bajo TCP/IP.⁴

En JdeRobot la podemos encontrar como librería y es utilizada como protocolo de comunicaciones entre los diferentes componentes de JdeRobot. En nuestro trabajo se ha usado la versión 3.5.1 y nos ha servido para establecer la comunicación entre el componente y el driver del sensor, recogiendo las imágenes de éste.

3.4. Biblioteca Point Cloud Library (PCL)

PCL es una librería desarrollada en C++ para el procesamiento de imágenes 2D/3D y nubes de puntos. Está publicada con licencia BSD y libre bajo usos comerciales y de investigación. Está financialmente soportada por un consorcio de compañías comerciales y su propia organización sin ánimo de lucro, **Open Perception**. A parte de los donadores y contribuidores individuales que aportan al proyecto.⁵

Para simplificar el uso y el desarrollo, esta librería se encuentra dividida en módulos individuales de los que destacan el filtrado de puntos *outliers* o de ruido, estructuras de datos, estimación 3D, algoritmos para la detección de puntos de interés, combinación, segmentación, algoritmos para el reconocimiento de objetos.

En su página web disponen de mucha información y ejemplos prácticos que ayudan mucho a la compresión de todas las funcionalidades de esta librería. PCL también dispone de una librería i/o de entrada y salida para leer o crear nubes de puntos a partir de diferentes dispositivos, así como visualizadores 3D.

3.5. Biblioteca OpenCV

OpenCV (Open Source Computer Vision Library) es una librería de código abierto que fue desarrollada para proporcionar una infraestructura común en aplicaciones de visión artificial y facilitar la inteligencia máquina, con mecanismos de aprendizaje y de interpretación de datos. Con licencia BSD da facilidades para su uso y su modificación bajo fines comerciales.⁶

⁴<https://zeroc.com/products/ice>

⁵<http://pointclouds.org/>

⁶<http://opencv.org/>

La librería contiene más de 2500 algoritmos. Estos algoritmos pueden ser usados para detectar y reconocer rostros, identificar objetos, clasificar acciones humanas determinadas en videos, seguimiento del movimiento de cámaras, seguimiento de objetos, extraer modelos de objetos 3D, producir nubes de puntos a partir de cámaras, encontrar imágenes similares de un conjunto, juntar trozos de imágenes para producir una imagen final con más resolución, etc... OpenCV tiene más de 47 miles de usuarios en la comunidad, excediendo los 14 millones de descargas, la librería es usada ampliamente en empresas, grupos de investigación y organismos gubernamentales.

OpenCV a sido diseñada de forma eficiente y con un fuerte enfoque en aplicaciones de tiempo real. Escrita en C/C++, la librería obtiene las ventajas del procesamiento multi-núcleo. Dispone de interfaces en C++, C, Python, Java y MATLAB y es soportada por diferentes sistemas operativos como Windows, Linux, Android y Mac OS.

En este trabajo se ha utilizado la versión 2.4.8 y se ha usado a la hora de identificar puntos de interés y para los distintos métodos de emparejamiento. En la Figura 3.5 se puede apreciar un ejemplo de su uso en la detección de puntos de interés sobre un entorno real.

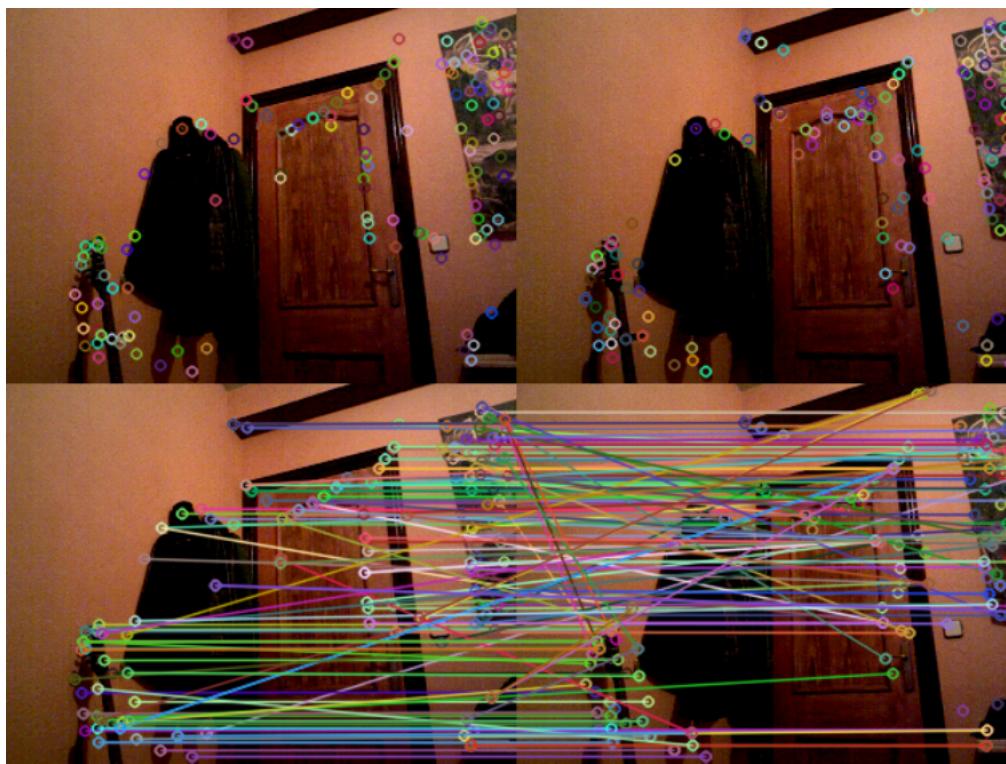


FIGURA 3.5: Detección y emparejamiento de puntos de interés con OpenCV.

3.6. Biblioteca Eigen

Eigen es una librería de álgebra lineal que permite hacer operaciones aritméticas con matrices y vectores, a través de los operadores comunes de C++, tales como +, -, *,

o a través de métodos especiales tales como `dot()`, `cross()`, etc... Para la clase `Matrix` (matrices y vectores) los operadores solo soportan operaciones de álgebra lineal. En Figura 3.6 se puede ver un ejemplo de como es simple hacer una multiplicación y una división por un escalar.⁷

Example:	Output:
<pre>#include <iostream> #include <Eigen/Dense> using namespace Eigen; int main() { Matrix2d a; a << 1, 2, 3, 4; Vector3d v(1,2,3); std::cout << "a * 2.5 =\n" << a * 2.5 << std::endl; std::cout << "0.1 * v =\n" << 0.1 * v << std::endl; std::cout << "Doing v *= 2;" << std::endl; v *= 2; std::cout << "Now v =\n" << v << std::endl; }</pre>	<pre>a * 2.5 = 2.5 5 7.5 10 0.1 * v = 0.1 0.2 0.3 Doing v *= 2; Now v = 2 4 6</pre>

FIGURA 3.6: Ejemplo de una multiplicación y división por un escalar con Eigen.

Eigen es *software* libre y desde la versión 3.1.1 tiene licencia MPL2 (GPL3+ para las anteriores versiones). Se ha usado la versión 3.2.0 y ha sido de utilidad en este proyecto para realizar los cálculos de la matriz RT a través de los vectores de puntos 3D ya emparejados.

3.7. Biblioteca de interfaz gráfica GTK+

GTK+, o the GIMP Toolkit es una herramienta multiplataforma de creación de interfaces gráficas. Es multiplataforma y está escrito en C, pero a sido diseñado para tener soporte para un gran rango de lenguajes, tales como Perl y Python. GTK++ tiene una gran colección de *widgets* y interfaces para usar en la aplicación, tales como ventanas, botones, selectores, cajas de texto, etc.

La versión utilizada ha sido la 3.10.8. Es *software* libre y parte del proyecto GNU. Con licencia LGPL, permite que sea utilizado por todos los desarrolladores, incluyendo aquellos que están desarrollando un *software* privativo. GTK+ ha sido utilizado en muchos proyectos y en grandes plataformas.⁸

3.7.1. Glade

Glade es una *RAD tool* (Rapid Application Development Tool) que permite desarrollar de manera fácil y rápida interfaces de usuario en GTK+ para el entorno de escritorio GNOME. La interfaz gráfica diseñada en Glade es guardada en un XML

⁷<http://eigen.tuxfamily.org/>

⁸<https://www.gtk.org/>

que usando los objetos GTK+ de **GtkBuilder** pueden ser cargados y utilizados por aplicaciones de forma dinámica como se ha hecho en este trabajo.⁹

3.8. OpenGL

OpenGL es el principal entorno para el desarrollo de aplicaciones gráficas 2D y 3D interactivas. Desde 1992, OpenGL se ha convertido en la interfaz de aplicaciones gráficas más utilizada y soportada en la industria 2D y 3D, con miles de aplicaciones disponibles en diferentes plataformas. OpenGL ayuda al desarrollo de aplicaciones al incorporar un amplio conjunto de renderizado, mapeo de texturas, efectos especiales y otras potentes funciones de visualización. Se puede usar OpenGL en la mayoría de entornos de escritorio y diferentes plataformas. Es muy utilizada y conocida en la industria de los videojuegos.

Algunas de las ventajas de las que presume OpenGL son; que es un estándar de la industria, con soporte, multiplataforma y el único libre. Es estable, dispone de compatibilidad hacia atrás, escalable, fácil de usar y bien documentado.¹⁰

Se ha usado la librería **Mesa 3D Graphics** en linux que es una implementación de la especificación de OpenGL con código abierto¹¹.

OpenGL en este trabajo se ha usado para visualizar la posición de la cámara, su estela y la colección de nubes de puntos obtenida y procesada de las imágenes RGB y de profundidad. En la Figura 3.7 se puede apreciar una captura de pantalla con la posición de la cámara dibujada en el espacio tridimensional con el visualizador utilizado con OpenGL.

