



Universidad  
Rey Juan Carlos

Escuela Técnica Superior de Ingeniería de Telecomunicación

**Ingeniería de Telecomunicación - Ingeniería Técnica en  
Informática de Sistemas**

**Proyecto Fin de Carrera**

# Odometría visual con sensor RGBD en JdeRobot

**Autor:** Javier Benito Díaz  
**Tutor:** José María Cañas Plaza

Curso académico 2016/2017



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

<http://jderobot.org/J.benitod-tfg>



Esta obra está distribuida bajo la licencia de "Reconocimiento-CompartirIgual 3.0 España (CC BY-SA 3.0 ES)" de Creative Commons.

Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-sa/3.0/es/> o envíe una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

© 2017 Javier Benito Díaz



# Agradecimientos

En primer lugar, agradecer a mi familia por su amor constante e incondicional, no solo durante mi formación, sino a lo largo de toda mi vida. Por animarme a estudiar y hacer siempre lo que quisiera.

Gracias a José María por su dedicación y empeño a lo largo de todo el desarrollo de este proyecto. Por la paciencia infinita y la gran capacidad de resolver cualquier problema.

También quisiera agradecer a mis compañeros de carrera, sobre todo a Mario y Fran, por todo el apoyo y porque siempre he podido contar con ellos para todo lo que necesitara.

Por último, gracias a Pilar por todo su cariño, ánimo y alegría constante especialmente cuando más lo necesitaba.

¡Gracias a todos! Sin ellos, esto no habría sido posible.



# Resumen

Con la llegada de las consolas de videojuegos de última generación se ha mejorado enormemente la experiencia de usuario. Uno de los avances más revolucionarios de los últimos años ha sido la aparición de los sensores RGBD, como el Kinect para la consola Xbox. Este sensor permite la interacción con la consola sin contacto físico, simplemente con movimientos del cuerpo.

Debido a las grandes ventajas que aportan este tipo de sensores, se ha usado en otros campos de investigación como la visión artificial y la robótica. La utilización de estos sensores ha permitido un avance en áreas como la autolocalización visual.

El presente proyecto fin de carrera se apoya en este contexto. Se ha diseñado y desarrollado un sistema de odometría visual basado en sensores RGBD. El sistema implementado cuenta con algoritmos capaces de estimar la posición y la trayectoria 3D del sensor en tiempo real, basándose en la información recogida por éste. El resultado es mostrado en una ventana OpenGL con el mapeado de objetos, la posición de la cámara y su recorrido. Además ha sido validado experimentalmente en un entorno real.

Para el desarrollo del componente se ha empleado el lenguaje de programación C++. Se ha utilizado la plataforma JdeRobot 5.4 y varias librerías externas como ICE para la interfaz de comunicación, OpenCV para el procesado de imágenes, Eigen para los cálculos de álgebra lineal y GTK+ para el desarrollo de la interfaz de usuario, entre otras.



# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Visión artificial . . . . .	1
1.2. Autolocalización visual . . . . .	4
1.2.1. Structure from Motion (SfM) . . . . .	5
1.2.2. Visual SLAM . . . . .	6
1.2.3. Odometría visual . . . . .	7
1.3. Autolocalización visual en el laboratorio de robótica URJC . . . . .	8
<b>2. Objetivos</b>	<b>11</b>
2.1. Descripción del problema . . . . .	11
2.2. Requisitos . . . . .	12
2.3. Metodología . . . . .	12
2.4. Planificación del trabajo . . . . .	13
<b>3. Infraestructura</b>	<b>15</b>
3.1. Sensores RGBD . . . . .	15
3.2. JdeRobot . . . . .	16
3.2.1. Biblioteca Progeo . . . . .	17
3.2.2. Biblioteca parallelIce . . . . .	18
3.2.3. Servidor OpenniServer . . . . .	18
3.2.4. Herramienta RGBDViewer . . . . .	18
3.2.5. Pose3D . . . . .	19
3.3. Biblioteca ICE de comunicaciones . . . . .	19
3.4. Biblioteca Point Cloud Library (PCL) . . . . .	19
3.5. Biblioteca OpenCV . . . . .	19
3.6. Biblioteca Eigen . . . . .	20
3.7. Biblioteca de interfaz gráfica GTK+ . . . . .	21
3.7.1. Glade . . . . .	21
3.8. OpenGL . . . . .	22
<b>4. Desarrollo</b>	<b>25</b>
4.1. Diseño . . . . .	25
4.2. Análisis 2D . . . . .	27
4.2.1. Detección de puntos de interés . . . . .	27
4.2.2. Cálculo de descriptores . . . . .	29
Descriptores SIFT . . . . .	29
Descriptores SURF . . . . .	31
4.3. Emparejamiento . . . . .	33
4.3.1. Emparejamiento por Fuerza Bruta . . . . .	33
4.3.2. Emparejamiento por FLANN . . . . .	34
4.3.3. Resolución de errores de emparejamiento . . . . .	35
4.4. Obtención de puntos 3D . . . . .	37

4.5. Cálculo de movimiento tridimensional . . . . .	40
4.5.1. Matriz RT . . . . .	40
4.5.2. Cálculo RT mediante SVD . . . . .	41
4.5.3. Optimización mediante RANSAC . . . . .	43
4.6. Interfaz gráfica . . . . .	45
<b>5. Experimentos</b>	<b>49</b>
5.1. Validación experimental . . . . .	49
5.1.1. Detección de puntos de interés . . . . .	49
Efecto del filtro de puntos frontera . . . . .	49
5.1.2. Cálculo de emparejamientos . . . . .	50
Necesidad del filtro de sobresaliente . . . . .	50
Funcionamiento en Traslación y Rotación . . . . .	51
5.1.3. Resolución 3D . . . . .	51
Validación de la traslación . . . . .	51
Validación de la Rotación . . . . .	53
Validación de trayectorias combinadas . . . . .	53
5.1.4. Acumulación de error en estático . . . . .	54
5.2. Tiempos de cómputo del procesamiento . . . . .	55
<b>6. Conclusiones</b>	<b>65</b>
6.1. Conclusiones . . . . .	65
6.2. Trabajos futuros . . . . .	66
<b>Bibliografía</b>	<b>67</b>

# 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 . . . . .	4
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
1.10.	Captura de pantalla, PFC de Luis Miguel . . . . .	9
1.11.	Captura de pantalla, TFM de Alberto López-Cerón . . . . .	9
1.12.	Captura de pantalla, PFC de Daniel Martín . . . . .	10
2.1.	Ciclo de vida en espiral . . . . .	13
3.1.	Sensor Kinect . . . . .	15
3.2.	Sensor Xtion . . . . .	16
3.3.	Modelo <i>Pinhole</i> . . . . .	17
3.4.	Captura de RGBDViewer . . . . .	18
3.5.	Detección y emparejamiento con OpenCV . . . . .	20
3.6.	Ejemplo con Eigen . . . . .	21
3.7.	Captura visualizador 3D . . . . .	23
4.1.	Esquema general del componente RealRTEstimator . . . . .	26
4.2.	Diagrama interno del componente RealRTEstimator . . . . .	26
4.3.	Ejemplo de puntos de interés . . . . .	27
4.4.	Ejemplo con <i>Shi-Tomasi Corner Detector</i> . . . . .	28
4.5.	Ejemplo de sistema escalarmente variante . . . . .	29
4.6.	Diferencia de gaussianas en SIFT . . . . .	30
4.7.	Ejemplo de local-extrema en SIFT . . . . .	31
4.8.	Ejemplo de <i>Box filter</i> en SURF . . . . .	32
4.9.	Cálculo de orientación con SURF . . . . .	32
4.10.	Captura real con SIFT . . . . .	35
4.11.	Captura de los mejores puntos con SIFT . . . . .	36
4.12.	Captura con error de los dos mejores puntos en SIFT . . . . .	36
4.13.	Diagrama con la obtención de puntos 3D . . . . .	37
4.14.	Cálculo de punto 3D con la información de distancia $d$ . . . . .	38
4.15.	Distancia real y la dada por el sensor . . . . .	38
4.16.	Ejemplo de corrección de distancia . . . . .	39
4.17.	Ejemplo de una nube de puntos mal y bien calculada . . . . .	40
4.18.	Matriz RT . . . . .	41
4.19.	Cálculo visual de movimiento en coordenadas absolutas . . . . .	42
4.20.	Captura del mapa de funcionalidades de la herramienta gráfica . . . . .	45

4.21. GUI: Actualización de imagen . . . . .	46
4.22. GUI: Detección de puntos . . . . .	47
4.23. GUI: Cálculo de emparejamientos . . . . .	47
4.24. GUI: Estimación de la matriz RT . . . . .	48
5.1. Extracción de características con SURF y con SIFT . . . . .	50
5.2. Captura de la extracción de puntos de interés con filtro frontera . . . . .	50
5.3. Captura con un error de emparejamiento sin filtro de sobresalencia . . . . .	51
5.4. Captura con los emparejamientos con imágenes sometidas a una rotación y a una traslación . . . . .	52
5.5. Ejes de coordenadas $x$ , $y$ y $z$ . . . . .	53
5.6. Punto de partida para las pruebas de traslación y rotación . . . . .	54
5.7. Prueba con desplazamiento en el eje $y$ . . . . .	57
5.8. Prueba con desplazamiento en los ejes $x$ y $z$ . . . . .	58
5.9. Explicación gráfica del movimiento <i>pitch</i> , <i>yaw</i> y <i>roll</i> . . . . .	59
5.10. Prueba con desplazamiento del ángulo <i>pitch</i> . . . . .	60
5.11. Pruebas con desplazamiento en los angulos <i>yaw</i> y <i>roll</i> . . . . .	61
5.12. Experimento con el cálculo de una trayectoria en forma de círculo . . . . .	62
5.13. Experimento con desplazamiento vertical hacia el eje $z$ y después una rotación del ángulo <i>roll</i> . . . . .	62
5.14. Capturas con ruido estático . . . . .	63

# Índice de cuadros

3.1. Especificaciones técnicas del Asus Xtion PRO LIVE . . . . .	16
5.1. Tiempos medios de detección en relación a los puntos de interés . . . . .	55
5.2. Tiempos medios de cálculo de SVD en relación a los emparejamientos . . . . .	56
5.3. Tiempos medios de cálculo con RANSAC en relación a las iteraciones para una media de 100 emparejamientos por iteración . . . . .	56



## Capítulo 1

# Introducción

En este primer capítulo se propone dar una visión general del contexto en que se encuadra el proyecto fin de carrera, que es la visión artificial. Dentro de ésta se abordará el problema al que vamos a hacer frente: La autolocalización visual, o dicho de otro modo, la estimación de la posición y orientación 3D de un sensor con seis grados de libertad (SLAM).

### 1.1. Visión artificial

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 y 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.

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, especialmente la visión, 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 cómo 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 el 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.<sup>1</sup>

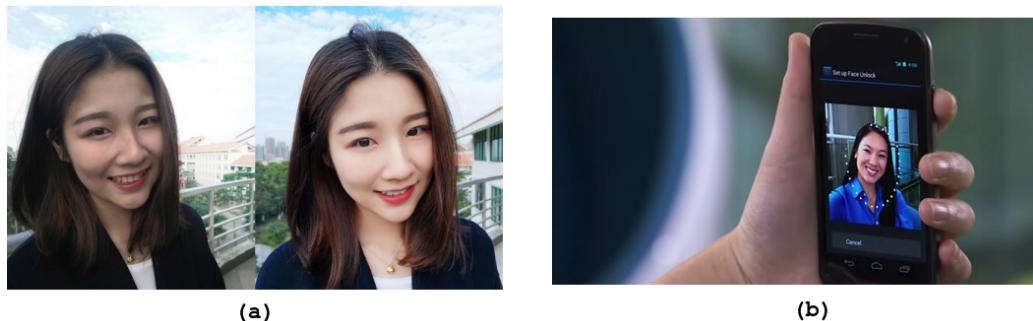


FIGURA 1.1: Embellecimiento de fotos (Meitu T8) (a). Reconocimiento facial (b).

Sin ir más lejos, la reciente aplicación que ha desatado revuelo en las diferentes redes sociales; FaceApp<sup>2</sup>. La aplicación, disponible 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 pretende extraer información sobre las manchas blancas que aparecen en la misma.

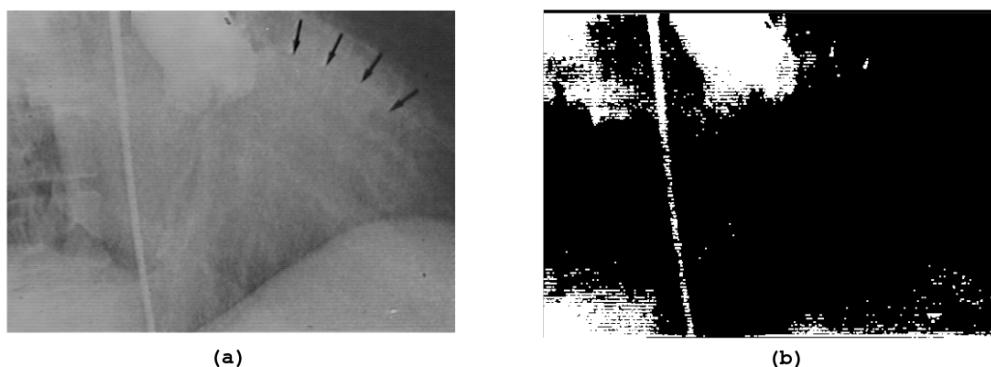


FIGURA 1.2: Radiografía inicial, con los puntos a analizar (a), imagen final procesada (b).

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

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

<sup>2</sup><https://www.faceapp.com/>

sistema EggInspector<sup>3</sup> 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.

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. Además, 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 recorridos por cada jugador en el terreno de juego, entre muchos otros.

Actualmente se pueden encontrar sistemas visuales 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.

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

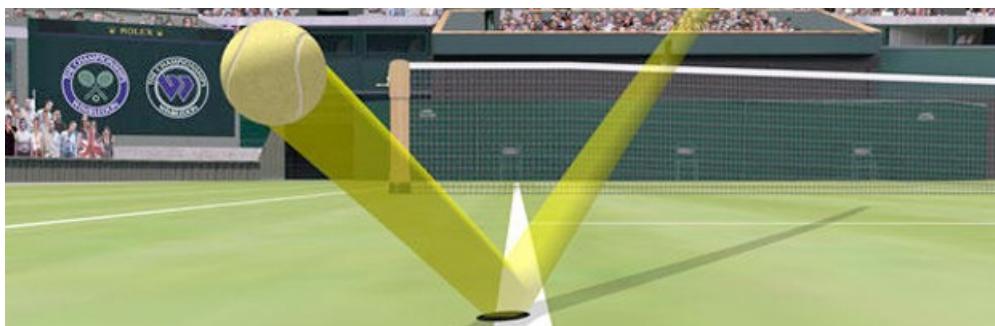


FIGURA 1.4: Ojo de Halcón en tenis

La empresa israelí **Mobileye**<sup>4</sup> 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.

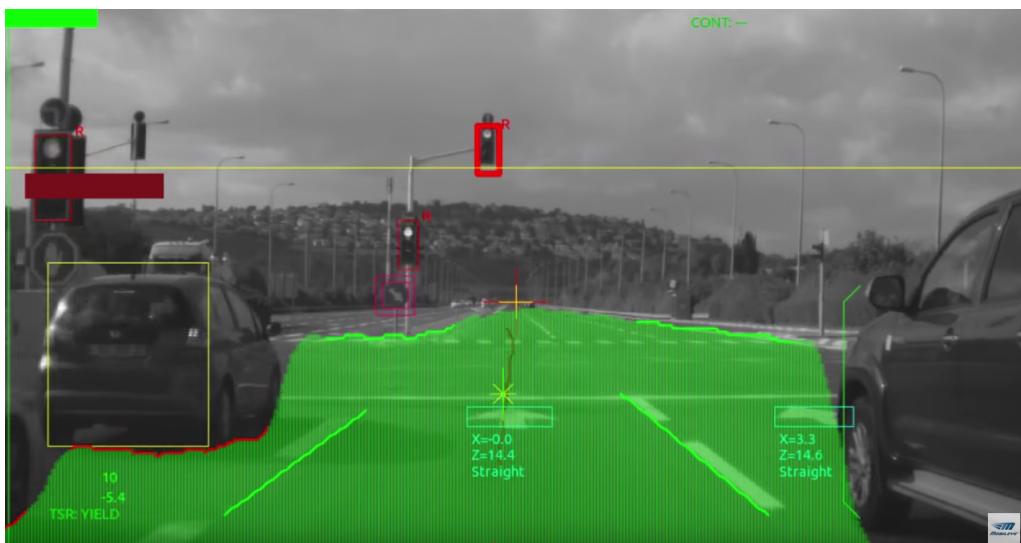


FIGURA 1.5: Vista del coche antes de parar en un semáforo.