⁹<https://glade.gnome.org/>

¹⁰<https://www.opengl.org/>

¹¹<https://www.mesa3d.org/>

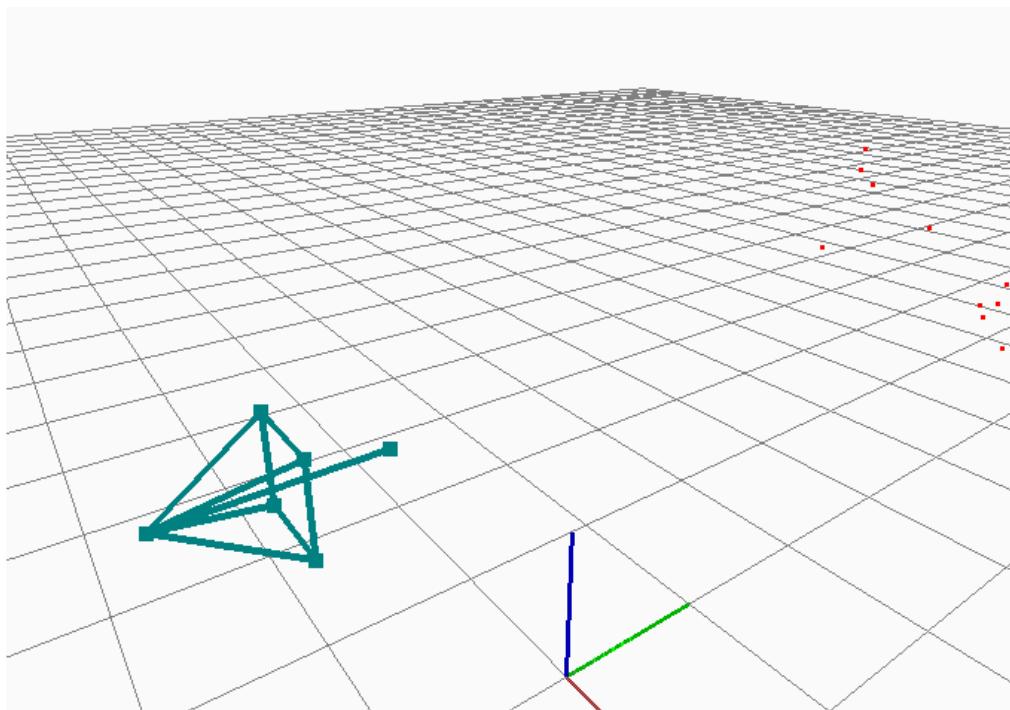


FIGURA 3.7: Captura de la posición de la cámara en el visualizador 3D con OpenGL.

Capítulo 4

Desarrollo

Una vez presentado el contexto, los objetivos, así como las herramientas empleadas y los fundamentos teóricos, en este capítulo se detallará la solución software final desarrollada. Primero se presenta el diseño global utilizado y después se analizará en detalle el componente en cuestión realizado con una visión profunda del desarrollo por bloques y su funcionamiento.

4.1. Diseño

El trabajo se basa principalmente en dos componentes; un componente de JDeRobot (**OpenniServer**) que funciona como driver del sensor y proporciona las imágenes obtenidas por éste y el componente realizado (**RealRTEstimator**) que se encargará, una vez recogidas las imágenes, de toda la lógica restante.

El objetivo del componente, como ya se ha comentado, consiste en analizar en tiempo real la posición y movimiento del sensor, por lo que el componente deberá dar una estimación en todo momento.

En la Figura 4.1 se puede apreciar el diagrama global de funcionamiento del componente desarrollado y su conexión con otros componentes para los diferentes datos de entrada.

OpenniServer se encarga de preparar y enviar las imágenes del sensor. El componente recoge las imágenes a través de ICE y éste es el encargado de procesarlas. También recibe los datos de los parámetros intrínsecos de la cámara así como algunos parámetros de configuración, como pueden ser la activación/desactivación de la interfaz de usuario o algunos parámetros configurables de algunos de los algoritmos internos. A su salida entrega una matriz RT que describe la posición y orientación absolutas en ese preciso instante de tiempo.

Respecto al funcionamiento interno del componente se puede ver a grandes rasgos el diagrama en la Figura 4.2. Se observa el diseño implementado así como sus bloques funcionales:

- Extracción de puntos de interés (análisis 2D) del fotograma actual.
- Emparejamientos de puntos de interés en t con respecto a los puntos extraídos en el instante anterior (t-1).
- Transformación de puntos (pixeles) en 2D más imagen de profundidad a nube de puntos en 3D.

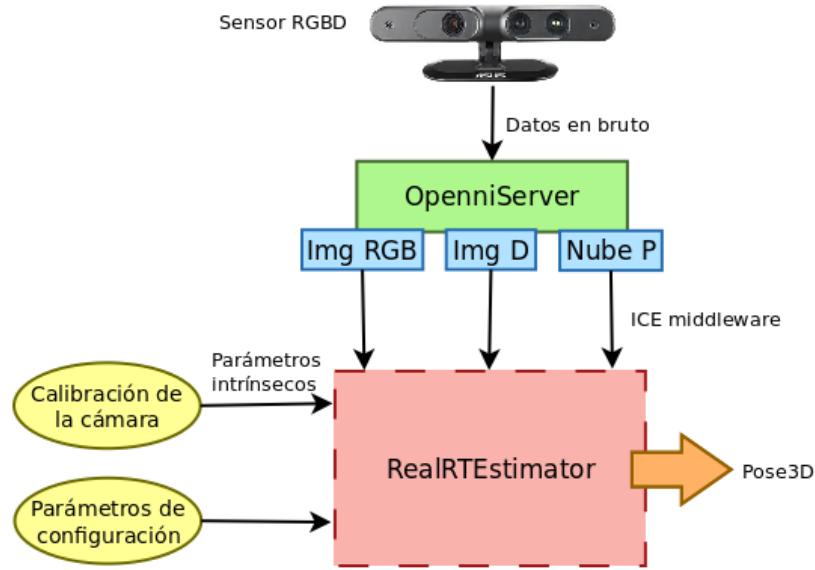


FIGURA 4.1: Esquema global de funcionamiento.

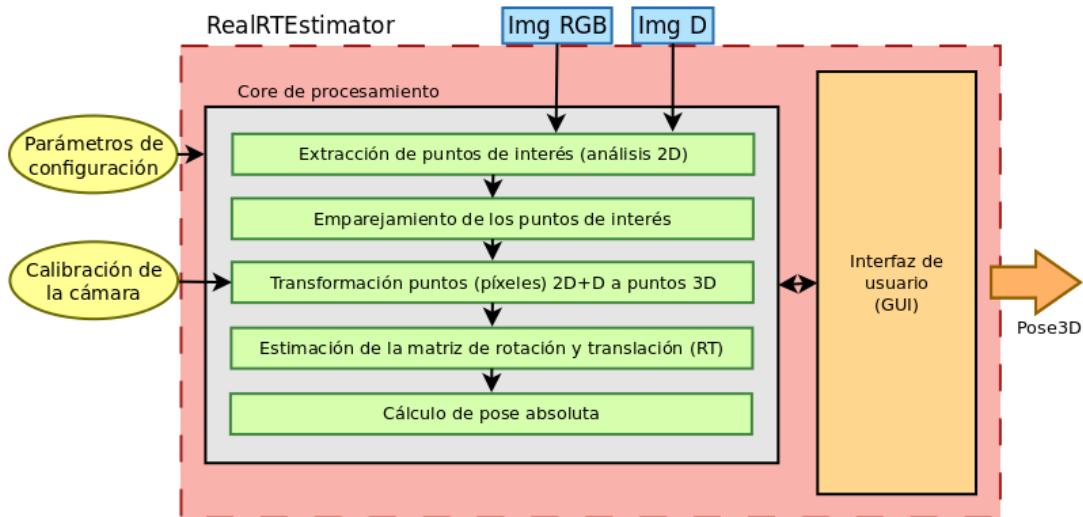


FIGURA 4.2: Diagrama del componente RealRTEstimator.

- Cálculo de movimiento. Es decir, estimación de la matriz de rotación y translación (Matriz RT).
- Cálculo de pose 3D absoluta.

En las siguientes secciones desglosaremos el funcionamiento de estos diferentes bloques funcionales.

4.2. Análisis 2D

El primer bloque del componente RealRTEstimator es el de análisis 2D. A partir de dos imágenes; la imagen de color y la de profundidad, se procede a la extracción de puntos de interés.

4.2.1. Detección de puntos de interés

El término puntos de interés o detección de características (*Feature Detection* en inglés) hace referencia a la tarea de localizar en una imagen puntos relevantes o característicos. Estos puntos suelen ser comunes y son fáciles de seguir de fotograma en fotograma.

Para entender cuales son estos puntos característicos podemos observar un ejemplo sencillo en la Figura 4.3. El cuadrado azul se encuentra en una área plana, y es difícil de seguir o encontrar. En cualquier lugar por donde se desplace parecerá que es el mismo. Para el cuadrado negro, que es un borde, igual para el desplazamiento lateral, sin embargo, para el desplazamiento vertical el punto ya cambia. Por último está el cuadrado rojo, que es una esquina. Para cualquier desplazamiento de esta figura, el punto ya es diferente, lo que significa que ese punto en la figura es único y por lo tanto vamos a poder identificarlo o seguirlo en diferentes imágenes. Así pues, las esquinas suelen ser candidatos idóneos para la detección puntos de características en una imagen (en algunos casos las manchas también pueden ser consideradas buenas zonas).

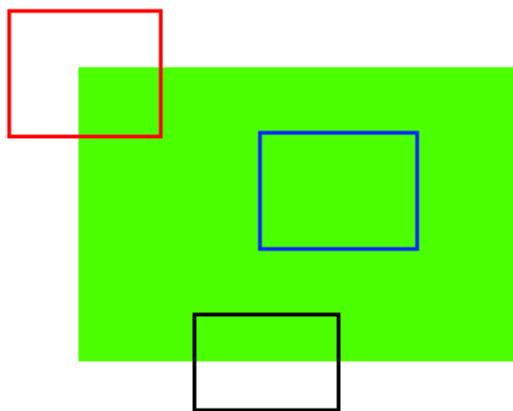


FIGURA 4.3: Ejemplo sencillo de puntos característicos.

Una vez entendido el concepto, el siguiente paso consiste, en averiguar cómo encontrar estos puntos de interés en una imagen real. Por ejemplo, una manera sencilla de hacerlo es buscar las regiones en las imágenes que contienen una gran variabilidad cuando son desplazadas (una pequeña distancia) hacia todas las direcciones de los alrededores.

Existen multitud de implementaciones para calcular estas características en las imágenes. Uno de los primeros intentos en encontrar estas esquinas fue hecho por Chris Harris y Mike Stephens (Harris y Stephens, 1988). El método, llamado *Harris Corner*

Detector transforma la simple idea a una fórmula matemática (4.1) que básicamente encuentra la diferencia en intensidad por un desplazamiento (u,v) en todas las direcciones.

$$E(u, v) = \sum_{x,y} w(x, y) [I(x + u, y + v) - I(x, y)]^2 \quad (4.1)$$

Donde $w(x,y)$ es una ventana rectangular o gaussiana e $I(x,y)$ corresponde a la intensidad. Aplicando algunos cálculos matemáticos que no vamos a entrar en detalle podemos llegar a la ecuación (4.1) básica que determina si una ventana contiene una esquina o no.

$$\begin{aligned} R &= \det(m) - k(\text{trace}(M))^2 \\ R &= \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2 \end{aligned} \quad (4.2)$$

λ_1 y λ_2 son los autovalores de la matriz M , que determinarán si una región es esquina, borde o zona plana.

- Cuando $|R|$ es pequeño, que sucede cuando λ_1 y λ_2 son pequeños, la región es plana.
- Cuando $R < 0$, que sucede cuando $\lambda_1 \gg \lambda_2$ o viceversa, la región es un borde.
- Cuando R es grande, que sucede cuando λ_1 y λ_2 son grandes y más o menos iguales, la sección es una esquina.

Más tarde, J. Shi y C. Tomasi hicieron una pequeña modificación que obtuvo mejores resultados comparados con los obtenidos en el detector de Harris (Shi y Tomasi, 1994). El resultado del detector *Shi-Tomasi Corner Detector* se puede ver en la ecuación (4.3)

$$R = \min(\lambda_1, \lambda_2) \quad (4.3)$$

Si R es mayor que un determinado umbral, o dicho de otro modo; solo cuando λ_1 o λ_2 se encuentran por encima de un valor mínimo λ_{\min} , se considera que cierta región es esquina. En la Figura 4.4 se puede observar el resultado de aplicar dicho algoritmo en una imagen.

Existen varias implementaciones para el cálculo de características de una imagen. A parte de las mencionadas, OpenCV proporciona entre otras **SIFT** y **SURF** que son las que hemos usado para el trabajo ya que permiten además de la detección de puntos de interés, el cálculo de descriptores.

4.2.2. Cálculo de descriptores

Una vez que se conoce el punto de interés, necesitamos asignarle una huella, algo característico que nos permita encontrar el mismo en otra imagen. Para ello, se procede al cálculo de descriptores (*Feature Description* en inglés).

Consiste en definir la región alrededor del punto de interés para poder buscar el punto con la misma región en otra imagen. Es decir, se guarda una descripción de

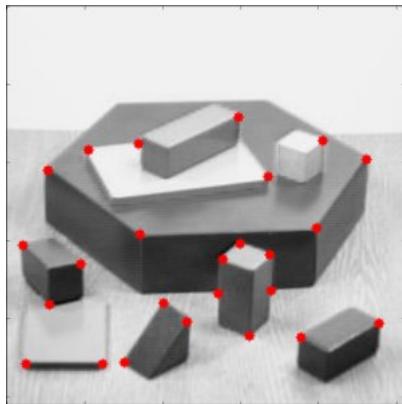


FIGURA 4.4: Resultado de encontrar las mejores 25 esquinas de la imagen con *Shi-Tomasi Corner Detector*.

la región del punto dado y se busca el mismo (o el que más se parezca) a otro punto perteneciente a otra imagen.

Una vez localizado el punto se podrá llevar un seguimiento de dónde está ese punto en otra imagen. No en todos los casos, se va a encontrar un descriptor perfecto para un cierto punto, por lo que al estudiar los emparejamientos se evaluará cuanto se parecen los descriptores entre sí. Esto lo veremos con detalle en la siguiente sección.

SIFT

SIFT (Scale-Invariant Feature Transform) soluciona uno de los problemas que se encontraban en los métodos anteriormente mencionados. Los métodos hasta ahora vistos para el cálculo de puntos de interés o esquinas, se suponen invariantes a la rotación, es decir, incluso si la imagen es rotada es posible encontrar las mismas esquinas. Esto es así porque una esquina sigue siendo una esquina si la imagen a sido rotada. Sin embargo, no contemplan los cambios de escala, un esquina no puede ser una esquina si la imagen ha sido escalada. En la Figura 4.5 podemos ver un ejemplo de este hecho; una esquina en una pequeña imagen con una ventana no lo es cuando la imagen se amplia y se usa la misma ventana.

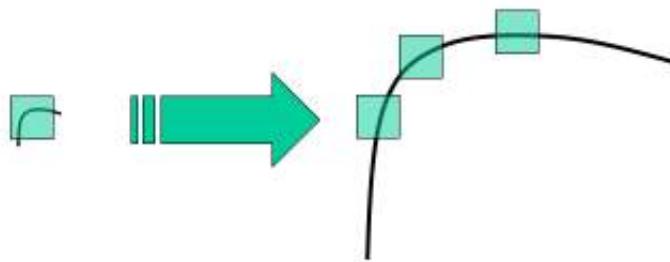


FIGURA 4.5: Ejemplo de diferentes puntos con escala

SIFT nace para proveer esta carencia de mano de D. Lowe (D.Lowe, 2004). Un algoritmo escalarmente invariante que localiza puntos de interés y calcula descriptores. Hay principalmente 4 etapas básicas en el algoritmo de SIFT:

1. Extrema detección en espacio-escala

Para poder detectar características en diferentes escalas es necesario poder variar el tamaño de la ventana a ampliar. Para ello se utiliza un filtro de espacio-escala; un filtro LoG (Laplaciana de una Gaussiana) que con diferentes valores de σ es capaz de detectar puntos de interés para diferentes escalas. σ actúa como un parámetro de escala, para bajos niveles de σ la gaussiana devuelve altos valores para las pequeñas esquinas, sin embargo, altos valores de σ encajan bien para grandes esquinas.

Por lo tanto se busca a lo largo de la imagen y en diferentes escalas para encontrar el punto

Así pues a lo largo de la imagen y en diferentes escalas tenemos una lista de (x, y, σ) valores, donde (x, y) representa el espacio y σ la escala.

Sin embargo, el filtro LoG es muy costoso por lo que SIFT calcula una aproximación; la diferencia de gausianas con diferente σ y el proceso se repetirá para diferentes octavas ($k\sigma$) como se puede apreciar en la Figura 4.6.

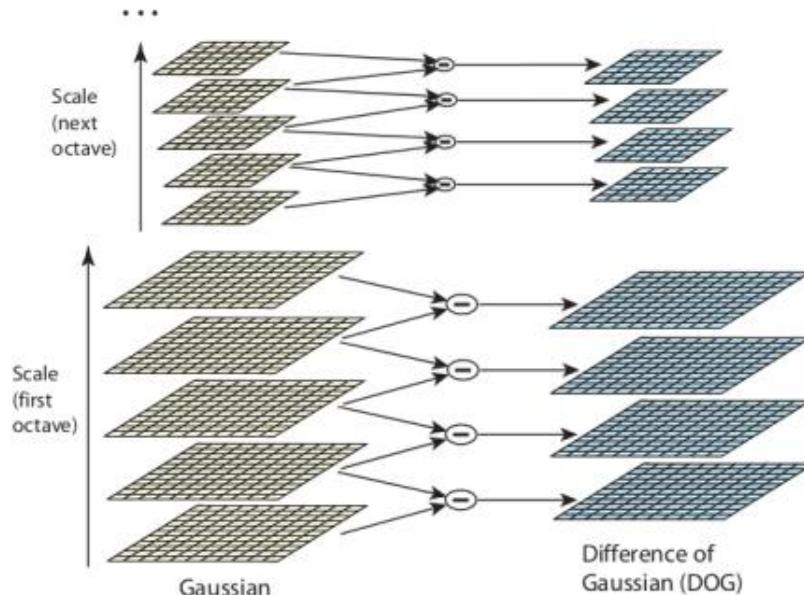


FIGURA 4.6: Proceso del cálculo de la diferencia de Gaussianas para diferentes octavas.

Una vez obtenida la diferencia de gaussianas (DOG) se calcula el *local-extrema*, por ejemplo, un píxel en una imagen es comparado con sus 8 vecinos y también con los 9 píxeles de la escala anterior y la posterior (Figura 4.7).

2. Localización de puntos de interés

Una vez que los puntos de interés se han localizado, se tienen que refinar a fin de obtener unos resultados más precisos. Por ello, se elimina cualquier punto de interés de bajo contraste, definido por el umbral *contrastThreshold* y los puntos caracterizados como bordes por el umbral *edgeThreshold*.

3. Asignación de orientación

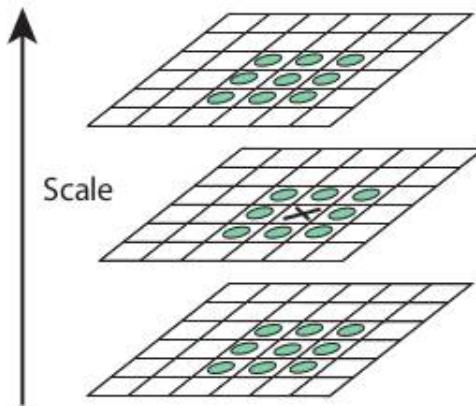


FIGURA 4.7: Proceso del cálculo de la diferencia de Gaussianas para diferentes octavas.

En este punto cada característica obtenida de la imagen se le asigna una orientación para mantener el algoritmo invariante ante la rotación. Se crean por tanto puntos característicos con la misma localización y escala pero en diferentes direcciones.

4. Descriptores

Aquí el descriptor del punto es creado con una ventana 16X16 alrededor del punto.

5. Emparejamiento de puntos

Los puntos de interés son emparejados en base a los vecinos. Se van seleccionando emparejamientos y cuando se encuentra un emparejamiento con menor distancia se selecciona. Los emparejamientos con una distancia mayor de 0.8 se eliminan.