## 1.2. Autolocalización visual

Dentro de la visión artificial se encuentra la autolocalización visual que consiste en conocer la localización 3D de la cámara en todo momento sólamente con las imágenes capturadas y sin disponer de ninguna información extra. Debido al gran abanico de posibilidades que abre resolver este problema, es uno de los retos más importantes dentro del campo de la robótica.

<sup>4</sup><https://www.mobileye.com/>

Esta técnica 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 parcialmente autónoma.

La autolocalización visual es una técnica que permite aplicaciones de realidad aumentada, que 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.

Aunque se ha popularizado con el juego de **Pokémon Go**<sup>5</sup>, 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.

Las técnicas de autolocalización han 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.

### 1.2.1. 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 procesan, lo que permite realizar optimizaciones para el cálculo de la trayectoria, como por ejemplo el ajuste de haces.

<sup>5</sup><http://www.pokemongo.com/es-es/>

Existen aplicaciones comerciales que utilizan estas técnicas como es el caso de la aplicación **PhotoTourism** (Noah Snavely y Szeliski, 2006) desarrollada por Microsoft. Consiste en el cálculo de la posición 3D en la que fueron captadas las imágenes, por ejemplo de un monumento, para después extraer el modelo 3D con el que el usuario puede interactuar libremente (Figura 1.7).

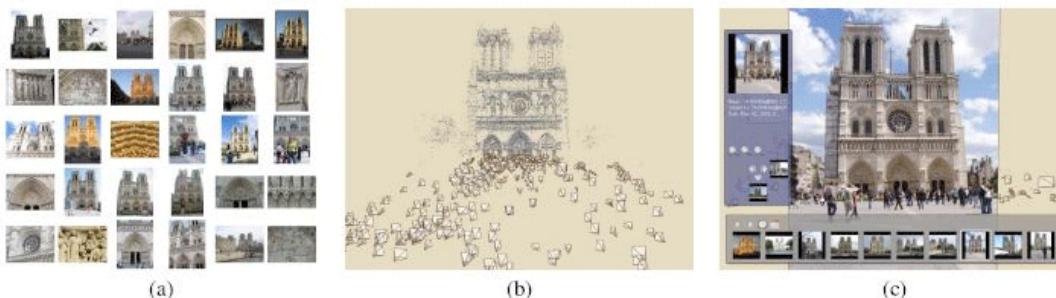


FIGURA 1.7: PhotoTourism: Se recogen una gran colección de imágenes (a), se reconstruyen los puntos 3D y los puntos de vista (b), por último la interfaz permite al usuario interactuar moviéndose a través del espacio 3D mediante la transición entre fotografías.

### 1.2.2. 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 encontrara 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 visual ha suscitado un gran interés en el campo de la robótica y se han propuesto muchas técnicas y algoritmos para dar solución al problema, como es el caso del artículo de 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 completamente estos problemas sigue siendo una tarea pendiente.

Uno de los trabajos más importantes en el ámbito es el de monoSLAM de Davison<sup>6</sup> (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

<sup>6</sup><http://www.doc.ic.ac.uk/~ajd/>

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.

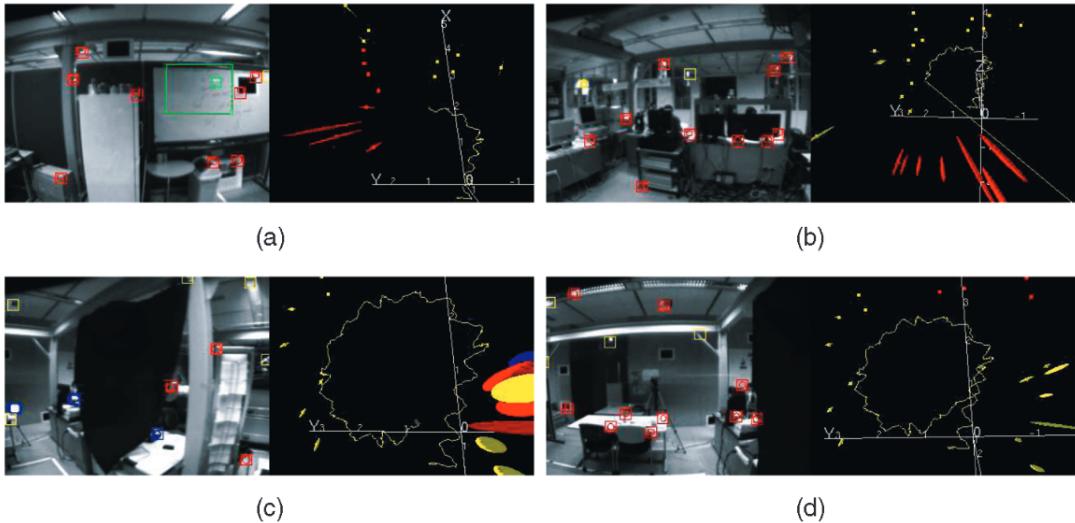


FIGURA 1.8: MonoSLAM: Un robot humanoide camina en una trayectoria circular de radio 0.75m. La estela amarilla muestra la trayectoria estimada del robot, y las elipses muestran los errores de localización.

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.

### 1.2.3. Odometría visual

Dentro de las familias de técnicas pertenecientes a Visual SLAM se encuentra la de odometría visual, que es la que abordaremos en este trabajo. Consiste en la estimación del movimiento 3D incremental de la cámara en tiempo real. Es decir, el cálculo de la rotación y traslación de la cámara a partir de imágenes consecutivas. Se trata de una técnica incremental ya que se basa en la posición anterior para calcular la nueva.

En 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: una vez calculados los puntos emparejados se calcula la matriz fundamental o esencial y descomponerla mediante SVD para obtener la matriz de rotación y traslación (RT) (Scaramuzza y Fraundorfer, 2011, Fraundorfer y Scaramuzza, 2012).

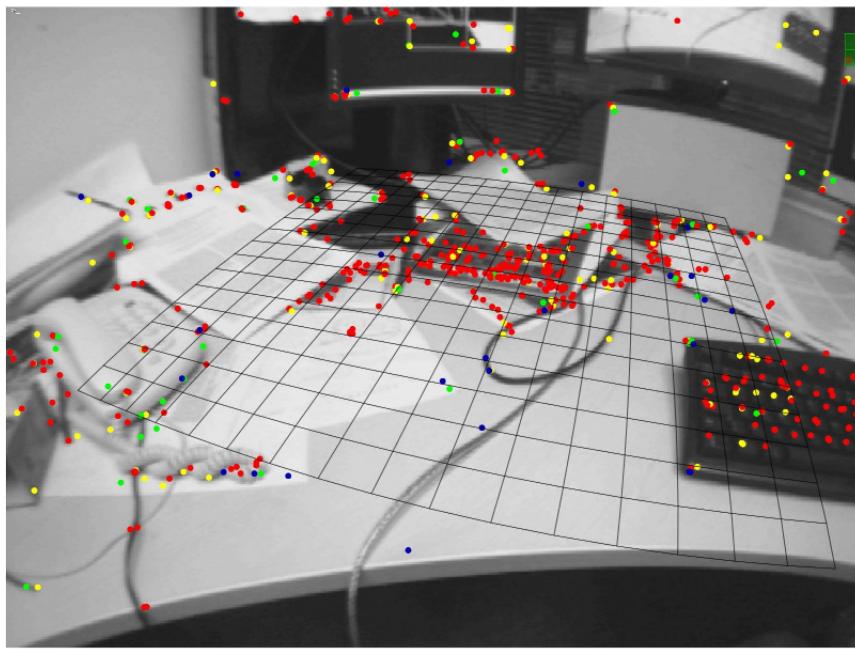


FIGURA 1.9: PTAM: Funcionamiento típico del sistema sobre un escritorio.

### 1.3. Autolocalización visual en el laboratorio de robótica URJC

En esta sección se recapitulan algunos de los proyectos, dentro del campo de la autolocalización, realizados por compañeros en el laboratorio de robótica de la Universidad Rey Juan Carlos. Sirven de contexto cercano a este Proyecto Fin de Carrera.

En el proyecto de Luis Miguel López Ramos (Ramos, 2010) se diseña y programa un algoritmo que estima en tiempo real la posición y orientación de una cámara móvil autónoma en un entorno estático, utilizando exclusivamente las imágenes obtenidas por la cámara. El algoritmo se valida experimentalmente haciendo uso de una cámara de videoconferencia real y en condiciones de laboratorio.

Se calcula, por tanto, la trayectoria realizada y se muestra en una ventana de gráficos OpenGL junto con un modelo de la cámara y las regiones de confianza de los puntos de referencia. (Figura 1.10).

Eduardo Perdices García (García, 2010) propone en su trabajo de fin de master un algoritmo de autolocalización para robots humanoides en la **RoboCup**, donde un grupo de robots autónomos deben jugar al fútbol de forma cooperativa variando su comportamiento en función de su posición en el campo, por lo que es muy importante que el robot conozca su posición en todo momento. A partir de los distintos sensores con los que cuenta el robot, como sensores de ultrasonido, sensores laser o cámaras, el robot tiene que estimar su posición en el mundo que le rodea.

En este proyecto se han desarrollado las técnicas necesarias para autolocalizar a un robot humanoide Nao<sup>7</sup> dentro de un campo de fútbol de la plataforma estándar de la RoboCup utilizando solo una cámara como sensor externo.

<sup>7</sup><http://aliverobots.com/nao/>

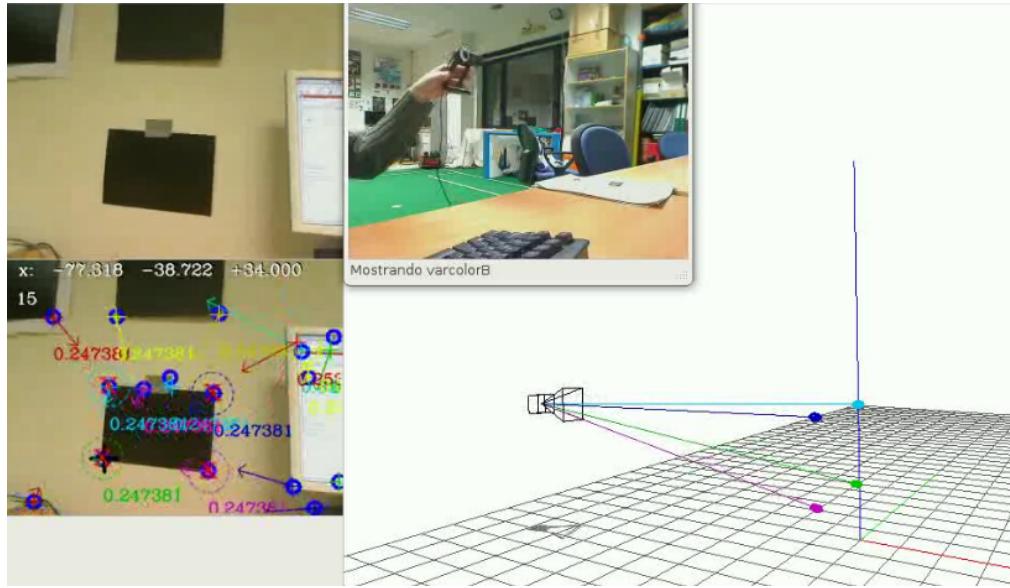


FIGURA 1.10: Captura de pantalla del componente desarrollado por Luis Miguel.

Alberto López-Cerón Pinilla (Pinilla, 2015) caracteriza un algoritmo de autolocalización visual basado en marcadores. Para ello, desarrolla un componente que a partir de la detección de balizas visuales en una imagen, estima la posición y la orientación 3D de la cámara, mostrando el modelo de la cámara resultante en una ventana OpenGL.

El algoritmo se ha validado experimentalmente realizando estudios de precisión en dos ámbitos: por un lado el de entorno simulado de Gazebo, haciendo uso de un modelo robótico virtual con sus cámaras asociadas, y por otro en un entorno real con una cámara de videoconferencia (Figura 1.11).

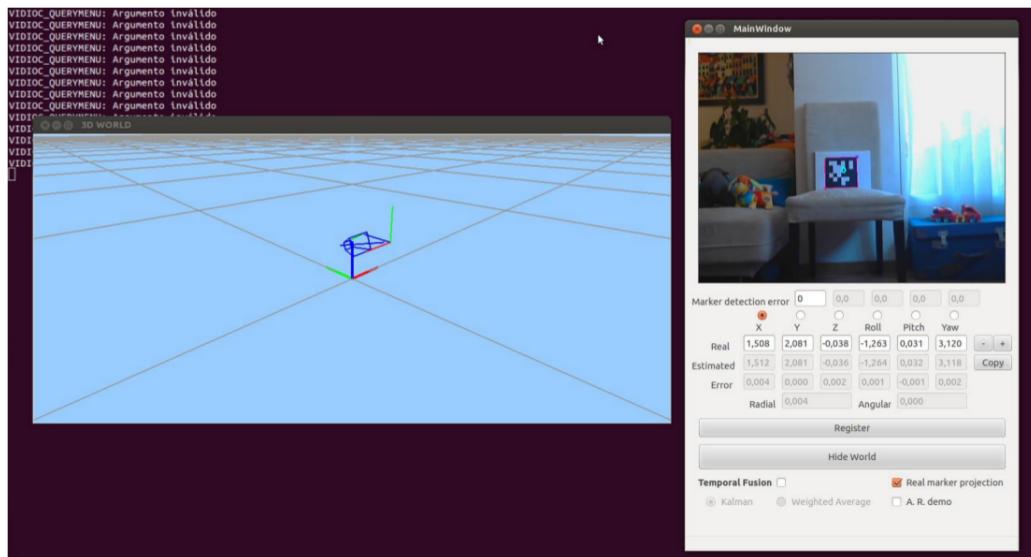


FIGURA 1.11: Captura de pantalla del componente desarrollado por Alberto López-Cerón.

En el proyecto fin de carrera de Daniel Martín Organista (Organista, 2014) se ha planteado un sistema de odometría visual, basado en sensores RGBD. El sistema desarrollado consta de algoritmos de estimación de la posición y trayectoria 3D de manera incremental en tiempo real, basándose en la información 3D ofrecida por el sensor RGBD. El sistema ha sido validado experimentalmente tanto en entornos reales como simulados (Figura 1.12).

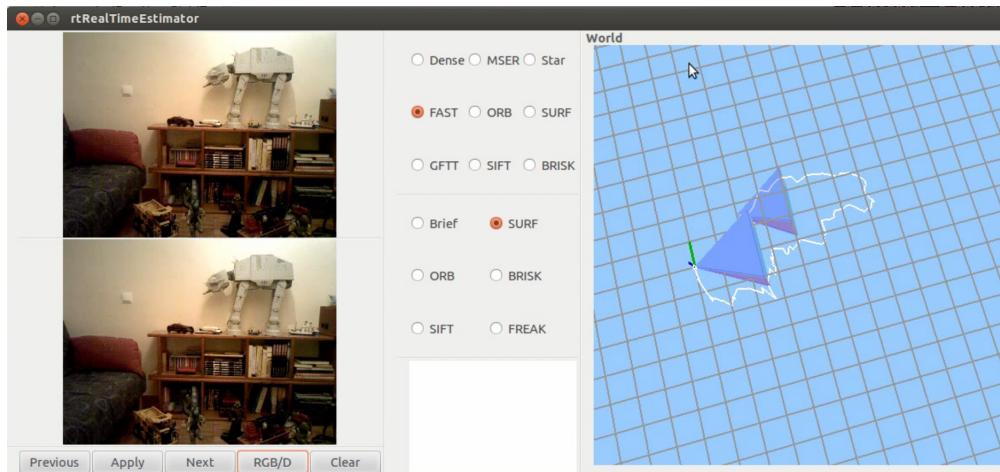


FIGURA 1.12: Captura de pantalla del componente desarrollado por Daniel Martín.

Por último, en el trabajo fin de master de Ignacio San Román Lana (Lana, 2015) se desarrolla un sistema clásico de odometría visual 3D en el que se emparejan puntos característicos entre fotogramas consecutivos o seguidos en el tiempo, para después optimizar un sistema de ecuaciones que estima la matriz fundamental. El resultado obtenido es mostrado en una ventana OpenGL en donde se puede ver la posición actual de la cámara y su recorrido.

El presente proyecto fin de carrera pretende unirse a los trabajos descritos en la búsqueda de algoritmos y técnicas de autolocalización visual, concretamente la odometría visual. En los próximos capítulos se detallará la solución adoptada.

El resto de la memoria consiste en 5 capítulos más:

- Objetivos - Se detallarán los objetivos del trabajo, así como la metodología llevada a cabo para la resolución del problema planteado.
- Infraestructura - Se mostrarán las bases y tecnologías empleadas.
- Desarrollo - Se explicará el desarrollo del componente realizado y se detallarán tanto las bases teóricas como los algoritmos implementados.
- Experimentos - Se muestran los experimentos realizados a fin de validar la solución elegida.
- Conclusiones - Se plantearán las conclusiones extraídas en base a los resultados obtenidos y se propondrán algunas líneas de trabajo futuras.

## Capítulo 2

# Objetivos

Una vez presentado el contexto, en este capítulo se describen los objetivos concretos que se pretenden alcanzar. Tras una descripción del problema abordado y los requisitos, 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 un programa que solucione el problema de visualSLAM, para un sensor RGBD, a través de técnicas de odometría visual incrementales. El sistema debe ser capaz de averiguar la posición y orientación 3D de un sensor RGBD que se mueve libremente por el espacio y que va alimentando al sistema con las imágenes obtenidas en tiempo real.

Este objetivo principal se ha articulado en los siguientes cuatro subobjetivos:

1. Detección de puntos de interés. Creación de un módulo encargado de recoger de manera dinámica las imágenes RGB y DEPTH del sensor.  
Desarrollo 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.
2. Emparejamiento de puntos. Desarrollo del cálculo de correspondencias entre los puntos de interés en el fotograma actual y en el fotograma previo.
3. Estimación del movimiento 3D. Implementación de la matemática necesaria para una vez tenidos los puntos emparejados entre dos fotogramas consecutivos, calcular la matriz de rotación y traslación (RT) que se necesitará para calcular en ese instante el movimiento incremental del sensor.
4. Pruebas y experimentos. El programa diseñado y construido se validará experimentalmente con un sensor real.

Cada uno de los subobjetivos incluirán sus pruebas unitarias y su interfaz gráfico, desde donde se podrá analizar y verificar cada uno de los hitos.

## 2.2. Requisitos

A parte de las funcionalidades mencionadas en el apartado anterior, la solución final del proyecto debe satisfacer además los siguientes requisitos:

- Desarrollo del proyecto haciendo uso de la plataforma **JdeRobot 5.4.0**, 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.
- Como la mayoría de componentes de JdeRobot están en C++, el trabajo también se ha desarrollado utilizando el mismo lenguaje de programación.
- Debe funcionar bajo sistema operativo linux, en este caso Ubuntu 14.04 LTS.
- Tiene que funcionar con un sensor RGBD real.

## 2.3. Metodología

Para abordar un proyecto de tal envergadura es necesaria una metodología de desarrollo adecuada para ir progresando de una manera ordenada y efectiva. 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 que 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, 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 evalúa de qué manera 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.

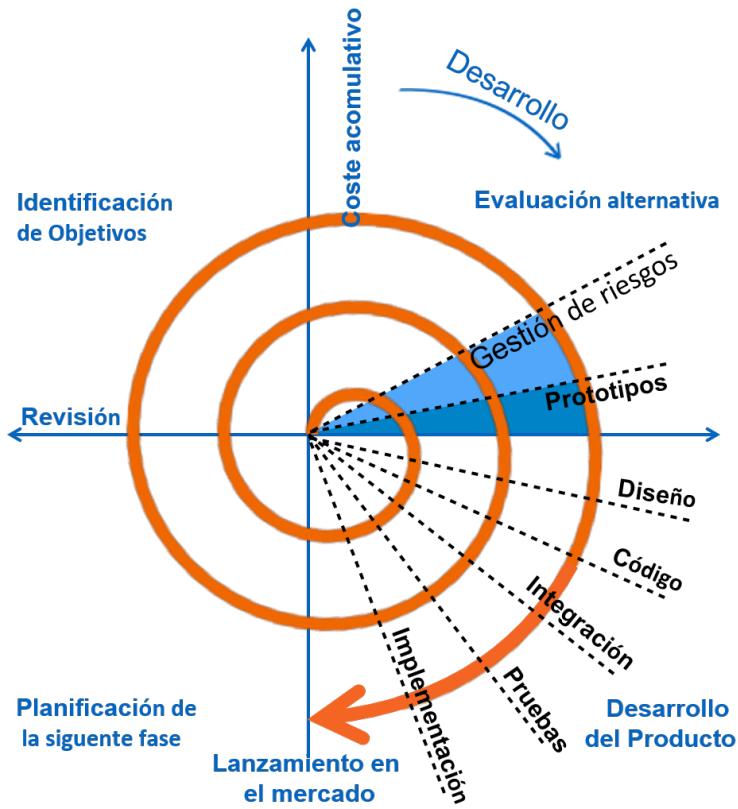


FIGURA 2.1: Ciclo de vida del desarrollo del *software* en el modelo espiral.

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 MediaWiki<sup>1</sup> con los detalles de las diferentes iteraciones, ayudadas a veces por imágenes y/o vídeos.

Para la gestión de código se han usado herramientas *software* de control de versiones, primeramente con Subversion (SVN)<sup>2</sup> y finalmente con GIT en un repositorio de GitHub<sup>3</sup>. Todo el código desarrollado es *software* libre y está accesible a quien lo quiera con licencia GPL3.

## 2.4. Planificación del trabajo

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

### 1. Familiarización de la herramienta JdeRobot.

<sup>1</sup><http://jderobot.org/J.benitod-tfg>

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

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

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 y técnicas de optimización.

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

- Se realizó un componente utilizando **PCL** para el cálculo de planos desde nubes de puntos.
- Se utilizó **Eigen** para la resolución mediante optimización de sistemas sobredimensionados de ecuaciones, con descomposición QR y en valores singulares (SVD).
- **GSL**, para la resolución de sistemas de ecuaciones, en este caso aplicado a un componente rectificador de imágenes.

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

Aquí se empezó a desarrollar el componente final. Esta fase corresponde al desarrollo de la solución a los subobjetivos (1) y (2), en donde se comenzó por la extracción de puntos de interés con SIFT de las imágenes y los diferentes algoritmos para los emparejamientos.

## 4. Registro entre dos nubes de puntos, relacionadas por una rotación, una traslación y ruido.

En esta fase se propuso una práctica dedicada al subobjetivo (3) de estimación de movimiento incremental, 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.

## 5. Integración y pruebas experimentales.

Esta fue la etapa más larga y costosa. 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.

## 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 3.1) 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.

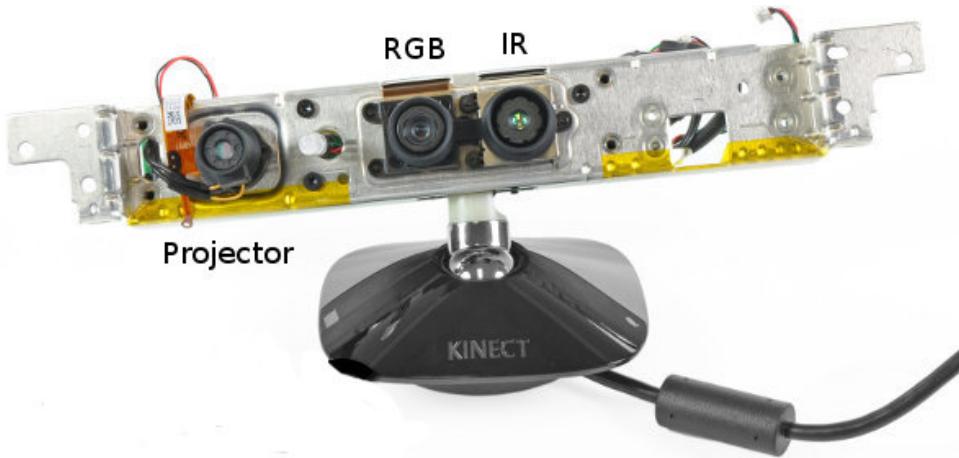


FIGURA 3.1: Sensor Microsoft Kinect

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* o NUI), reconstrucción y realidad virtual o cartografía 3D.

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




---

FIGURA 3.2: Asus Xtion PRO LIVE

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

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)

## 3.2. JdeRobot

JdeRobot es un proyecto desarrollado por el grupo de robótica de la Universidad Rey Juan Carlos<sup>2</sup>. 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 *middleware ICE*<sup>3</sup>, 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.

---

<sup>1</sup>[https://www.asus.com/3D-Sensor/Xtion\\_PRO\\_LIVE/](https://www.asus.com/3D-Sensor/Xtion_PRO_LIVE/)

<sup>2</sup><http://jderobot.org>

<sup>3</sup><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 (píxeles) 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.

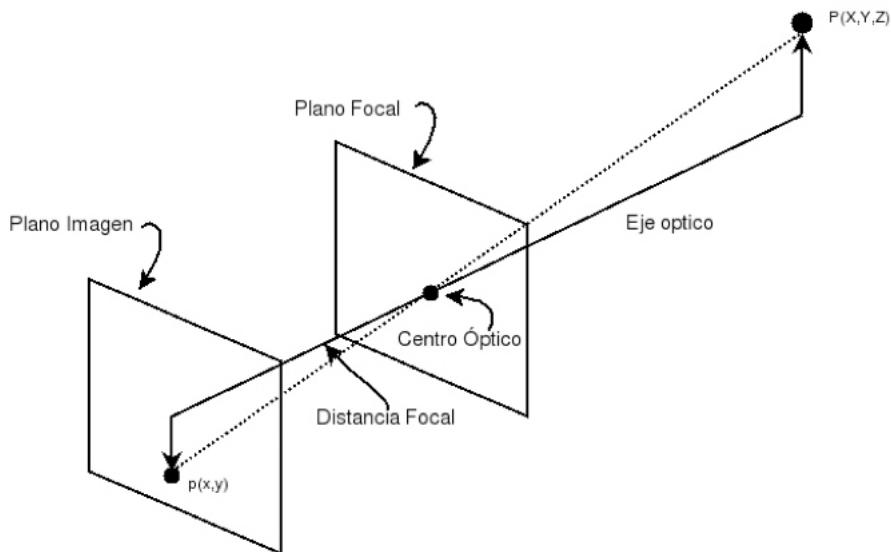


FIGURA 3.3: Modelo de cámara Pinhole

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 obtienen 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 proveniendo 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 ha servido como referencia para la realización de este trabajo. En la Figura 3.4 podemos ver una captura de pantalla con las diferentes visualizaciones.

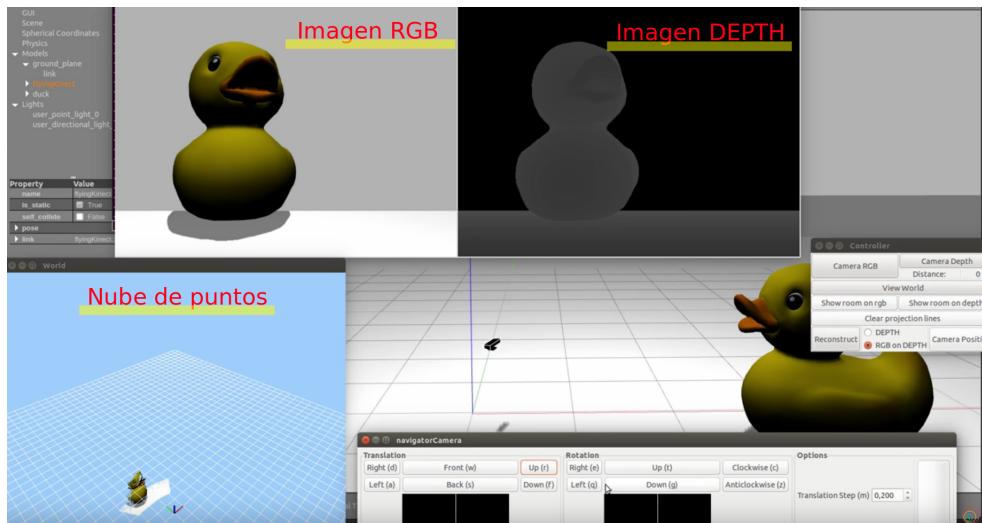


FIGURA 3.4: RGBDViewer: Captura de pantalla con las tres vistas de los diferentes datos; imagen de color, de profundidad y nube de puntos.

### 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 entorno RPC 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.<sup>4</sup>

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 financiada por un consorcio de compañías comerciales y su propia organización sin ánimo de lucro, **Open Perception**. A parte de los donantes y contribuidores individuales que aportan al proyecto.<sup>5</sup>

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 comprensión 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.<sup>6</sup>

---

<sup>4</sup><https://zeroc.com/products/ice>

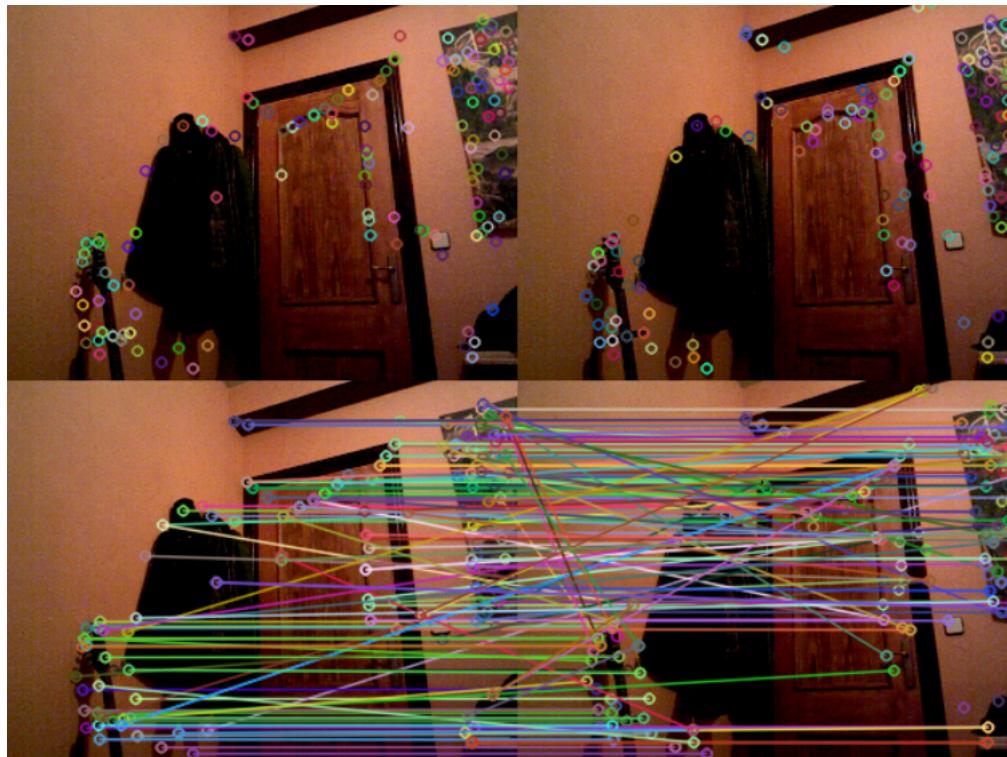
<sup>5</sup><http://pointclouds.org/>

<sup>6</sup><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 vídeos, 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 47000 usuarios en la comunidad, excediendo los 14 millones de descargas. La librería es usada ampliamente en empresas, grupos de investigación y organismos gubernamentales.