La implementación de SIFT en OpenCV es sencilla, partiendo de dos imágenes en escala de grises tenemos:

```

1 SiftDescriptorExtractor extractor;
2 SiftFeatureDetector detector;
3 std::vector<KeyPoint> keypoints_1, keypoints_2;
4
5 // Keypoints detection
6 detector.detect( img_1, keypoints_1 );
7 detector.detect( img_2, keypoints_2 );
8
9 // Descriptors calculation
10 extractor.compute(img_1, keypoints_1, descriptors_1);
11 extractor.compute(img_2, keypoints_2, >descriptors_2);

```

SURF

SURF (Speeded Up Robust Features) podría ser la evolución y versión rápida de SIFT. Desarrollada por Bay, H., Tuytelaars, T. y Van Gool, L. (Bay H. y L., 2008) se introduce un nuevo algoritmo para la detección y cálculo de descriptores.

En vez de utilizar Laplaciana de la gaussiana (LoG) para encontrar el espacio de escala como SIFT. SURF va un poco más lejos y aproxima LoG con filtros de cuadros (*Box Filter*). En la Figura 4.8 se puede ver un ejemplo de tal aproximación.

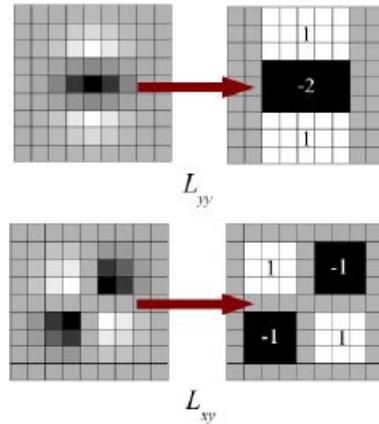


FIGURA 4.8: Proceso con filtros de cuadros, *Box filter*.

La principal ventaja es que la convolución con cuadros es más fácil de calcular con la ayuda de imágenes integrales. A parte, puede realizarse en paralelo para diferentes escalas.

Una vez calculado el escalado, se propone el cálculo de la orientación del punto de interés. Para obtener un punto invariante a las rotaciones, iluminación y orientación se utiliza *wavelet de Haar* en la dirección x e y en una región circular de radio $6s^1$. Después de haber evaluado las respuestas se busca la dirección predominante calculando la suma de todos los resultados dentro de una ventana con un ángulo de 60° . La región en la que se haya obtenido un mayor valor determinará la orientación buscada (Figura 4.9).

Para el cálculo del descriptor se construye una sección cuadrada en base a la orientación obtenida, alrededor del punto de interés y con un tamaño $20s \times 20s$. Esta región es dividida en sub-regiones de tamaño $4s \times 4s$ y para cada una de ellas se calcula la respuesta de *wavelet de Haar* de tamaño $2s$ tanto en x como para y . Se genera el siguiente vector:

$$v = \left(\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y| \right) \quad (4.4)$$

Con el vector representado, para cada característica detectada el descriptor con SURF contendrá 64 dimensiones. A menor dimensión, mayor velocidad de cómputo y de emparejamiento, pero se consigue mayor distinción entre características. Para esto SURF tiene una versión extendida con 128 dimensiones.

En resumen, SURF añade un montón de mejoras para incrementar la velocidad en cada paso. Pruebas experimentales han demostrado que puede ser tres veces más rápido de SIFT. Aunque en robustez es comparable con SIFT, SURF se comporta bien en imágenes borrosas y con rotación pero no tanto cuando se cambian los puntos de vista o con una diferente condición de iluminación.

¹La s hace referencia a un parámetro en el cómputo de descriptores de SURF que representa una escala que define los límites de las regiones circulares entorno al punto de interés.

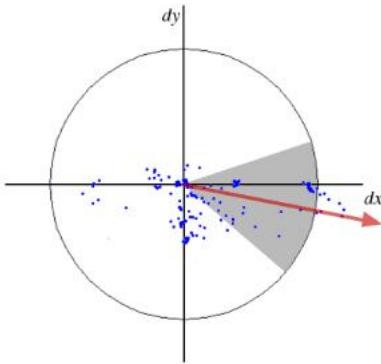


FIGURA 4.9: Cálculo de orientación con SURF.

La implementación en código es similar a la de SIFT:

```

1 //— Step 1: Detect the keypoints using SURF Detector
2 int minHessian = 400;
3
4 SurfFeatureDetector detector( minHessian );
5
6 std :: vector<KeyPoint> keypoints_1 , keypoints_2 ;
7
8 detector.detect( img_1, keypoints_1 );
9 detector.detect( img_2, keypoints_2 );
10
11 //— Step 2: Calculate descriptors (feature vectors)
12 SurfDescriptorExtractor extractor;
13
14 Mat descriptors_1 , descriptors_2 ;
15
16 extractor.compute( img_1, keypoints_1 , descriptors_1 );
17 extractor.compute( img_2, keypoints_2 , descriptors_2 );;
```

4.3. Emparejamiento (*matching*)

En esta sección abordaremos las distintas estrategias que se han implementado en el componente para el emparejamiento de puntos de interés. Los emparejamientos se harán por cada imagen/fotograma que llegue de la cámara después de la extracción de puntos de interés.

El componente está recibiendo continuamente imágenes del sensor, por lo que el cálculo, al igual que la extracción, se hará en cada iteración. Consiguiendo así una relación entre dos fotogramas consecutivos para analizar y posteriormente calcular el desplazamiento sucedido. Estos emparejamientos nos darán margen para desechar algunos peores emparejamientos y filtrar por los mejores utilizando diferentes técnicas. Así pues, se intentará coger los mejores emparejamientos entre una imagen en (t) y otra en $(t + 1)$.

Se presentarán dos soluciones para este problema; una primera solución que calcula el emparejamiento de puntos mediante un mecanismo de **Fuerza Bruta** y en segundo lugar el uso de la librería **FLANN**, ambas proporcionadas por OpenCV.

4.3.1. Fuerza Bruta

El mecanismo de Fuerza Bruta es simple. Se coge el descriptor de una de las características del primer fotograma (imagen en (t)) y se comprueba el parecido con todos los puntos de características del segundo fotograma (imagen en $(t + 1)$). Estos emparejamientos se evalúan a través de un parámetro de distancia. De todas las características, el descriptor que más se parezca al primero o el emparejamiento que tenga la menor distancia es el devuelto.

Para el cálculo de este emparejamiento se usará OpenCV. En primer lugar se tendrá que crear un objeto del tipo *BruteForceMatcher* y pasárle como parámetro el tipo de medida para calcular la distancia, ya que depende del tipo de descriptor a utilizar.

Una vez creado el objeto dos métodos importantes son *.match()* y *.knnMatch()*. El primero devuelve el mejor emparejamiento. El segundo devuelve los k mejores emparejamientos, donde k es definido por el usuario.

El siguiente ejemplo de código se puede observar como calcular los emparejamientos a través de OpenCV:

```

1 // matching descriptors
2 BruteForceMatcher<L2<float>> matcher;
3 vector<DMatch> matches;
4 matcher.match(descriptors1, descriptors2, matches);

```

matches por tanto será un array de objetos *DMatch* (objeto de emparejamiento) con los siguientes atributos:

- *DMatch.distance* - Parámetro de distancia entre descriptores. Cuanto menor distancia mejor emparejamiento.
- *DMatch.trainIdx* - Índice del descriptor de la segunda imagen con el resultado.
- *DMatch.queryIdx* - Índice del descriptor de la primera imagen a buscar.
- *DMatch.imgIdx* - Índice de la imagen resultado.

En Figura 4.10 podemos ver un ejemplo real de una prueba casera utilizando el algoritmo de Fuerza Bruta. Como se puede apreciar el tener que emparejar todos los puntos de una imagen al más parecido de la otra proporciona, incluso en una imagen muy parecida, errores que se van a tener que filtrar.

Por último, para la visualización OpenCV dispone del método *.drawMatches()* que a partir de las dos imágenes y los emparejamientos obtenidos nos ayuda a dibujar los mismos para su visualización. Colocando las dos imágenes a tratar en horizontal y dibujando las líneas con los emparejamientos de una imagen a otra. En el caso de usar y querer visualizar los k mejores emparejamientos existe también el método *.drawMatchesKnn*.

4.3.2. FLANN

FLANN (Fast Library for Approximate Nearest Neighbors) es una librería que contiene una colección de algoritmos optimizados para encontrar emparejamientos.

Esta librería de OpenCV, es una implementación del trabajo de Marius Muja y David G. Lowe (Muja y Lowe., 2009). Está pensada para trabajar con grandes conjuntos de

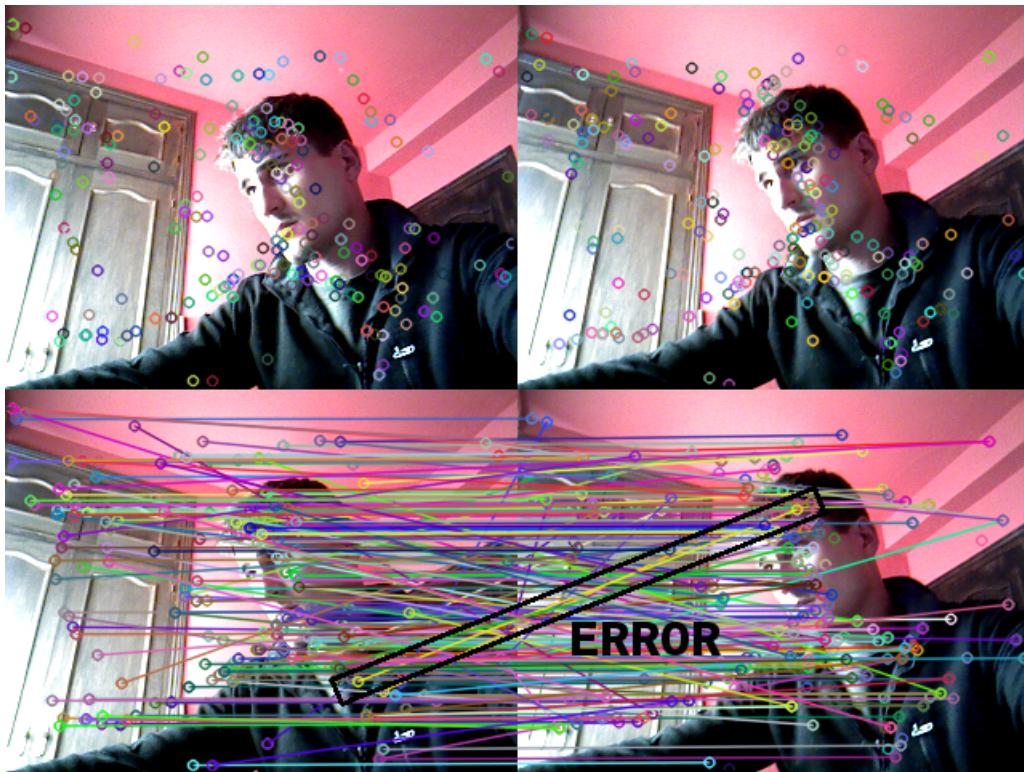


FIGURA 4.10: Detección y emparejamiento de puntos de interés con OpenCV usando SIFT para el cálculo de puntos de interés y descriptores y Fuerza Bruta para el cálculo de los emparejamientos.

datos y cuando los descriptores son representados por vectores de grandes dimensiones. Por lo que en estos entornos trabajará más rápido que el algoritmo de Fuerza Bruta.