OpenCV ha 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 cómo es simple hacer una multiplicación y una división por un escalar.<sup>7</sup>

Example:	Output:
<pre>#include &lt;iostream&gt; #include &lt;Eigen/Dense&gt;  using namespace Eigen;  int main() {     Matrix2d a;     a &lt;&lt; 1, 2,         3, 4;     Vector3d v(1,2,3);     std::cout &lt;&lt; "a * 2.5 =\n" &lt;&lt; a * 2.5 &lt;&lt; std::endl;     std::cout &lt;&lt; "0.1 * v =\n" &lt;&lt; 0.1 * v &lt;&lt; std::endl;     std::cout &lt;&lt; "Doing v *= 2;" &lt;&lt; std::endl;     v *= 2;     std::cout &lt;&lt; "Now v =\n" &lt;&lt; v &lt;&lt; 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 (LGPL3+ 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 del 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 escrita en C, pero ha sido diseñada para tener soporte para un gran rango de lenguajes, tales como Perl y Python. GTK++ tiene una gran colección de *widgets* e 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 utilizada en muchos proyectos y en grandes plataformas.<sup>8</sup>

### 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

<sup>7</sup><http://eigen.tuxfamily.org/>

<sup>8</sup><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.<sup>9</sup>

### 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.<sup>10</sup>

Se ha usado la librería **Mesa 3D Graphics** en linux que es una implementación de la especificación de OpenGL con código abierto.<sup>11</sup>

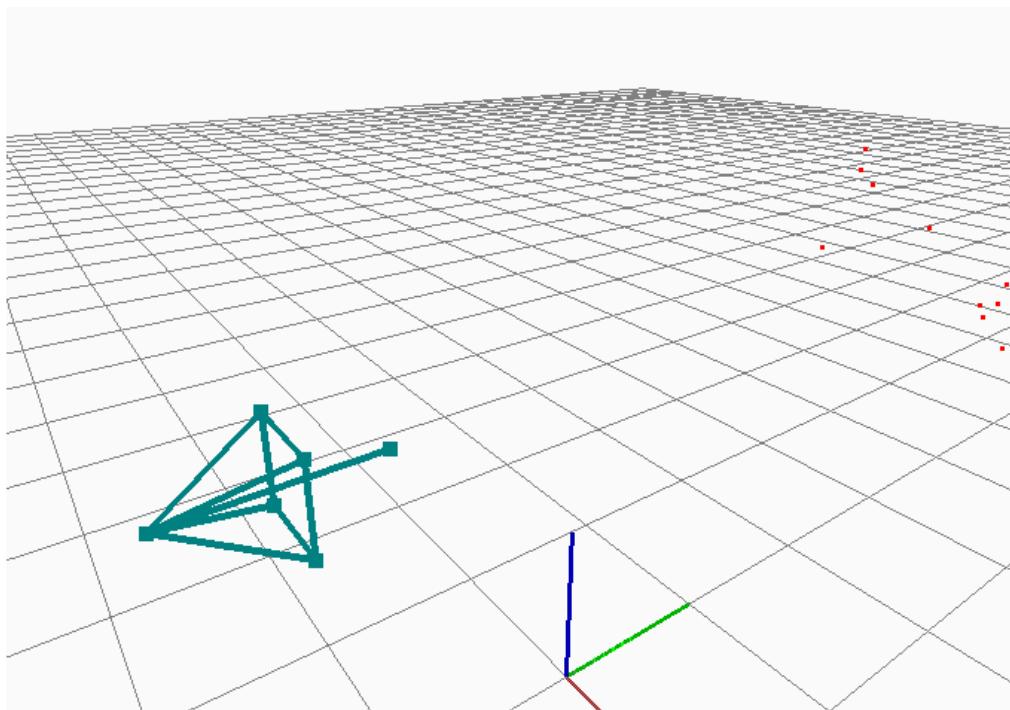
OpenGL en este trabajo se ha usado para visualizar la posición de la cámara, su estela y la colección de nubes de puntos obtenida y procesada de las imágenes RGB y de profundidad. En la Figura 3.7 se puede apreciar una captura de pantalla con la posición de la cámara dibujada en el espacio tridimensional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

---

<sup>9</sup><https://glade.gnome.org/>

<sup>10</sup><https://www.opengl.org/>

<sup>11</sup><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, en este capítulo se detalla la solución software desarrollada. Primero se presenta el diseño global y después se analiza en detalle el componente realizado con una visión profunda del desarrollo por bloques y su funcionamiento.

### 4.1. Diseño

El sistema se basa principalmente en dos componentes; uno existente en JdeRobot (**OpenniServer**) que funciona como driver del sensor y proporciona las imágenes obtenidas por éste y el componente realizado en este proyecto (**RealRTEstimator**) que se encarga, una vez recogidas las imágenes, de toda la lógica restante.

El objetivo del componente, consiste en estimar en tiempo real la posición y movimiento incremental y en 3D del sensor, por lo que deberá entregar 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 RealRTEstimator recoge las imágenes a través de ICE y 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 los algoritmos internos. A su salida entrega una matriz RT que describe la posición y orientación 3D absolutas en ese preciso instante de tiempo.

Respecto al funcionamiento interno del RealRTEstimator se puede ver a grandes rasgos el diagrama de bloques 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 (píxeles) en 2D usando la imagen de profundidad a nube de puntos en 3D.

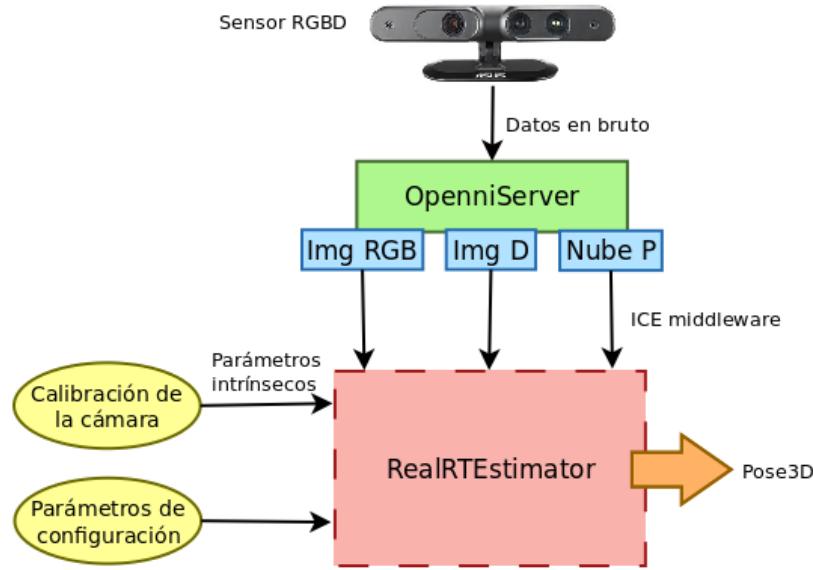


FIGURA 4.1: Esquema global de funcionamiento, entradas y salidas.

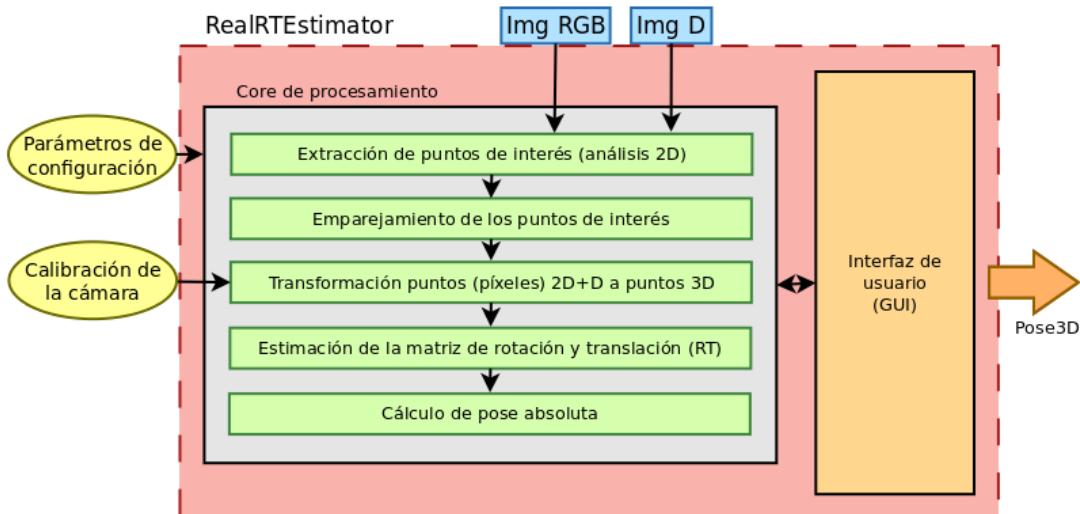


FIGURA 4.2: Diagrama interno del componente RealRTEstimator.

- 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.

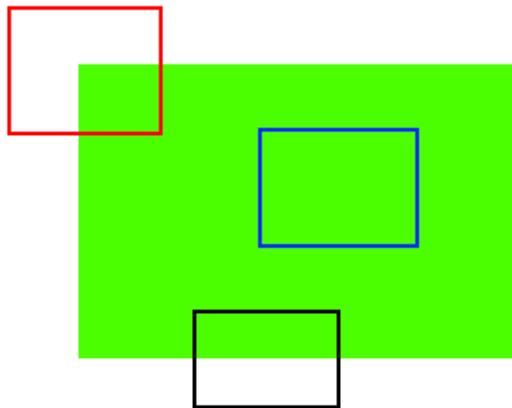
## 4.2. Análisis 2D

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 cuáles 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 se llega a la ecuación básica (4.1) que determina si una ventana contiene una esquina o no. OpenCV tiene la función `cv2.cornerHarris()` para este propósito.

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2 \quad (4.2)$$

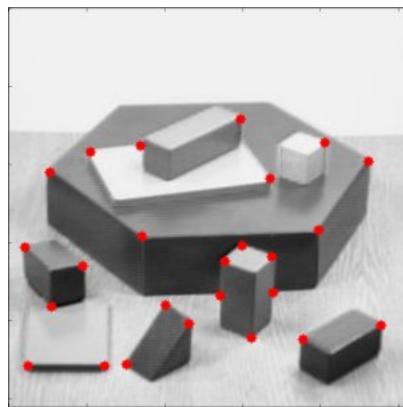
$\lambda_1$  y  $\lambda_2$  son los autovalores de la matriz  $M$ , que determinarán si una región es esquina, borde o zona plana.

- Cuando  $|R|$  es pequeño, que sucede cuando  $\lambda_1$  y  $\lambda_2$  son pequeños, la región es plana.
- Cuando  $R < 0$ , que sucede cuando  $\lambda_1 >> \lambda_2$  o viceversa, la región es un borde.
- Cuando  $R$  es grande, que sucede cuando  $\lambda_1$  y  $\lambda_2$  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  $\lambda_1$  o  $\lambda_2$  se encuentran por encima de un valor mínimo  $\lambda_{min}$ , se considera que cierta región es esquina. En la Figura 4.4 se puede observar el resultado de aplicar dicho algoritmo en una imagen. La función en OpenCV de este detector es `cv2.goodFeaturesToTrack()`.




---

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

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. Se detallarán más adelante.

### 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 la región del punto dado y se busca el mismo (o el que más se parezca) en 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 cuánto se parecen los descriptores entre sí.

#### Descriptores 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 puede no 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.

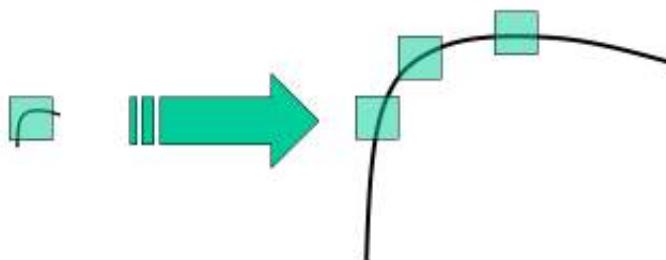


FIGURA 4.5: Ejemplo de diferentes puntos con escala

SIFT nace para resolver esta carencia de mano de D. Lowe (D.Lowe, 2004). Un algoritmo 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 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  $\sigma$  es capaz de detectar puntos de interés para diferentes escalas.  $\sigma$  actúa como un parámetro de escala. Para bajos niveles de  $\sigma$  la gaussiana devuelve altos valores para las pequeñas esquinas, sin embargo, altos valores de  $\sigma$  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  $(x, y, \sigma)$  valores, donde  $(x, y)$  representa el espacio y  $\sigma$  la escala.

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

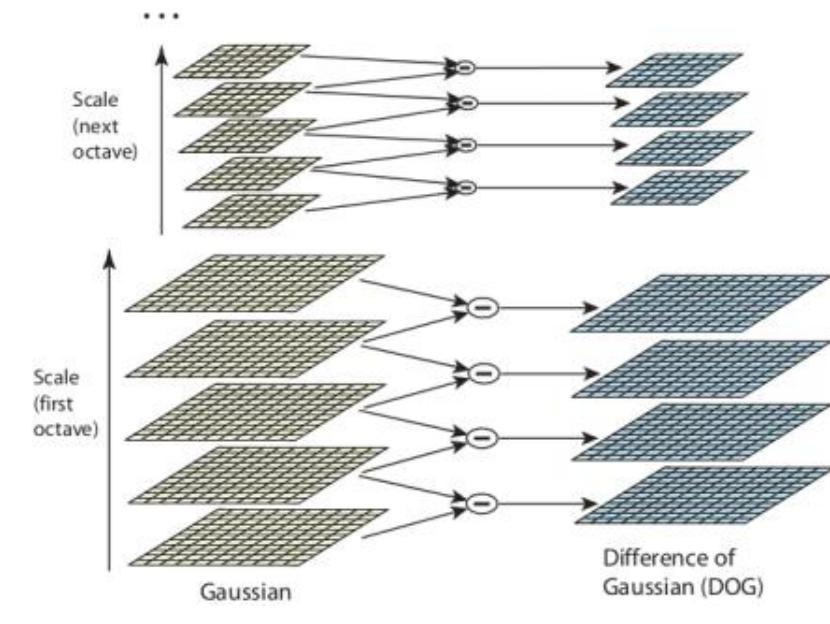


FIGURA 4.6: Proceso del cálculo de la diferencia de Gaussianas para diferentes octavas.

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

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

## 4. Descriptores

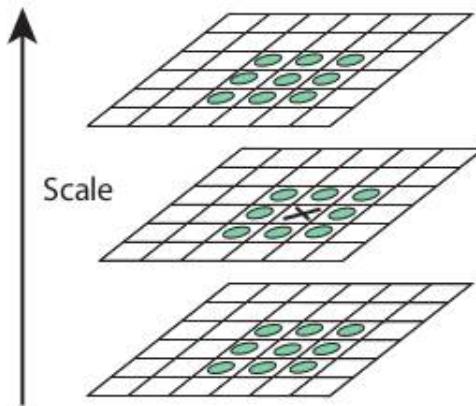


FIGURA 4.7: Ejemplo de local-extrema en SIFT.

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);

```

## Descriptores 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.

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

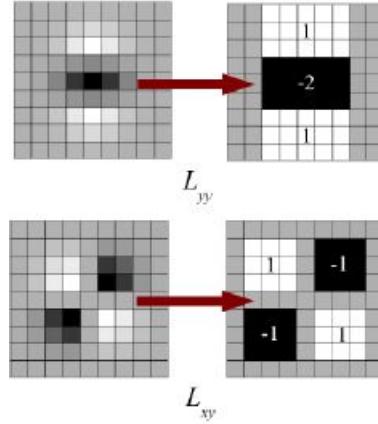


FIGURA 4.8: Proceso con filtros de cuadros, *Box filter*.

Una vez calculado el escalado, se propone el cálculo de la orientación principal 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  $60^\circ$ . 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:

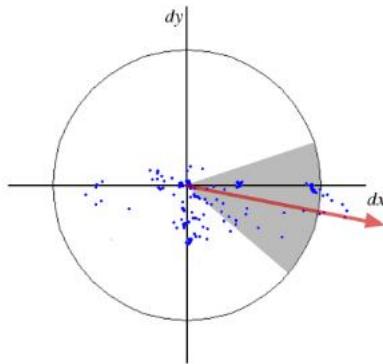


FIGURA 4.9: Cálculo de orientación con SURF.

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

Con el vector representado, para cada característica detectada el descriptor con SURF contendrá 64 dimensiones. A menor dimensión, mayor velocidad de cómputo y de

<sup>1</sup>La  $s$  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.

emparejamiento, pero se consigue menor distinción entre características. Para esto SURF tiene una versión extendida con 128 dimensiones.

En resumen, SURF añade muchas mejoras para incrementar la velocidad en cada paso. Pruebas experimentales han demostrado que puede ser tres veces más rápido que 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.

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

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

El componente RealRTEstimator 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 los peores y filtrar por mejores utilizando diferentes técnicas. Así pues, se intentará coger los mejores emparejamientos entre una imagen en ( $t$ ) y otra en ( $t + 1$ ).

Se presentan 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. Emparejamiento por 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 usa OpenCV. En primer lugar se tiene que crear un objeto del tipo *BruteForceMatcher* y pasarle 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.

En el siguiente ejemplo de código se puede observar cómo 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 ayuda a dibujar los mismos para su visualización. Coloca las dos imágenes a tratar en horizontal y dibuja 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. Emparejamiento por 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 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

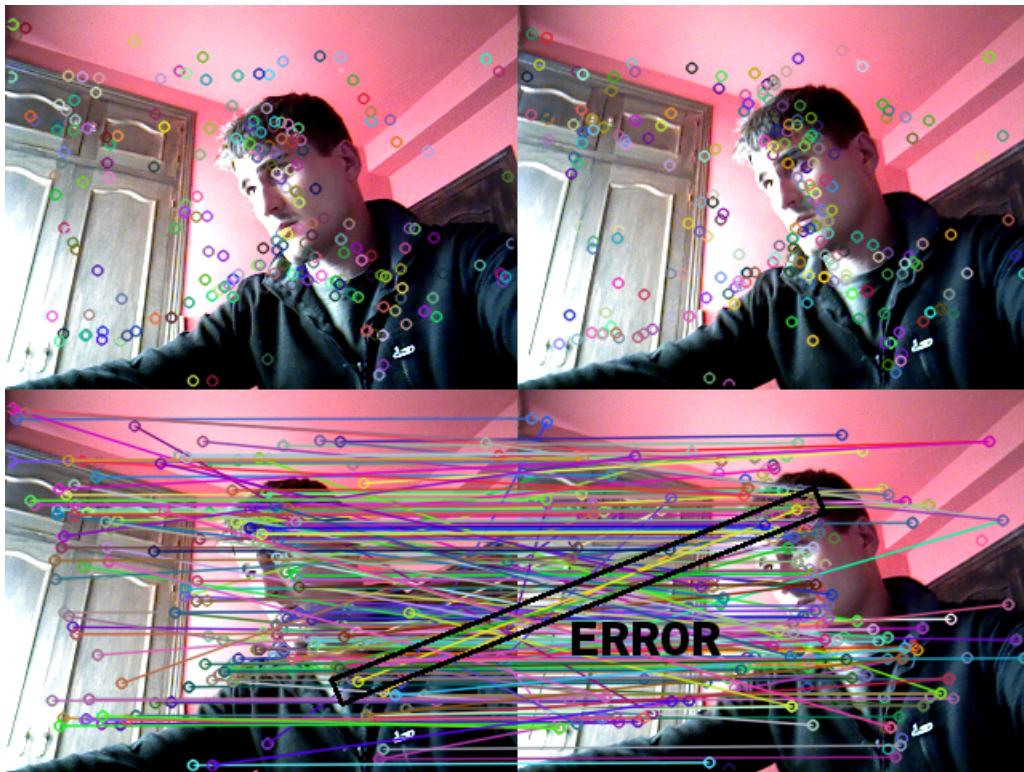


FIGURA 4.10: Detección y emparejamiento de puntos de interés con OpenCV usando SIFT para el cálculo de puntos de interés y descriptores y Fuerza Bruta para el cálculo de los emparejamientos.

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 de emparejamiento

Para el 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.

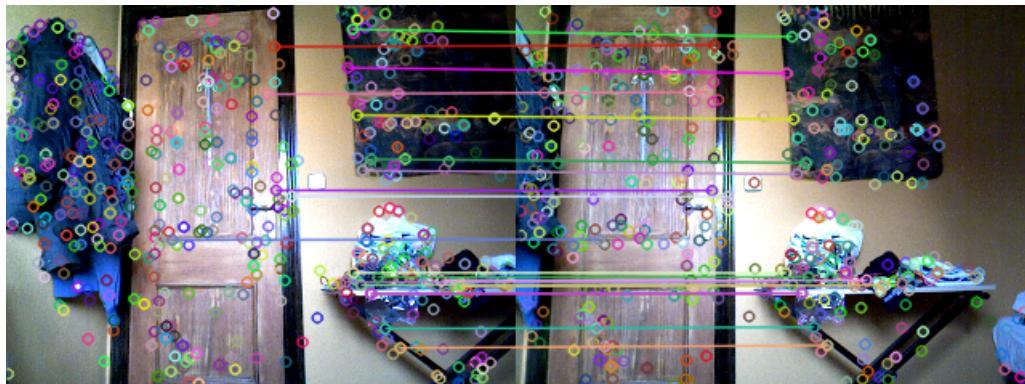


FIGURA 4.11: Emparejamiento de los mejores X puntos ordenados por distancia.

- 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 varias 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 y es origen de muchos errores en la estimación.

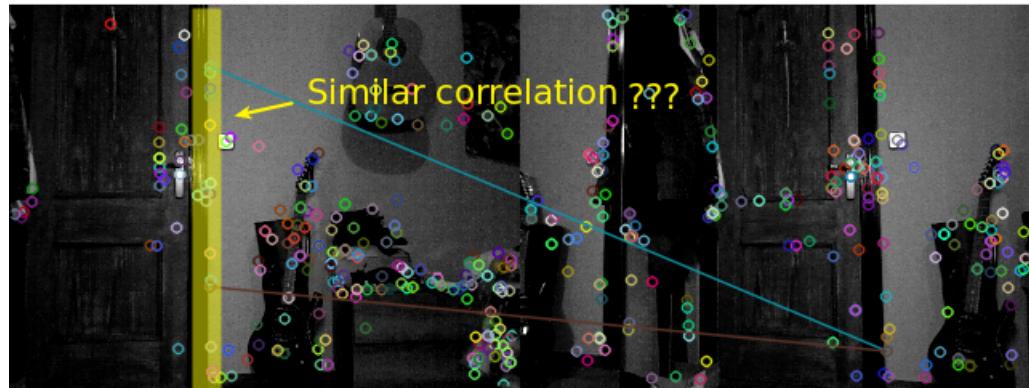


FIGURA 4.12: Detección y emparejamiento de los mejores dos puntos.

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. Pero 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 pequeña ( $< 100$ ) 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].distance;
6     if (outNumber >= OUTSTANDING_DISTANCE) {
7         // Guardamos el emparejamiento
8     }
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 posible 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 necesita llevar a 3D los puntos calculados para el instante ( $t$ )<sup>2</sup>, para 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 de puntos 3D se necesitará además la imagen DEPTH correspondiente. O lo que es lo mismo, el mismo píxel o punto de la imagen a color debe relacionarse con su homólogo en la imagen de profundidad. Se puede deducir que ambas imágenes deberán estar perfectamente sincronizadas para asegurar que ambas se corresponden con el mismo instante de tiempo. En la Figura 4.13 tenemos el diagrama de transformación de los puntos.

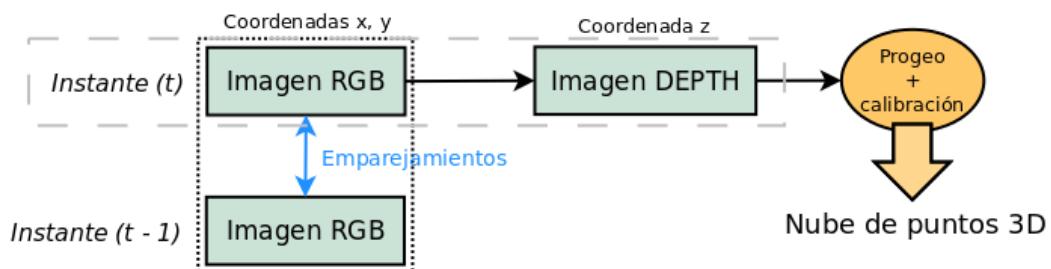


FIGURA 4.13: Diagrama de transformación a puntos 3D.

<sup>2</sup>Como 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.

Para conseguir la información 3D se ha utilizado la librería de Progeo, disponible en JdeRobot, 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, que será el que se encuentre a una distancia  $d$  de la recta de retroproyección (ej: punto  $P$  en la Figura 4.14).

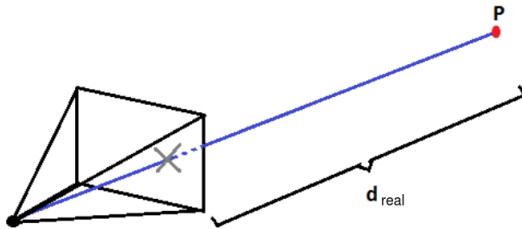


FIGURA 4.14: Cálculo de punto 3D con la información de distancia  $d$ .

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. La distancia que devuelve el sensor, sin embargo, 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  $P_r$  que corresponde al punto 3D real.

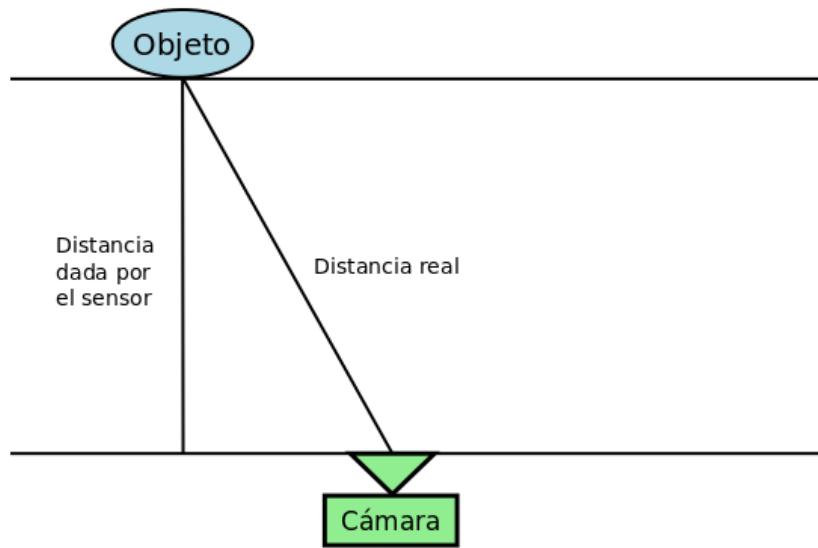


FIGURA 4.15: Distancia real y la dada por el sensor.

Para calcular el punto  $P_r$  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 y marca su orientación 3D junto con el *roll*<sup>3</sup> (parámetros extrínsecos). El proceso matemático es el siguiente:

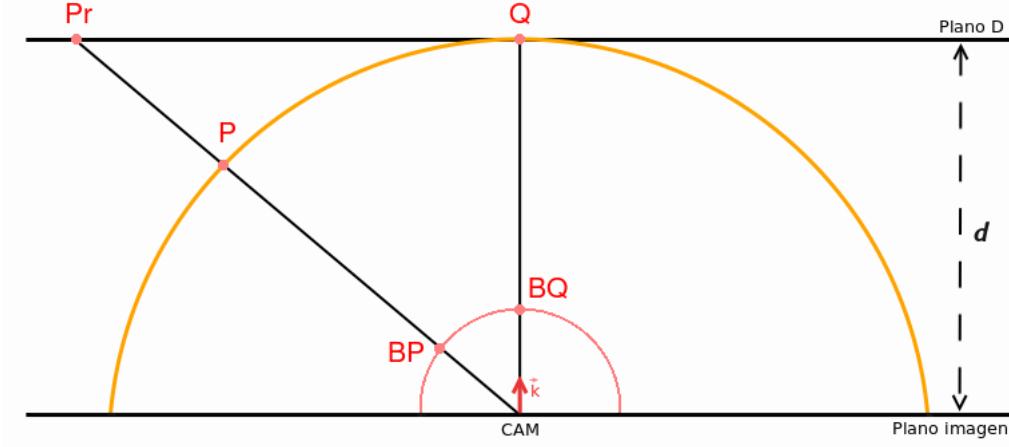


FIGURA 4.16: Ejemplo de corrección de distancia.

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

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

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:

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

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*;  $\vec{v}$ :

$$\vec{v} = \frac{\overrightarrow{CMBP}}{|CMBP|} \quad (4.7)$$

Utilizando la ecuación paramétrica de la recta:

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

<sup>3</sup>Movimiento del sensor sobre el eje central del sensor, paralelo al *foa* o foco de atención.

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  $\vec{k}$ :

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

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  $Pr$ .

En la Figura 4.17 se puede ver un ejemplo de una nube de puntos mal calculada, con la distancia dada por el sensor y bien calculada obteniendo la distancia real de los puntos 3D.

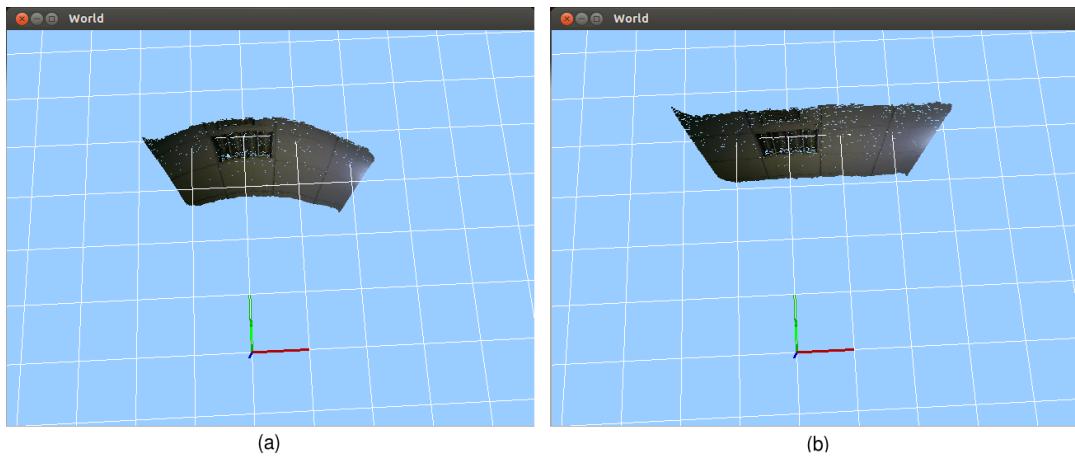


FIGURA 4.17: Ejemplo de una nube de puntos mal (a) y bien calculada (b).

## 4.5. Cálculo de movimiento tridimensional

Una vez calculados los puntos 3D, el siguiente bloque del componente realizado, RealRTEstimator, es la estimación de movimiento. En este caso hay que calcular continuamente el movimiento relativo entre los instantes  $(t - 1)$  y  $(t)$ . 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 + Traslación) 4x4 de la manera que muestra la Figura 4.18.

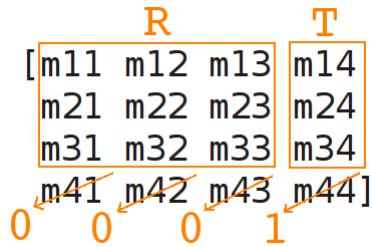


FIGURA 4.18: Matriz RT. Donde  $R$  corresponde a la rotación y  $T$  a la traslación.

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

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

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

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

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 define en total seis grados de libertad para caracterizar completamente el movimiento tridimensional.

#### 4.5.2. Cálculo RT mediante SVD

Al calcular los puntos 3D a través de Progeo desde el sensor se tienen 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**.

Como sistema de referencia absoluto se toma el de la primera posición de la cámara. Así las coordenadas absolutas iniciales vienen dadas por la matriz unidad. A partir de ahí se empieza a calcular con respecto a ese sistema de referencia absoluto la posición de los nuevos puntos en cada instante y de la cámara.

Los mismos puntos se encontrarán en la misma posición en coordenadas absolutas en diferentes instantes de tiempo. Como se puede apreciar en la Figura 4.19 el desplazamiento del sensor en coordenadas absolutas muestra:

- Puntos absolutos correspondientes únicamente al instante  $(t - 1)$ .
- Puntos absolutos correspondientes únicamente al instante  $(t)$ .
- Cámaras en absolutas correspondientes tanto al instante  $(t - 1)$  como  $(t)$ .

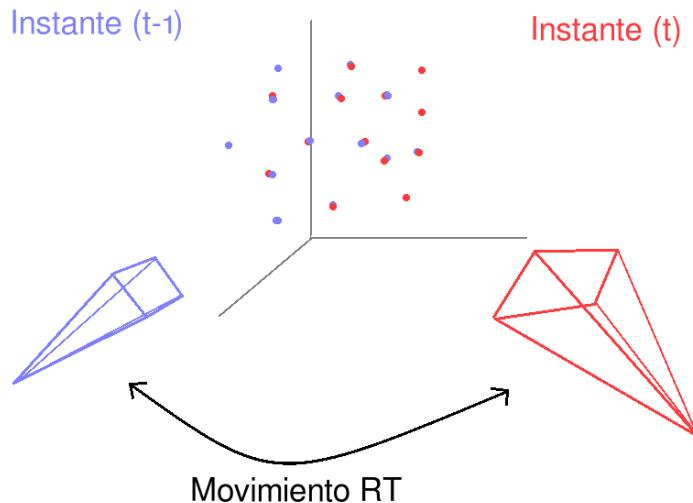


FIGURA 4.19: Cálculo visual de movimiento en coordenadas absolutas.

Esos puntos comunes son los utilizados para el cálculo de movimiento de la cámara y son los provenientes del emparejamiento entre píxeles de las imágenes.