FLANN provee de un sistema para elegir automáticamente el mejor algoritmo basado en la colección de datos. Dispone también de unos parámetros de entrada que permiten al usuario especificar la importancia de minimizar la memoria o el tiempo de compilación en lugar del tiempo de búsqueda.

La implementación en código con OpenCV es sencilla y muy parecida a la anterior:

```

1 //Matching descriptor vectors using FLANN matcher
2 FlannBasedMatcher matcher;
3 std::vector< DMatch > matches;
4 matcher.match( descriptors_1 , descriptors_2 , matches );

```

4.3.3. Resolución de errores

Sobre el tema de filtrado de errores se han usado dos estrategias; se han cogido de todos los emparejamientos un porcentaje relativamente pequeño donde se encuentran los emparejamientos más acertados, tomando como medida de calidad la distancia ofrecida por los diferentes algoritmos y se ha incluido un filtro de sobresalencia para el algoritmo de Fuerza Bruta.

- Mejores emparejamientos. De k emparejamientos obtenidos se ha implementado una función que ordena de menor a mayor la distancia de los emparejamientos obtenidos. Después y a través de la interfaz gráfica se podrá elegir el porcentaje de emparejamientos a emplear en la fase, que por defecto es un 20%, es decir, $(k * 0,2)$ puntos emparejados finales. Cuanto menor porcentaje mayor fiabilidad, pero menores resultados a pasar en la siguiente fase.

En la Figura 4.11 se puede ver el resultado de aplicar este método tras un desplazamiento horizontal. Se comprueba que el número de fallos se reduce considerablemente.

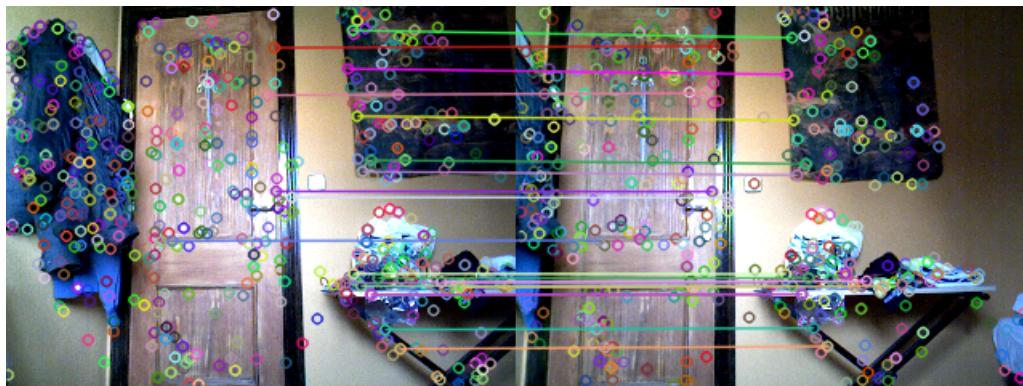


FIGURA 4.11: Emparejamiento de los mejores X puntos ordenados por distancia.

- Filtro de sobresalencia. Este filtro surge de la necesidad de corregir un error común que se encontraba en los mejores emparejamientos por distancia. Existen ciertas situaciones en las que hay características de una imagen muy similares a otras de otra imagen y que corresponden a errores en el emparejamiento.

En la Figura 4.12 se captura un ejemplo con dos de los mejores puntos medidos por distancia y se verifica que los dos mejores puntos se corresponden con el mismo punto en la imagen a buscar. Este error corresponde a una situación poco casual es origen de muchos errores en la estimación.

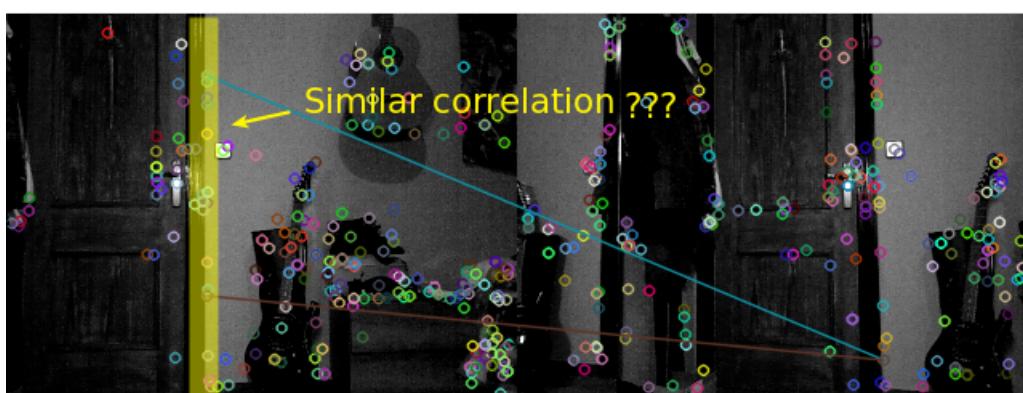


FIGURA 4.12: Detección y emparejamiento de los mejores dos puntos.

Se puede entender en la Figura 4.12 que el punto con el resultado final del emparejamiento (imagen derecha) tiene a nivel de descriptores una mayor similitud que los demás, de ahí ese resultado. Se puede decir por tanto, que los dos mejores puntos tienen una correlación muy similar, al igual que tendrán los demás puntos a lo largo del marco de la puerta subrayado en amarillo.

Para ello, se ha empleado el método `.knnMatch()` en el algoritmo de Fuerza Bruta que coge los k mejores puntos. En este caso, se cogen los 2 mejores emparejamientos de todos los puntos y si la diferencia entre estos dos mejores emparejamientos es muy similar (< 100) se desecha. Así pues, este filtro garantiza que el emparejamiento obtenido a través de los dos puntos es muy fuerte y no hay otro descriptor en la otra imagen con características similares.

El código para esta implementación es sencillo:

```

1 // Filtro de sobresalencia
2 vector<vector<DMatch>> matches_vector;
3 matcher.knnMatch(descriptors1, descriptors2, matches_vector, 2);
4 for (int i=0; i<matches_vector.size(); i++) {
5     outNumber = matches_vector[i][1].distance - matches_vector[i][0].
6         distance;
7     if (outNumber >= OUTSTANDING_DISTANCE) {
8         // Guardamos el emparejamiento
9     }

```

Donde `OUTSTANDING_DISTANCE` es el umbral mínimo de diferencia de distancias permitido, que por defecto se ha definido en 100.

Como diferencia, en esta ocasión como resultado del método `.knnMatch()` se obtiene un vector de vectores. Cada vector corresponde a un emparejamiento y por cada uno, otro vector con los k mejores emparejamientos.

4.4. Obtención de puntos 3D

Una vez obtenidos los puntos de interés y los mejores emparejamientos, se necesitará llevar a 3D los puntos calculados para el instante (t) ², para poder analizar en la siguiente fase el desplazamiento en tres dimensiones.

Esos puntos de interés en dos dimensiones, o mejor llamados; píxeles, se obtienen a partir de la imagen RGB que proviene del sensor, sin embargo, para este cálculo se necesitará además la imagen DEPTH correspondiente. O lo que es lo mismo, el mismo píxel o punto de la imagen a color debe corresponder con su homólogo en la imagen de profundidad. Se puede deducir que ambas imágenes deberán estar perfectamente sincronizadas para asegurarse de que ambas se corresponden con el mismo instante de tiempo. En la Figura 4.13 tenemos el diagrama de transformación de los puntos.

Para conseguir la información 3D se ha utilizado la librería de Progeo y su modelo de proyección *PinHole*. Una vez obtenidos los puntos 2D más su información de distancia, se busca la recta de retroproyección correspondiente a cada uno de los píxeles de la imagen que se quieran transformar. Después se calcula el punto 3D,

²Como explicaremos más adelante, no se necesitarán calcular los puntos 3D para el instante $(t - 1)$ ya que los puntos en ese instante ya se habrán calculado.

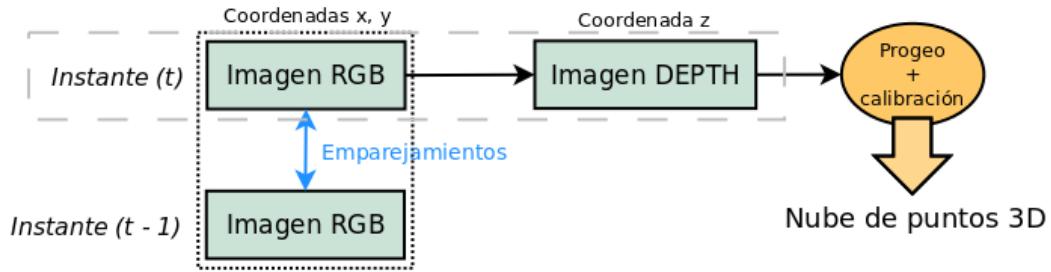
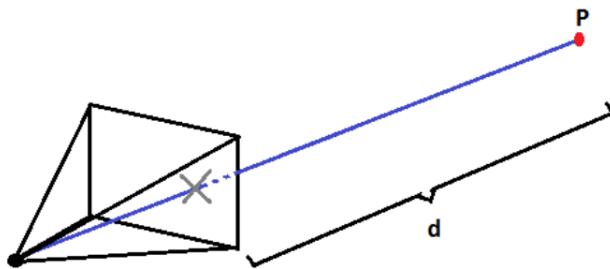


FIGURA 4.13: Diagrama de transformación a puntos 3D.

que será el que se encuentre a una distancia d de la recta de retroproyección (ej: punto P en la Figura 4.14).

FIGURA 4.14: Cálculo de punto 3D con la información de distancia d .

La transformación de un píxel a su homólogo en 3D es compleja. El proceso de reconstrucción está basado en un modelo proyectivo desde el centro óptico, por lo tanto, la distancia que se devuelve del sensor es la distancia perpendicular al plano imagen, y no la distancia real, tal y como se puede apreciar en la Figura 4.15. Para ello, si se quiere calcular la posición 3D asociada a un píxel cuya recta de retroproyección es la que une el foco de la cámara y un punto BP (Figura 4.16) el punto que se quiere calcular no será el punto P , punto a una distancia d del foco de la cámara y que nos da el sensor, sino el punto Pr que corresponde al punto 3D real.

Para calcular el punto Pr , se tiene que encontrar la intersección entre la recta de retroproyección y el plano D . Definimos el plano D como el plano con vector normal \vec{k} que pasa por el punto Q . El vector \vec{k} se obtiene como vector unitario que une el centro óptico de la cámara con el foco de atención, foa (*focus of attention*). Este vector es un vector perpendicular al plano imagen. El foa es un parámetro conocido que obtenemos en el proceso de calibración de la cámara junto con la orientación y el *roll*³ (parámetros extrínsecos). El proceso matemático es el siguiente:

Se calcula el vector unitario \vec{k} entre CAM y foa :

$$\vec{k} = \frac{\overrightarrow{CAMFOA}}{|\overrightarrow{CAMFOA}|} \quad (4.5)$$

³Movimiento del sensor sobre el eje central del sensor, paralelo al foa o foco de atención.

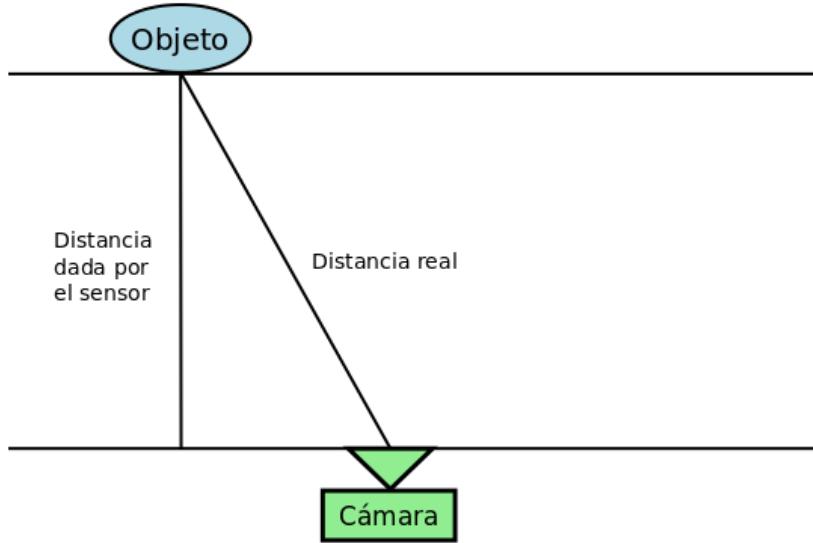


FIGURA 4.15: Distancia real y la dada por el sensor.

Una vez obtenido el vector \vec{k} se obtiene el punto Q que se encuentra a una distancia d del sensor, con las ecuaciones paramétricas de la recta:

$$\begin{aligned} Q_x &= CAM_x + d \cdot k_x \\ Q_y &= CAM_y + d \cdot k_y \\ Q_z &= CAM_z + d \cdot k_z \end{aligned} \tag{4.6}$$

Con esto se tendría el plano que contiene el punto Pr que es el que se quiere calcular. Es decir, el punto sobre la recta que pasa por BP a una distancia D (distancia corregida) del punto CAM .

Realizando los mismos cálculos, definimos el vector director de la recta $CAMB P$; \vec{v} :

$$\vec{v} = \frac{\overrightarrow{CAMB} \vec{P}}{|CAMB|} \tag{4.7}$$

Utilizando la ecuación paramétrica de la recta:

$$\begin{aligned} Pr_x &= CAM_x + D \cdot v_x \\ Pr_y &= CAM_y + D \cdot v_y \\ Pr_z &= CAM_z + D \cdot v_z \end{aligned} \tag{4.8}$$

Como el punto Pr se encuentra en el plano D , se tendrá que calcular la intersección de la recta recién calculada y el plano D . Para calcular el plano D se aplica la ecuación del plano con los datos obtenidos; Q , Pr y el vector normal al plano \vec{k} :

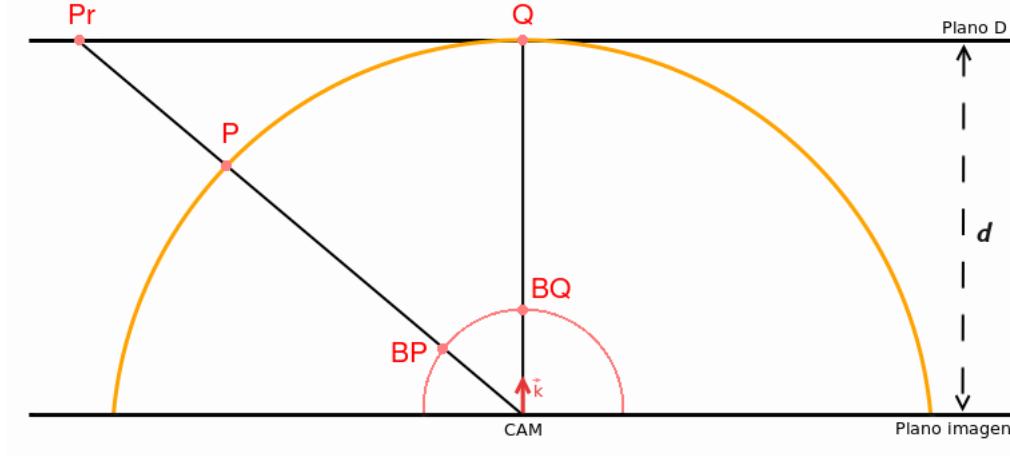


FIGURA 4.16: Ejemplo de corrección de distancia.

$$k_x(CAM_x + t \cdot v_x - Q_x) + k_y(CAM_y + t \cdot v_y - Q_y) + k_z(CAM_z + t \cdot v_z - Q_z) = 0 \quad (4.9)$$

Despejando t :

$$t = \frac{-k_x \cdot CAM_x + k_x \cdot Q_x - k_y \cdot CAM_y + k_y \cdot Q_y - k_z \cdot CAM_z + k_z \cdot Q_z}{k_x \cdot v_x + k_y \cdot v_y + k_z \cdot v_z} \quad (4.10)$$

Por último, aplicando el resultado de t sobre la ecuación 4.8 se obtiene el punto real buscado Pr .

4.5. Cálculo de movimiento

Una vez calculados los puntos 3D, el siguiente bloque del componente realizado es la estimación de movimiento. En este caso hay que calcular continuamente el movimiento relativo entre los instantes $(n - 1)$ y (n) . Esto permite ir siguiendo la trayectoria y la orientación seguida por el sensor a lo largo del tiempo.

4.5.1. Matriz RT

La trayectoria seguida se definirá por una **matriz RT** (Rotación + Translación) 4x4 de la manera que muestra la Figura 4.17.

Donde cualquier rotación podrá ser expresada como combinación de tres rotaciones por cada eje:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \quad (4.11)$$

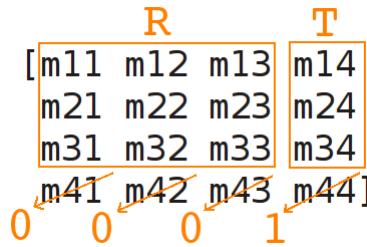


FIGURA 4.17: Matriz RT. Donde R corresponde a la rotación y T a la traslación.

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \quad (4.12)$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (4.13)$$

Y tres desplazamientos:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = R \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow x' = x + t_x; \quad y' = y + t_y; \quad z' = z + t_z \quad (4.14)$$

Se puede deducir, por tanto, que la matriz definirá en total seis grados de libertad para el movimiento del sensor.

4.5.2. Cálculo RT (SVD)

Al calcular los puntos 3D a través de *progeo* desde el sensor se tiene siempre los puntos en coordenadas **relativas**, por lo que al mover la cámara si calculamos de nuevo otros puntos estos se encontrarán en el mismo sistema de referencia. Por lo tanto, a parte de encontrar los puntos en 3D en relativas, habrá que hacer los cálculos para encontrar esos puntos en coordenadas **absolutas**.

En el instante inicial se toma como sistema de referencia absoluto el la cámara (matriz unidad). A partir de ahí se empezará a calcular el sistema de referencia absoluto de los nuevos puntos en cada instante.

En coordenadas absolutas se encontrarán en la misma posición los mismos puntos en diferentes instantes de tiempo. Como se puede apreciar en la Figura 4.18 el desplazamiento del sensor en coordenadas absolutas muestra:

- Puntos absolutos correspondientes únicamente al instante $(n - 1)$.
- Puntos absolutos correspondientes únicamente al instante (n) .
- Puntos absolutos correspondientes tanto al instante $(n - 1)$ como (n)

Esos puntos comunes son por tanto los utilizados para el cálculo de movimiento. Provenientes del emparejamiento entre píxeles de las imágenes del bloque anterior.

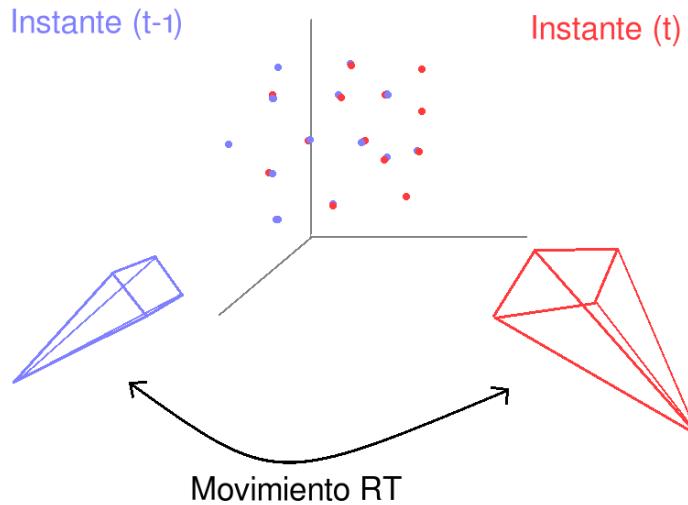


FIGURA 4.18: Cálculo visual de movimiento en coordenadas absolutas.