Como entrada en este bloque de cálculo RT 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 ( $t - 1$ ) y la nube de puntos en coordenadas relativas (cam) en el instante ( $t$ ); se calcula la matriz RT de la cámara con respecto al mundo que mejor hace encajar la nube relativa en el instante ( $t$ ) con respecto a la absoluta en el instante ( $t - 1$ ), tal y como expresa la siguiente ecuación:

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

Una vez calculada la matriz RT de la cámara con respecto al mundo, para expresar un punto  $P$  que viene dado en el sistema de referencia de la cámara (relativas) a sus coordenadas absolutas solo hay que multiplicar por la matriz obtenida, que contiene la RT de la cámara en el mundo absoluto, tal y como expresa la siguiente ecuación:

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

En este caso para calcular la nube de puntos del instante actual y con coordenadas absolutas solo se tiene que multiplicar por la inversa de la matriz calculada.

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

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

donde la nube de puntos absolutas, recién calculada, en ( $t$ ), pasa a ser la nube de puntos absolutas en ( $t - 1$ ).

#### 4.5.3. Optimización mediante RANSAC

El cálculo de movimiento ha sido uno de los puntos en los que se ha encontrado más problemas ya que el error es acumulativo y una iteración con un error demasiado grande produce que en los siguientes instantes el cálculo de la posición absoluta de la cámara sea erróneo y arrastre indefinidamente ese error. 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:

- Desechar el cálculo con demasiado error espacial y de reproyección.
- Implementación de RANSAC para eliminar fallos espurios.

El método de RANSAC (*Random Sample and Consensus*) nos ayuda con el ruido y los datos atípicos. Clasifica los datos en dos subconjuntos: uno de ellos son los datos que satisfacen un patrón o un modelo que consideramos bueno y otro subconjunto que corresponde con los datos malos o atípicos.

Esta discriminación permitirá en gran medida conseguir una solución que se adapte al conjunto de datos que nos interesa conseguir.

Los distintos parámetros de los que consta el algoritmo que se ha implementado son:

- Parámetros a elección del usuario, que en código están definidos como:

1. *RANSAC\_PERC*: Es el porcentaje de emparejamientos que se esperan que sean buenos.
  2. *RANSAC\_ITER*: Corresponde al número de iteraciones que ejecuta el algoritmo.
- Parámetros propios del algoritmo:
    1. *N*: Número total de emparejamientos 3D.
    2. *good\_sub*: Subconjunto de datos buenos.
    3. *bad\_sub*: Subconjunto de datos malos o atípicos.

Para lograr esto, el algoritmo de RANSAC implementado hace lo siguiente:

1. Del número total de emparejamientos (*N*) se selecciona un porcentaje aleatorio (*RANSAC\_PERC*). Este parámetro definido por el usuario determina el porcentaje de puntos en los que se supone que siempre va a haber un subconjunto bueno de datos. Cuanto más porcentaje más puntos para hacer la estimación de movimiento 3D, pero más posibilidad también de tener valores erróneos. Por lo tanto, habrá que escoger del número de emparejamientos, así como su porcentaje de error para cada iteración.
2. Con los *N* emparejamientos seleccionados se realiza la estimación de la matriz de movimiento 3D mediante SVD.
3. A partir de la matriz RT obtenida del paso anterior se coge la nube de puntos proveniente de la cámara y le aplicamos la RT para transformarlas a coordenadas absolutas. Para cada punto 3D transformado se calcula la distancia o el error que existe entre ese punto y el mismo en el instante anterior y se van sumando. El resultado de la suma de error o distancia total se mantiene guardado.
4. Se repiten los pasos 1, 2 y 3 tantas veces como esté definido en *RANSAC\_ITER*. Este parámetro cuanto mayor sea, mayor probabilidades de encontrar un subconjunto mejor. Por contra, el número de iteraciones compromete el tiempo de cómputo.

Para cada iteración completa, se comprueba la distancia total obtenida en el paso 3. Si es menor se guarda, si no, se desecha.

5. De todas las iteraciones se selecciona la que ha tenido una menor distancia o error y se guarda la matriz RT utilizada.
6. Se emplea la matriz RT del paso 4 para calcular la nube de puntos final.

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

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 calcular la distancia para una misma imagen.

Este algoritmo es muy robusto frente a valores espurios, ya que si se realiza el cálculo con alguno de estos valores o puntos ruidosos se desechará el resultado cogiendo como matriz RT la de otra iteración de RANSAC.

## 4.6. Interfaz gráfica

El componente desarrollado dispone de una interfaz gráfica en donde se pueden ver gráficamente los pasos realizados individualmente, así como el resultado final de toda la lógica del sistema. Ha sido desarrollada con *glade* y la apariencia se puede observar en la Figura 4.20.

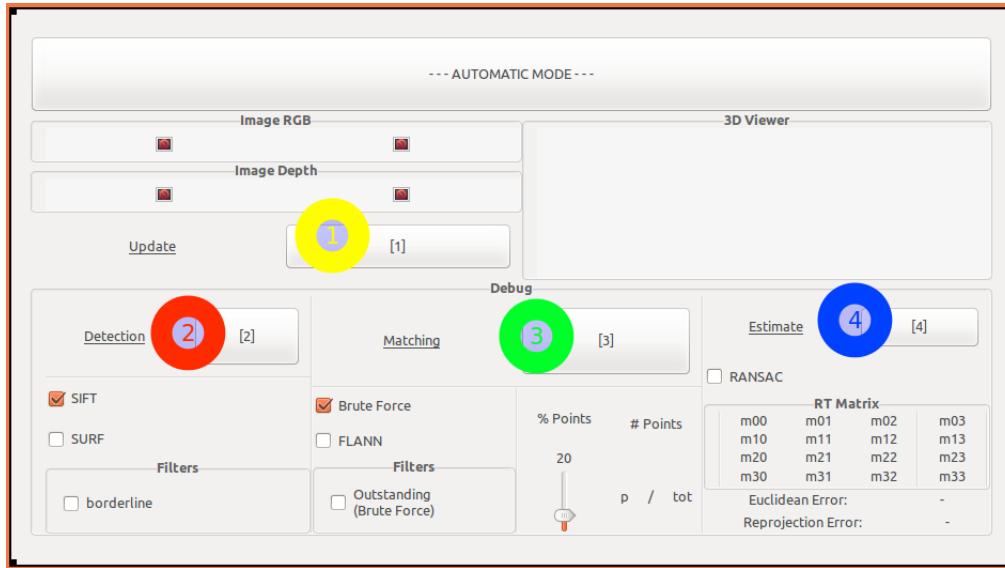


FIGURA 4.20: Captura del mapa de funcionalidades de la herramienta gráfica.

La interfaz gráfica, basada en GTK, permite la realización del procesado paso a paso, (muy útil para depurar) o de manera automática (funcionamiento continuo). La ventana del componente tiene cuatro zonas diferenciadas:

1. Actualización de imagen. (Figura 4.21)
2. Detección de puntos. (Figura 4.22)
3. Cálculo de emparejamientos. (Figura 4.23)
4. Estimación de la matriz. (Figura 4.24)

La herramienta permite, además, la modificación de algunos parámetros de configuración sobre la marcha, como pueden ser la elección algoritmos o filtros para el bloque de detección o el bloque de emparejamiento, 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.

En las siguientes Figuras (4.21, 4.22, 4.23, 4.24) se observan las capturas de cada uno de los pasos en la estimación de movimiento 3D. El botón [1] actualiza la imagen. A la izquierda aparecerá la imagen en el instante ( $t$ ) y a la derecha la imagen del instante ( $t - 1$ ). El botón [2] detecta los puntos de interés con las opciones seleccionadas y actualiza las imágenes del GUI con los puntos de interés extraídos. El botón [3] muestra los emparejamientos obtenidos sobre las imágenes y con un *slider* se puede

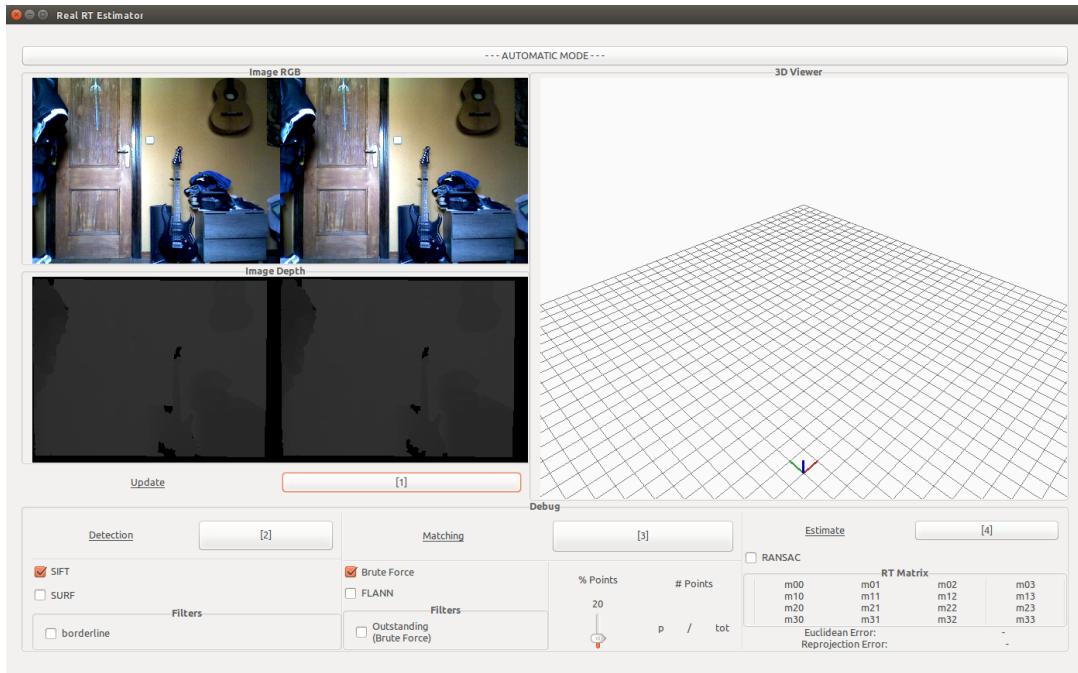


FIGURA 4.21: Paso 1: Actualización de imagen.

elegir el porcentaje de puntos emparejados. Por último, el botón [4] estima la matriz RT, donde se puede observar en vivo la matriz RT estimada y el resultado de la estimación en el visor 3D con la posición de la cámara y la nube de puntos.

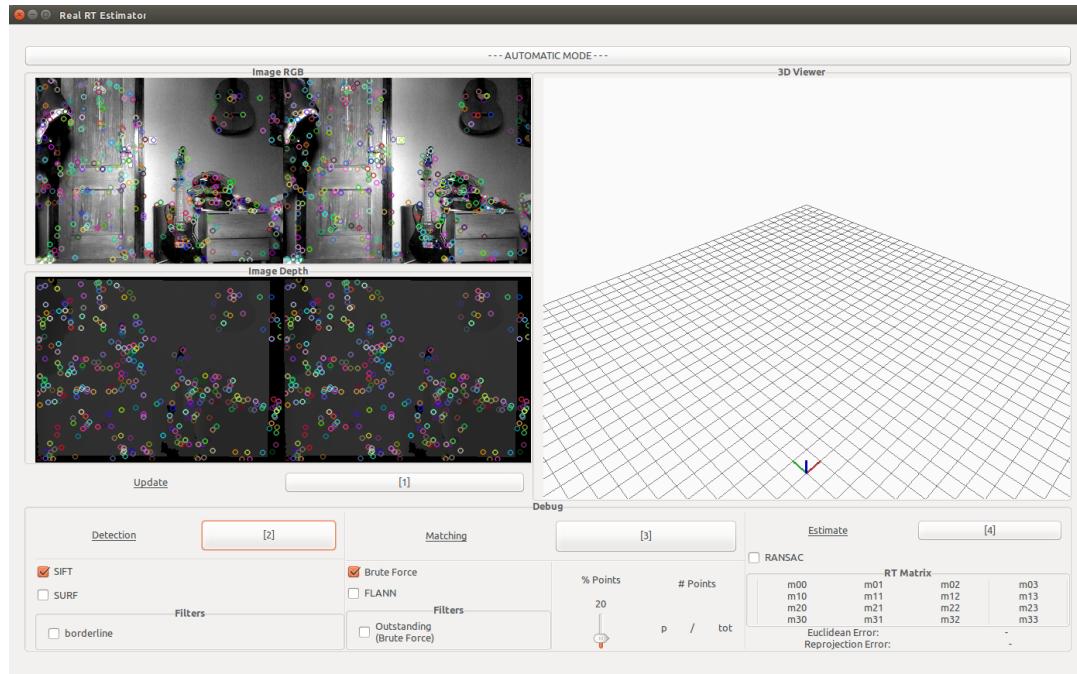


FIGURA 4.22: Paso 2: Detección de puntos.

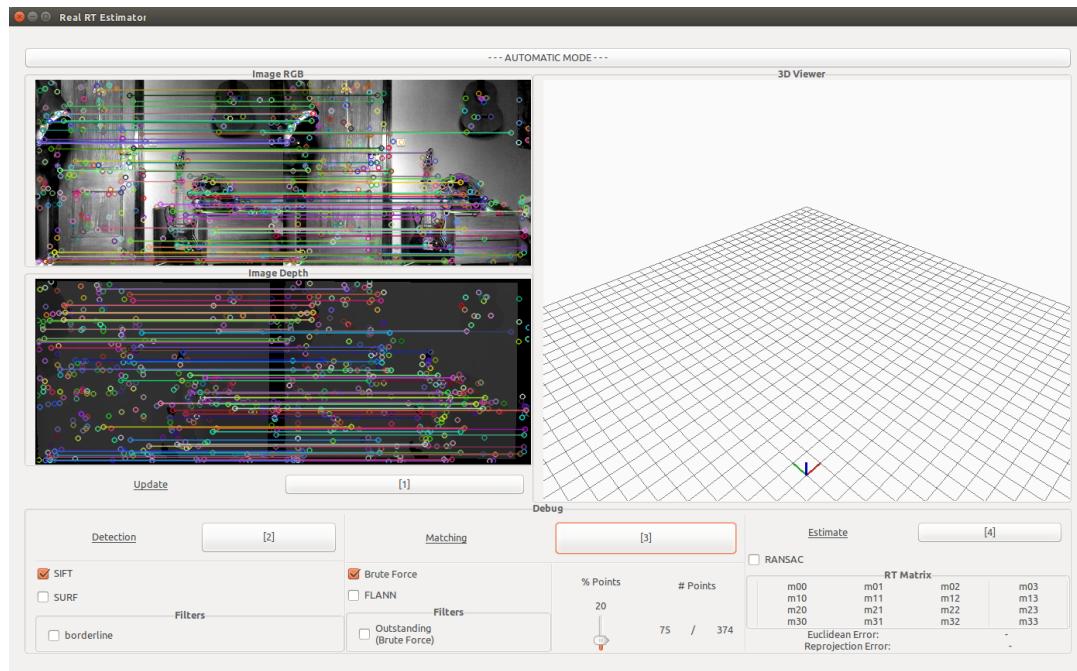


FIGURA 4.23: Paso 3: Cálculo de emparejamientos.

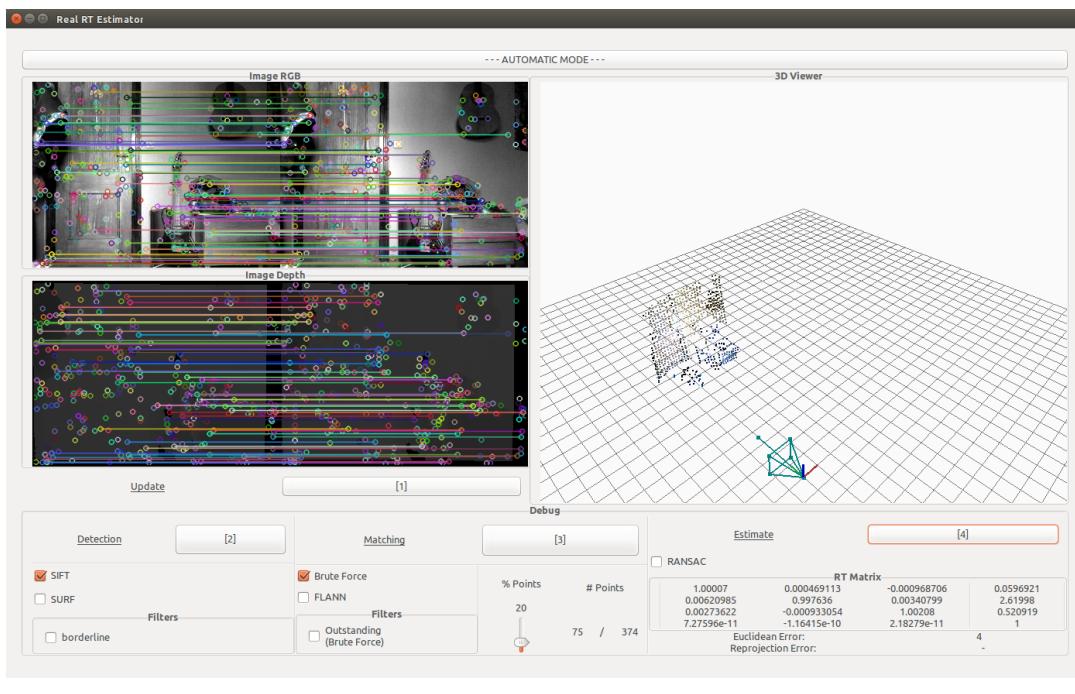


FIGURA 4.24: Paso 4: Estimación de la matriz RT.

## Capítulo 5

# Experimentos

En este capítulo se detallarán las pruebas realizadas, con el objetivo de validar la solución propuesta en este trabajo. Además, se comprobarán las diferentes configuraciones desarrolladas y se evaluarán sus prestaciones con los costes temporales. Estos experimentos han permitido depurar el componente para ajustar y mejorar la funcionalidad del mismo durante el desarrollo.

### 5.1. Validación experimental

El entorno de pruebas utilizado ha sido siempre un entorno real, trabajando con los datos en vivo proporcionados por OpenNIserver que extrae la información directamente del sensor RGBD.

Se propone en esta sección validar el componente en las diferentes fases unitarias que lo componen: detección de puntos de interés, cálculo de emparejamientos y estimación de movimiento.

#### 5.1.1. Detección de puntos de interés

La detección de puntos de interés resulta una tarea sencilla si nos ayudamos de OpenCV. En esta sección veremos algunos ejemplos de las diferencias entre técnicas empleadas, y el filtro frontera.

En la Figura 5.1 se puede ver la extracción de puntos de interés tanto con SIFT como con SURF, aparentemente no hay grandes diferencias. Como se vió en el capítulo 4 el cálculo de puntos de interés es muy similar y la diferencia fundamental radica en el tiempo de cómputo.

#### Efecto del filtro de puntos frontera

En la Figura 5.2 tenemos el resultado de aplicar el filtro frontera sobre una misma imagen. Como se puede observar, al añadir el filtro frontera se eliminan algunos de los puntos más controvertidos de la imagen, como son los puntos a lo largo del marco de la puerta o del mástil de la guitarra.

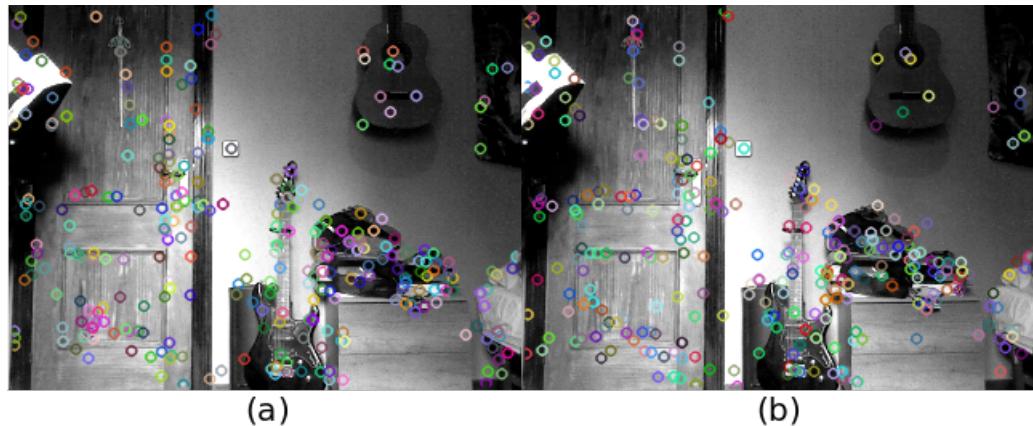