Como entrada en este bloque tendremos:

1. Los puntos relativos provenientes de la cámara en el instante (t).
2. Los puntos absolutos correspondientes al instante ($t - 1$).

La nube de puntos guardada será almacenada en un vector del siguiente tipo:

```

1 struct myPoint {
2     int x;
3     int y;
4     jderobot::RGBPoint rgbPoint;
5 };
6 std::vector<myPoint> myPrevPoints;
```

Donde se guardarán los píxeles y su correspondiente punto 3D en coordenadas absolutas, calculado de la iteración anterior. Con esto conseguimos la Matriz RT con un desplazamiento absoluto.

Para los cálculos de la matriz RT se ha usado la librería *Eigen* con la descomposición en valores singulares (SVD), ya que permite resolver sistemas sobredimensionados. En concreto se ha usado la clase *JacobiSVD* para la descomposición de una matriz rectangular.

Partiendo de la nube de puntos en coordenadas absolutas (mundo) del instante ($n - 1$) y la nube de puntos en coordenadas relativas (cam) en el instante (n); se calcula la matriz RT de la cámara con respecto al mundo:

$$RT_{cam}^{mundo} \cdot P_{pto(n-1)}^{mundo} = P_{pto(n)}^{cam} \quad (4.15)$$

Del mismo modo, para calcular la nube de puntos del instante actual y con coordenadas absolutas solo se tendrá que multiplicar por la inversa de la matriz calculada:

$$\left(RT_{cam}^{mundo} \right)^{-1} \cdot P_{pto(n)}^{cam} = P_{pto(n)}^{mundo} \quad (4.16)$$

Una vez llegados a este punto se guarda el resultado necesario para los cálculos de la siguiente iteración:

$$P_{pto(n)}^{mundo} \longrightarrow P_{pto(n-1)}^{mundo} \quad (4.17)$$

4.5.3. Optimización

El cálculo de movimiento ha sido uno de los puntos en los que se ha encontrado más problema ya que el error es acumulativo y una iteración con un error demasiado grande produce que en los siguientes instantes el cálculo sea erróneo. Por ello, hay que asegurarse de que todos los datos que vienen de los anteriores bloques, vengan con el menor error posible.

Para corregir el error en esta fase se ha optado por añadir dos filtros:

- Deshacer el cálculo con demasiado error espacial y de reproyección.
- Implementación de RANSAC.

Para el error espacial se ha medido la distancia entre los mismos puntos en el sistema de referencia absoluto para los instantes $(n - 1)$ y (n) .

Para el error de reproyección se mide la distancia entre píxeles de los diferentes instantes. Se calcula con *Progeo* la posición del nuevo píxel en el instante anterior para poder calcular la distancia para una misma imagen.

Si la distancia total es muy grande se desecha el cálculo y se comienza con otra instante de tiempo

La implementación con RANSAC se compone de 3 fases:

1. De los puntos emparejados se selecciona un porcentaje aleatorio
2. El cálculo RT se repite tantas veces como se deseé
3. De todas las iteraciones se busca la matriz que tenga menor error espacial o de reproyección.

Se puede encontrar que hay dos variables a elección del usuario; el porcentaje aleatorio de selección de puntos (por defecto; 80 %) y el número de repeticiones de cálculo de la matriz (por defecto; 10).

Como se puede deducir, este algoritmo es muy robusto frente a valores espurios, ya que si se realiza el cálculo con alguno de estos valores se desechará el resultado.

4.6. Interfaz gráfica

El componente desarrollado dispone de una interfaz gráfica en donde se pueden ver gráficamente los pasos realizados, así como el resultado de toda la lógica del sistema. Ha sido desarrollada con *glade* y la apariencia se puede observar en la Figura 4.19.

La interfaz gráfica, basada en GTK, permite la realización del procesado paso a paso o de manera automática:

1. Actualización de imagen. (Figura 4.20)

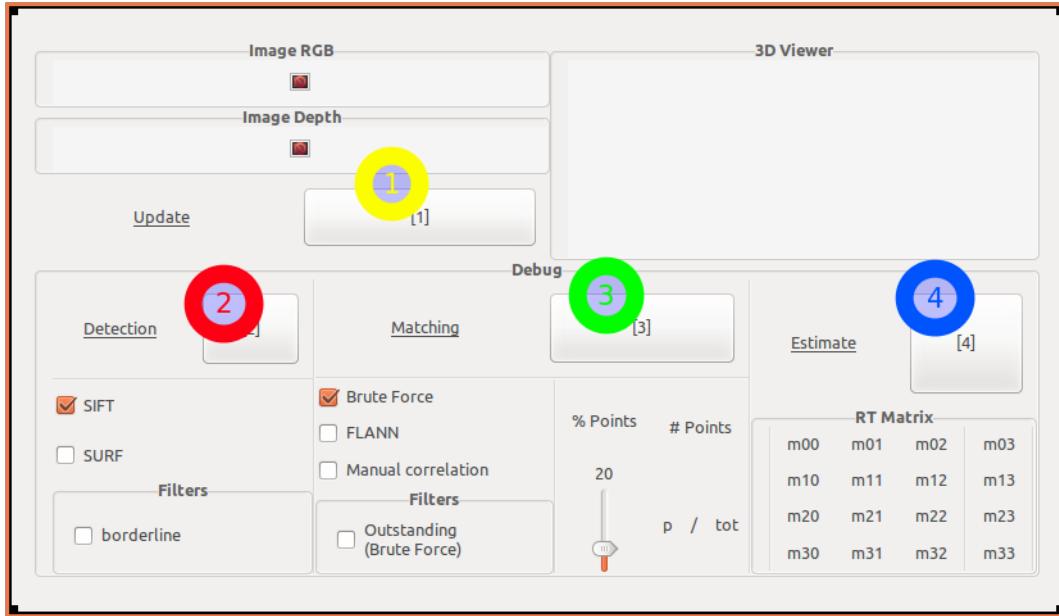


FIGURA 4.19: Captura del mapa de funcionalidades de la herramienta gráfica.

2. Detección de puntos. (Figura 4.21)
3. Cálculo de emparejamientos. (Figura 4.22)
4. Estimación de la matriz. (Figura 4.23)

La herramienta permite, además, la modificación de algunos parámetros de configuración, como pueden ser la elección algoritmos o filtros, así como el porcentaje de puntos emparejamientos a calcular.

Por último se ha implementado un visor 3D, donde se representa toda la información tridimensional para el sistema de referencia absoluto.

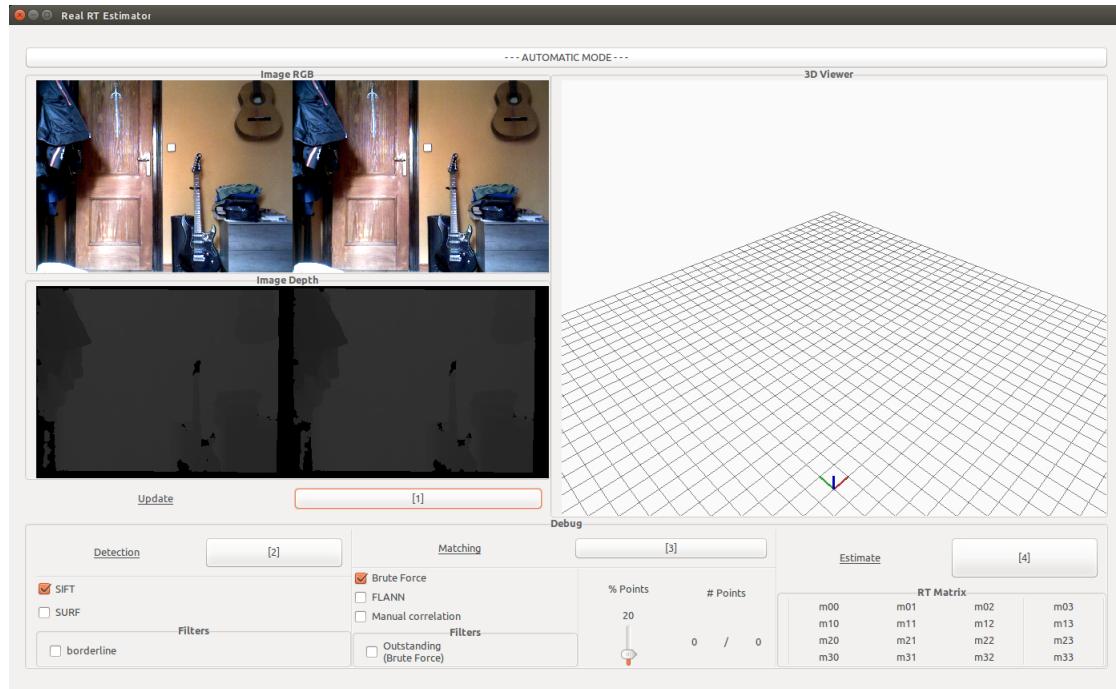


FIGURA 4.20: Paso 1: Actualización de imagen.

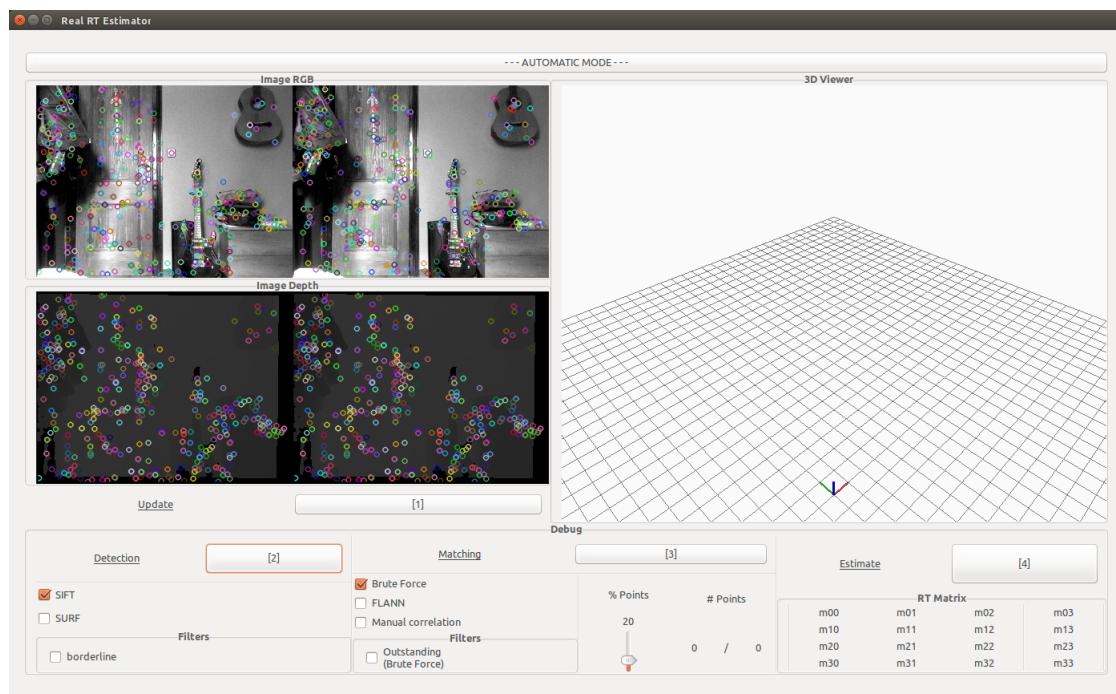


FIGURA 4.21: Paso 2: Detección de puntos.

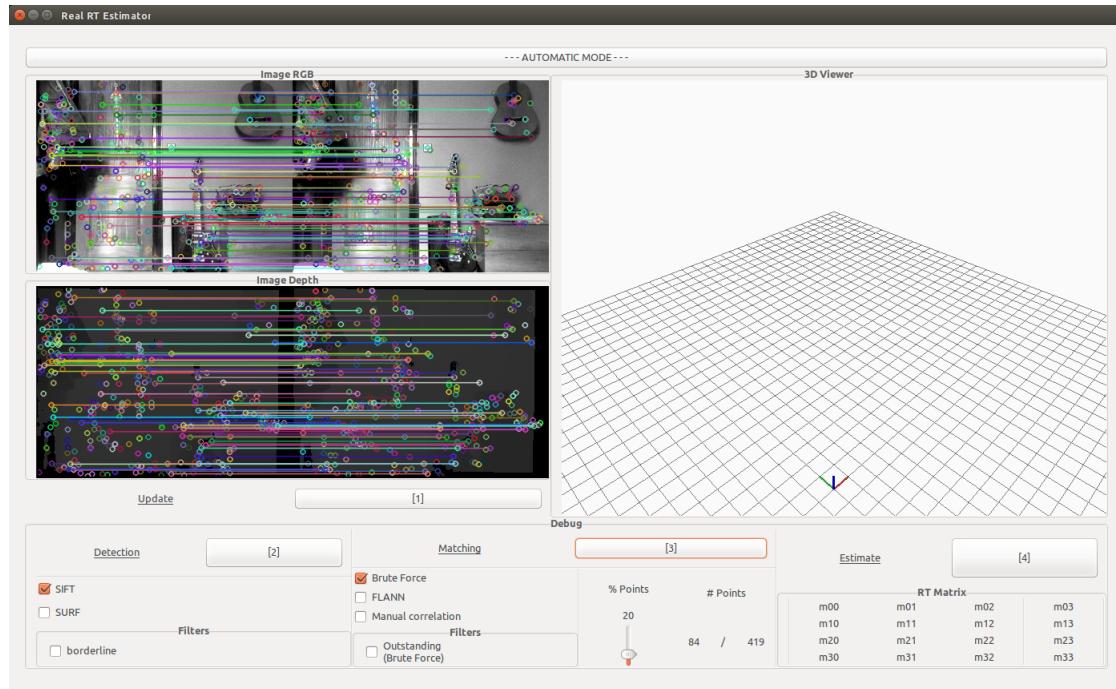


FIGURA 4.22: Paso 3: Cálculo de emparejamientos.

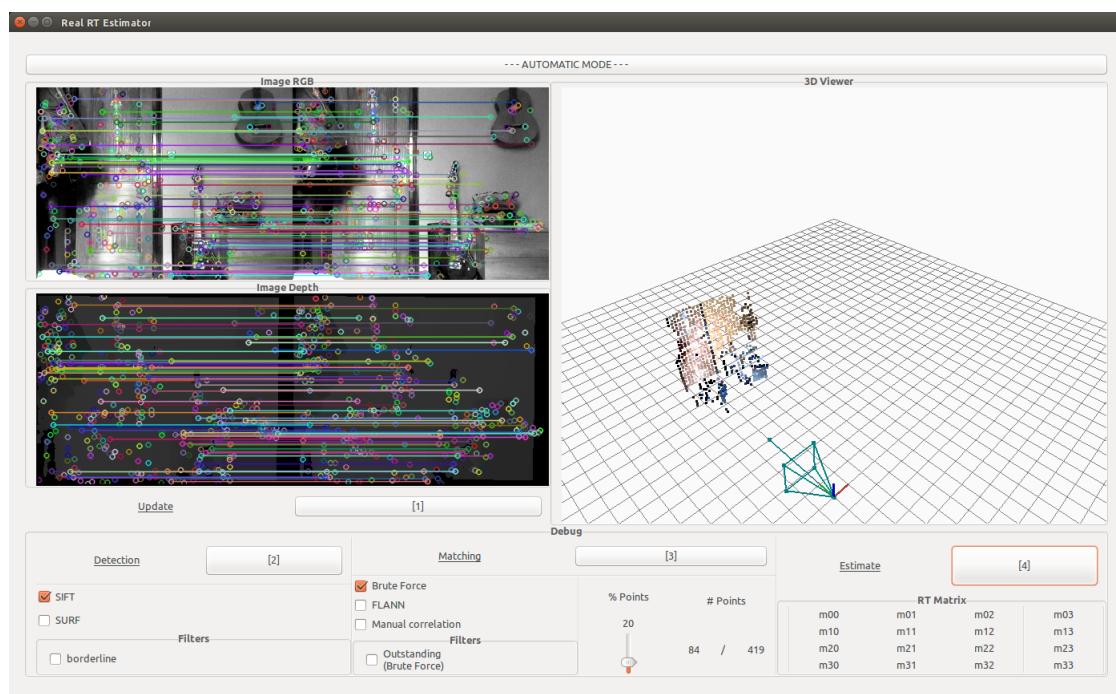


FIGURA 4.23: Paso 4: Estimación de la matriz.

Capítulo 5

Experimentos

Capítulo 6

Conclusiones

En los capítulos anteriores se ha mostrado una amplia descripción del problema abordado, las soluciones a las que se han llegado y los experimentos que se han desarrollado a fin de validar dicha solución.

En este capítulo se presentarán las conclusiones que se extraen del trabajo realizado y se propondrán una serie de líneas futuras de investigación que pueden continuar a partir de este proyecto fin de carrera.

6.1. Conclusiones

Después de un desarrollo de cerca de 5 mil líneas de código escritas en C++. Se puede decir que se ha conseguido en mayor o menor medida alcanzar cada uno de los objetivos planteados en el capítulo 2; un sistema capaz de averiguar la posición y orientación de un sensor RGBD que se mueve libremente por el espacio en tiempo real.

A continuación se repasan los diferentes subobjetivos marcados para recapitular y verificar lo realizado:

1. Actualización de los datos del sensor.

Este primer subobjetivo no tenía mayor complicación, se recogen las imágenes RGB y DEPTH del sensor cuando el componente ha terminado sus cálculos para dar comienzo a otra iteración o a otro instante de tiempo en el que se vuelve a hacer todo el proceso de estimación de movimiento.

2. Detección de puntos de interés.

El componente es capaz de extraer las características 2D de dos fotogramas consecutivos pertenecientes a dos instantes de tiempo, con técnicas como SIFT o SURF y filtro frontera.

Además de la extracción 2D se consigue transformar estas características de 2D a 3D, mediante la imagen de profundidad (D) asociada a la imagen RGB.

3. Emparejamiento de puntos.

El componente también es capaz de realizar los emparejamientos de los puntos de interés obtenidos del bloque anterior y poder ordenarlos de mayor a menor precisión. Se ha implementado también un filtro de sobresalencia que define que un emparejamiento sea único y no pueda confundirse con ningún otro.

4. Estimación de movimiento.

Se calcula la matriz RT a través de los emparejamientos obtenidos. Y a través de técnicas como el cálculo de error espacial, retroproyección y RANSAC se consigue una reducción de ruido importante.

5. Pruebas y experimentos.

Los experimentos mostrados en el capítulo 5...

6.2. Trabajos futuros

Para finalizar, se proponen algunas posibles líneas futuras de gran interés para la ampliación de este trabajo o mejoras que podrían realizarse.

- **Normalización y cierre de bucle.**

Para mejorar la estimación sería interesante, algún mecanismo de normalización de la matriz RT para eliminar el error acumulativo en lo máximo posible. Por ejemplo; si se sabe que los puntos obtenidos son de una pared o un terreno liso, se podría construir un plano y corregir dichos puntos en base a ese plano.

Así como algún mecanismo de cierre de bucle podría ser muy interesante para reajustar los cálculos y reducir en mayor medida ese error. Por ejemplo, cuando se ha pasado por una zona ya conocida y poder comparar el resultado de las dos veces.

- **Mejorar tiempo de cómputo.**

Aunque el trabajo ha sido implementado para reducir el tiempo de cómputo y encontrar un equilibrio entre velocidad y robustez, existe un margen de mejora que puede superarse. Por ejemplo, el cálculo de puntos de interés a través de FAST aligeraría el proceso.

- **Entornos complejos.**

Estaría bien investigar sobre entornos más complejos con presencia de objetos móviles. La presencia de objetos móviles no ha sido contemplada en la práctica, sin embargo, esta variable condicionará negativamente los resultados obtenidos. Por ello, y por que en el mundo que nos rodea no es estático podría ser interesante investigar esta línea de trabajo. Los espejos o reflejos podrían entrar también en este apartado como fuente de pruebas.