FIGURA 5.1: Extracción de características con SURF (a), y con SIFT (b).

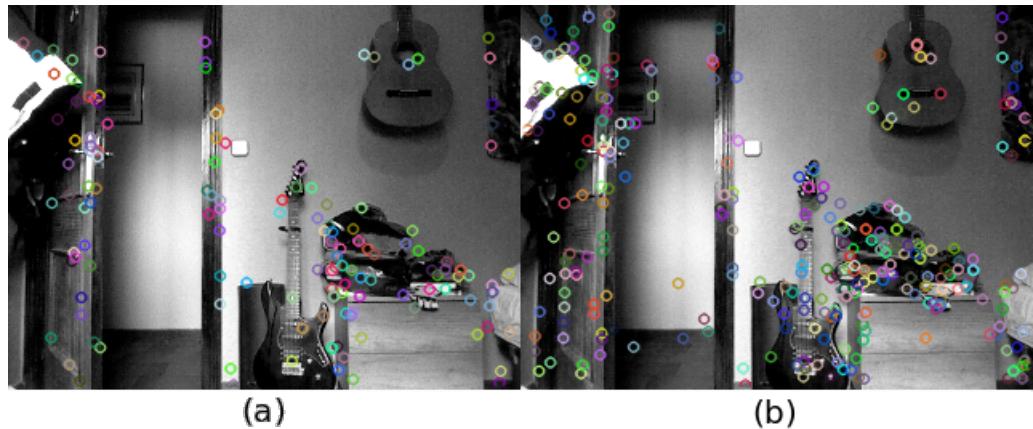


FIGURA 5.2: Captura de la extracción de puntos de interés con filtro frontera (a), y sin filtro (b).

### 5.1.2. Cálculo de emparejamientos

Para probar el cálculo de los emparejamientos se somete el componente a cambios de traslación y rotación para comprobar la veracidad de los puntos emparejados. Se parte, como se explicó en el anterior capítulo 4, de los 20 % mejores emparejamientos medidos por distancia, ya que en condiciones normales se tienen unos 300 y el 20 %(60) es un número razonable de puntos y se asegura consistencia.

#### Necesidad del filtro de sobresalencia

En la Figura 5.3 se puede observar cómo en condiciones normales se ha producido un error en el emparejamiento de dos puntos de interés. Como vimos en el capítulo 4, es debido a que los descriptores de esos puntos son los más similares y muy parecidos a todos los que, en este caso, forman el marco de la puerta.



FIGURA 5.3: Captura de un emparejamiento erróneo en condiciones normales.

Este caso se ha buscado a propósito para justificar la necesidad del **filtro de sobresaliente**. En todas las pruebas de ahora en adelante se ha incluido para añadir robustez a los experimentos y evitar estos casos de malos emparejamientos que estropean si no el cálculo tridimensional.

### Funcionamiento en Traslación y Rotación

En la Figura 5.4 se puede observar el resultado de aplicar los emparejamientos bajo circunstancias de traslación y rotación. Como se puede ver en los emparejamientos obtenidos, el resultado es el esperado y no hay error.

#### 5.1.3. Resolución 3D

La resolución 3D o estimación de movimiento compone el último paso que completa una iteración del sistema, por lo que en esta etapa podremos ver el resultado final del movimiento entre varios fotogramas consecutivos.

Las pruebas mostradas a continuación son realizadas con un funcionamiento normal del componente y en un tiempo determinado. El objetivo de estos experimentos es comprobar el comportamiento del sistema para los diferentes tipos de traslaciones ( $x, y, z$ ) y rotaciones ( $pitch, yaw, roll$ ).

Para estas pruebas y las siguientes se hará uso del componente con el comportamiento automático activado, repitiendo así de manera automática cada ciclo de estimación 3D. También se han incorporado en ellas, para añadir robustez, el filtro de sobresalencia y RANSAC.

### Validación de la traslación

En esta subsección se describen pruebas para comprobar cómo se comporta el sistema ante la traslación. Se movió la cámara a lo largo de los tres ejes ( $x, y$  y  $z$ ) (Figura 5.5).

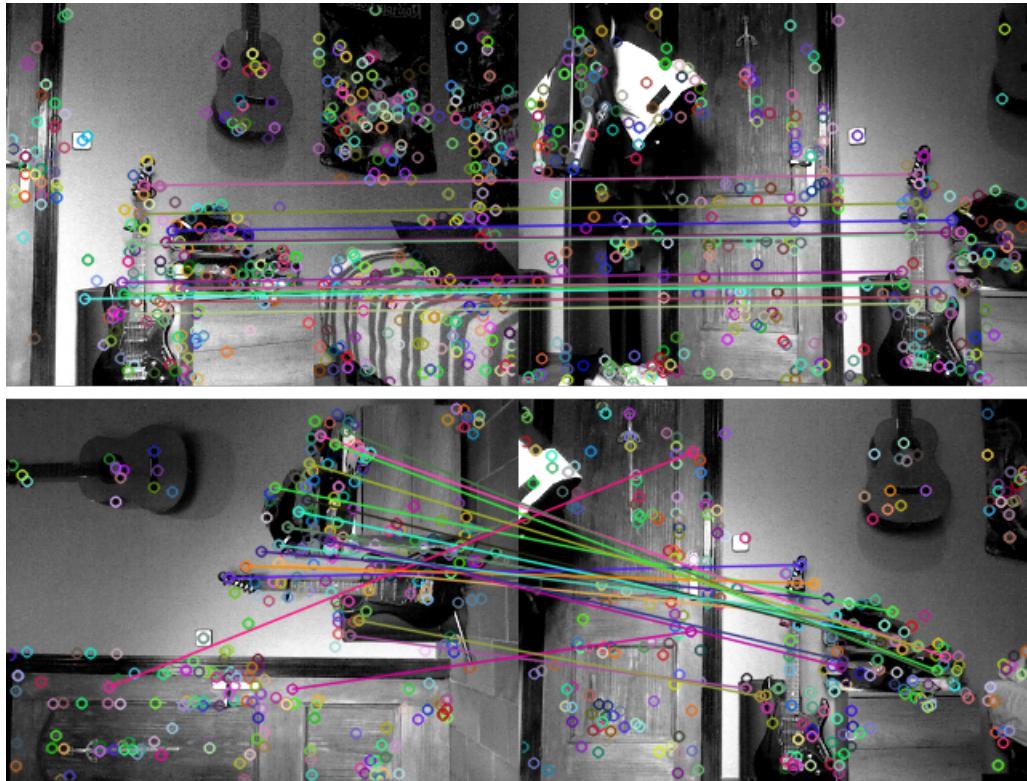


FIGURA 5.4: Captura de los emparejamientos con imágenes sometidas a una traslación y a una rotación, respectivamente.

Para estas pruebas se utiliza el mismo entorno de trabajo por lo que el punto de partida es el que se muestra en la Figura 5.6.

En la Figura 5.7 se muestra la estimación de movimiento sobre el eje  $y$ . El desplazamiento se ha intentado hacer lo más recto posible y el recorrido se muestra como estela en el visor 3D del componente. En ambas capturas se puede ver la ida y la vuelta recorriendo por cada una 1.5 metros aproximadamente. Después se ha vuelto a la posición de origen del movimiento.

Se puede observar, después de aproximadamente 100 iteraciones, cómo ha afectado el error a la estimación al volver el sensor al origen.

En la Figura 5.8 se puede ver el efecto de la traslación en los ejes  $x$  y  $z$  respectivamente. Se ve cómo el error en estos ejes es menor, percibiendo oscilaciones en su mayor parte únicamente en el eje  $y$ . La validación del eje  $x$  se ha realizado sobre una mesa recorriendo 1 metro, por lo que no se obtiene desplazamiento en  $z$ . En cuanto a las pruebas del eje  $z$  se aprecia un balanceo en  $x$  debido al movimiento vertical de la mano ya que no ha seguido una trayectoria recta.

Los resultados son aceptables en la estimación 3D. No se consigue un cierre de trayectoria perfecto pero se acerca bastante a ello. Pensando que se ha aplicado una traslación de más de 1 metro el error final obtenido se puede medir en centímetros, acentuándose más para el eje  $y$  correspondiente al perpendicular al plano imagen y el que se obtiene de la imagen de profundidad.

Como vimos en el capítulo 3 la información de profundidad del sensor no es perfecta y tiene un pequeño margen de error. Aparte de tener un rango de distancias

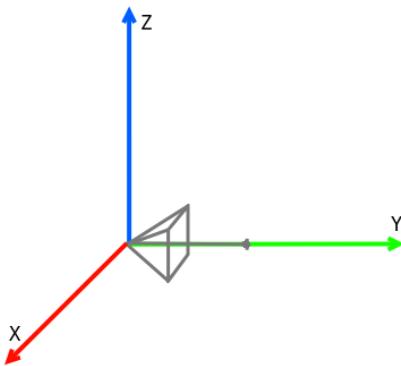


FIGURA 5.5: Ejes de coordenadas  $x$ ,  $y$  y  $z$ , con la orientación de la cámara.

óptimo establecido por el fabricante ( $0.8$  to  $3.5$  m), fuera de este rango la imagen de profundidad empieza a empeorar drásticamente y por consiguiente la estimación de movimiento se deteriora.

### Validación de la Rotación

Se valida ahora la rotación de sensor de los ángulos *pitch*, *yaw* y *roll*. En la Figura 5.9 se tiene un ejemplo gráfico de cuáles son estos movimientos y a qué rotación de ejes pertenecen.

En la Figura 5.10 se efectúa una prueba con el ángulo *pitch* y se comprueba cómo el sistema detecta una rotación total de  $90^\circ$ . Se visualiza el estado de la cámara al orientar hacia arriba el sensor y posteriormente orientarlo paralelo al suelo.

En cuanto a la translación, se observa, sin contar con el ruido, que el eje de la cámara se mantiene fijo en el origen, como era de esperar.

Para los ángulos *yaw* y *roll* se pueden observar en la Figura 5.11 las pruebas realizadas. En la primera captura se observa una rotación de  $45^\circ$  sobre el eje *yaw* y en la segunda sobre el eje *roll*. En ambos casos el movimiento estimado es el correcto.

### Validación de trayectorias combinadas

En esta subsección se muestra el funcionamiento del componente desarrollado ante trayectorias combinadas. El primer experimento consiste en la realización de un círculo a mano con el sensor RGBD. El resultado del experimento, en diferentes etapas, se puede observar en la Figura 5.12. Se aprecia cómo a pesar del movimiento irregular del brazo y el error acumulativo, el resultado obtenido se parece a un círculo.

El siguiente experimento se compone de un desplazamiento vertical hacia el eje  $z$  y después una rotación del ángulo *roll*. El resultado se visualiza en la Figura 5.13.

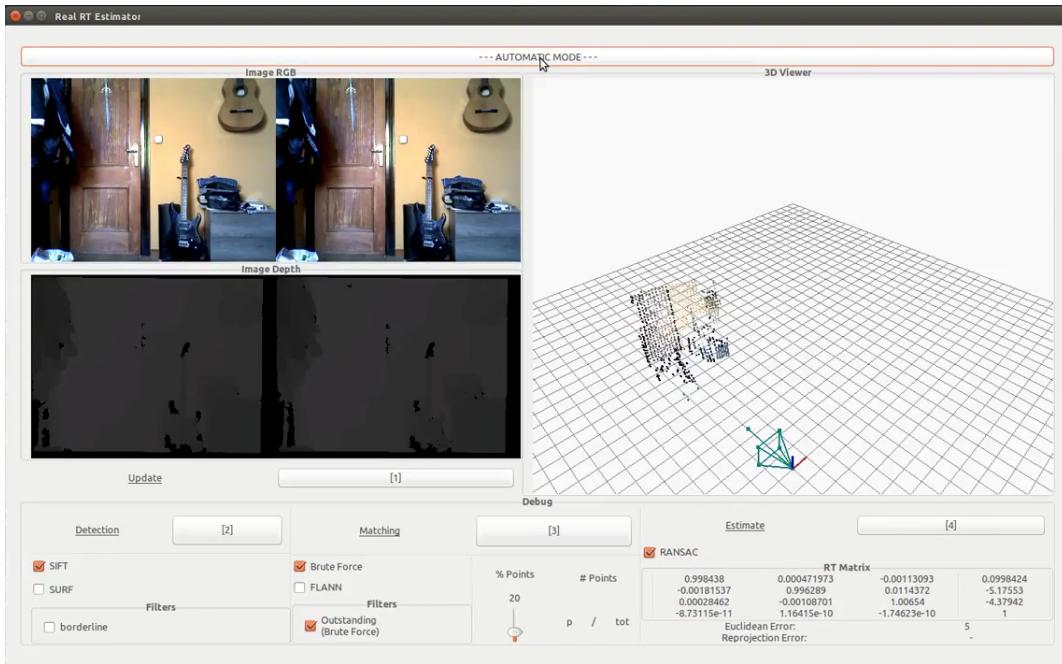


FIGURA 5.6: Punto de partida para las pruebas realizadas.

#### 5.1.4. Acumulación de error en estático

El error estático muestra el ruido que se esconde detrás de la resolución 3D en tiempo real. Como se va a poder observar en esta sección, el ruido acumulado en estático es recurrente, está presente en todas las configuraciones del algoritmo y refleja su naturaleza incremental, acumulativa. En el capítulo de conclusiones (6), se sugerirán métodos para poder eliminarlo en mayor o menor medida.

Se han realizado varios experimentos con la cámara quieta, con distintas configuraciones del algoritmo. En cada iteración se incurre en algún ligero error que se va acumulando y hace que pasados 30 o 60 segundos la posición estimada de la cámara sea distinta de la posición real, que no se ha movido nada.

En la Figura (5.14) se puede ver un ejemplo de este error para diferentes configuraciones. Como se puede apreciar, es un error que está presente en todos los casos. Las pruebas realizadas han sido capturadas a los 30 y 60 segundos respectivamente y la configuración aplicada se encuentra en la descripción de la Figura 5.14.

Si analizamos las matrices RT obtenidas se puede observar que el eje en que más crece el ruido es el eje  $y$ , el perpendicular al plano imagen, lo que es coherente con las pruebas de la sección anterior. Es precisamente en ese eje donde el algoritmo realizado se comporta peor, donde incurre en más error.

La conclusión a la que se llega con las pruebas realizadas es que el error perjudica más al sistema cuanto mayor es la duración de la estimación, ya que se trata de un error acumulativo.

## 5.2. Tiempos de cómputo del procesamiento

En esta sección se ven los tiempos de cómputo del componente realizado en este proyecto. Como el componente realiza distintas tareas hasta completar una iteración, se ha desarrollado una funcionalidad que mide el tiempo medio por cada una de ellas cuando se termina la ejecución del componente.

Se ha analizado el tiempo medio para diferentes configuraciones en un intervalo de 10 segundos. Para un uso por defecto de la aplicación, se tienen estos costes computacionales:

```

1 Average times :
2 - Update images: 0.678057 ms
3 - Points detection: 134.506 ms
4 - Points matching: 14.0233 ms
5 - Estimate matrix: 2.5231 ms
6 -
7 - TOTAL: 151.731 ms

```

Como se puede observar, la detección de puntos es la parte que más tiempo requiere. Por el contrario, la estimación de la matriz RT es la que menos. Esta prueba se ha realizado con una media de unos 300 puntos de interés por iteración. Como es lógico, el tiempo de detección variará con respecto al número de puntos de interés o características extraídas. Estos puntos de interés según las pruebas no solo están condicionados a la imagen que se está analizando, la intensidad de luz también desempeña un papel muy importante en la detección de más o menos características.

En el cuadro 5.1 podemos ver la variabilidad de tiempo de detección para diferentes números de puntos.

CUADRO 5.1: Tiempos medios de detección en relación a los puntos de interés

Número de puntos	Tiempo (ms)
100	102.27
200	117.136
250	126.019
300	134.506
350	137.805
400	144.408

La diferencia entre los distintos algoritmos y filtros no se nota en exceso en cuanto a tiempos, sin embargo, al utilizar SURF con respecto a SIFT se ha notado una reducción hasta de 10ms.

Por último, para el cálculo de la matriz de movimiento se realiza la descomposición en valores singulares (SVD). En este caso, el coste computacional depende directamente del número de puntos. La descomposición en valores singulares en relación a los emparejamientos se encuentra en el cuadro 5.2.

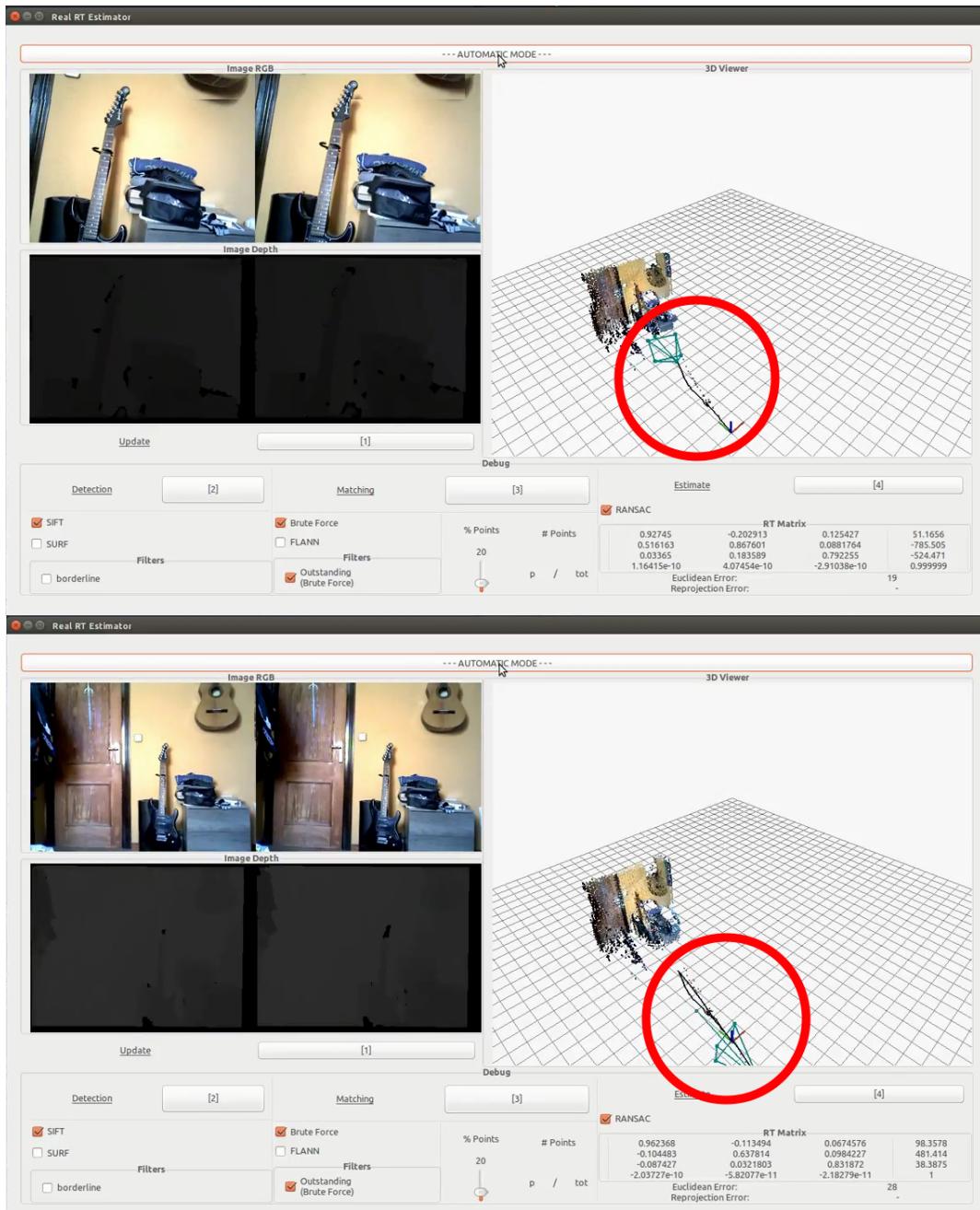
Como es de esperar, el tiempo de cómputo de la optimización RANSAC no dependerá tanto del número de emparejamientos, sino que dependerá del número de iteraciones del algoritmo. En el cuadro 5.3 se puede ver el tiempo por número de iteraciones del algoritmo para un total de unos 100 emparejamientos por iteración.

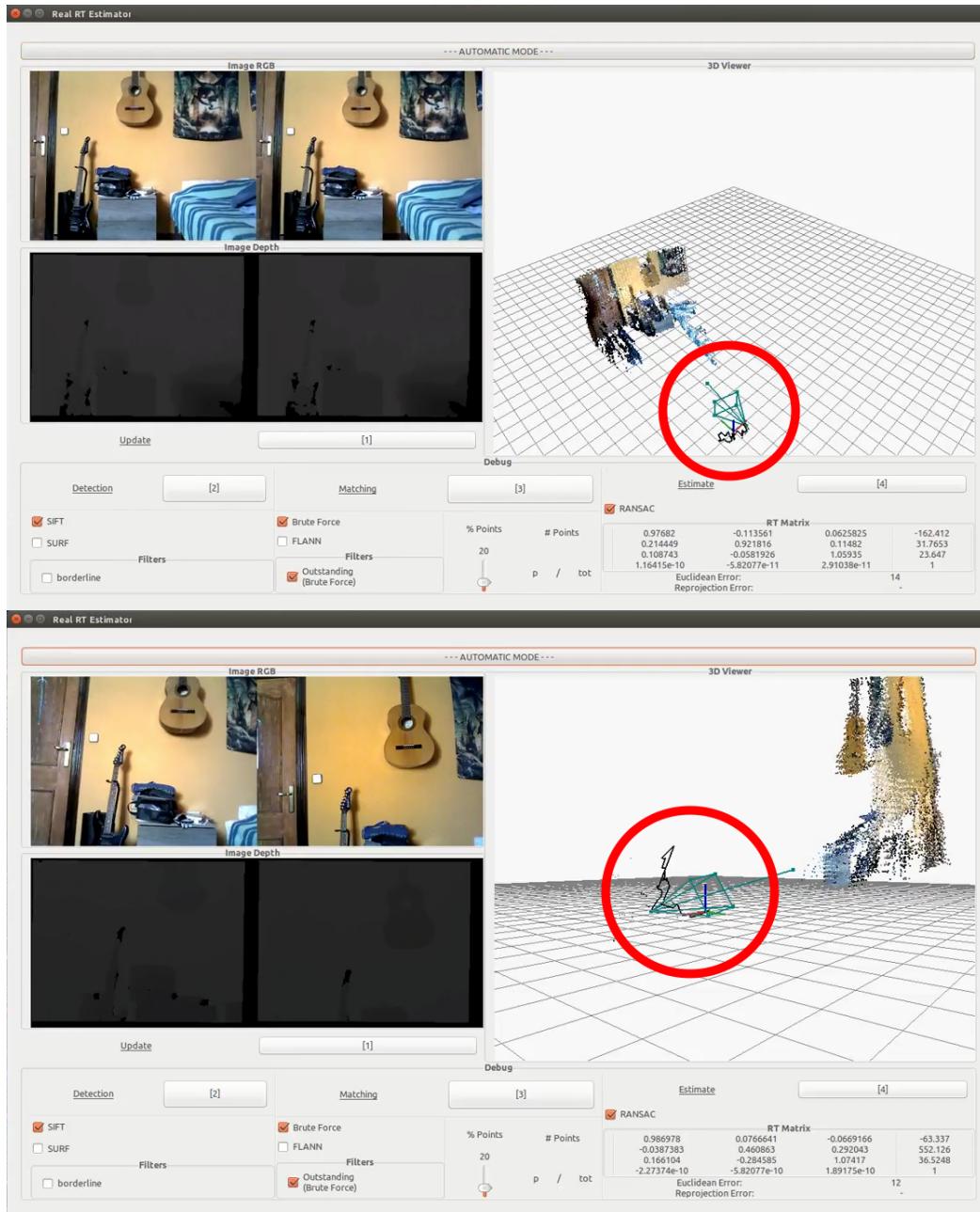
CUADRO 5.2: Tiempos medios de cálculo de SVD en relación a los emparejamientos

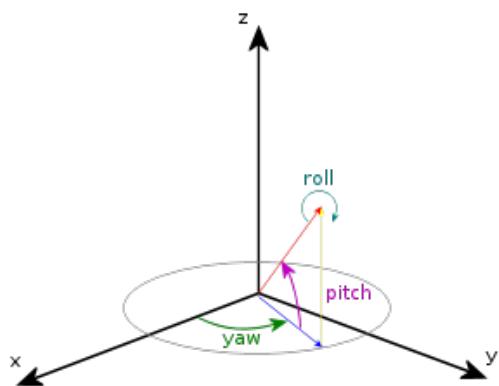
Número de emparejamientos	Tiempo (ms)
50	1.44
100	1.64
200	1.96
400	2.17

CUADRO 5.3: Tiempos medios de cálculo con RANSAC en relación a las iteraciones para una media de 100 emparejamientos por iteración

Número de iteraciones	Tiempo (ms)
10	7.11
20	16.45
50	40.01

FIGURA 5.7: Prueba con desplazamiento en el eje *y*.

FIGURA 5.8: Prueba con desplazamiento en los ejes  $x$  y  $z$ .



---

FIGURA 5.9: Dibujo con los movimientos *pitch*, *yaw* y *roll*.

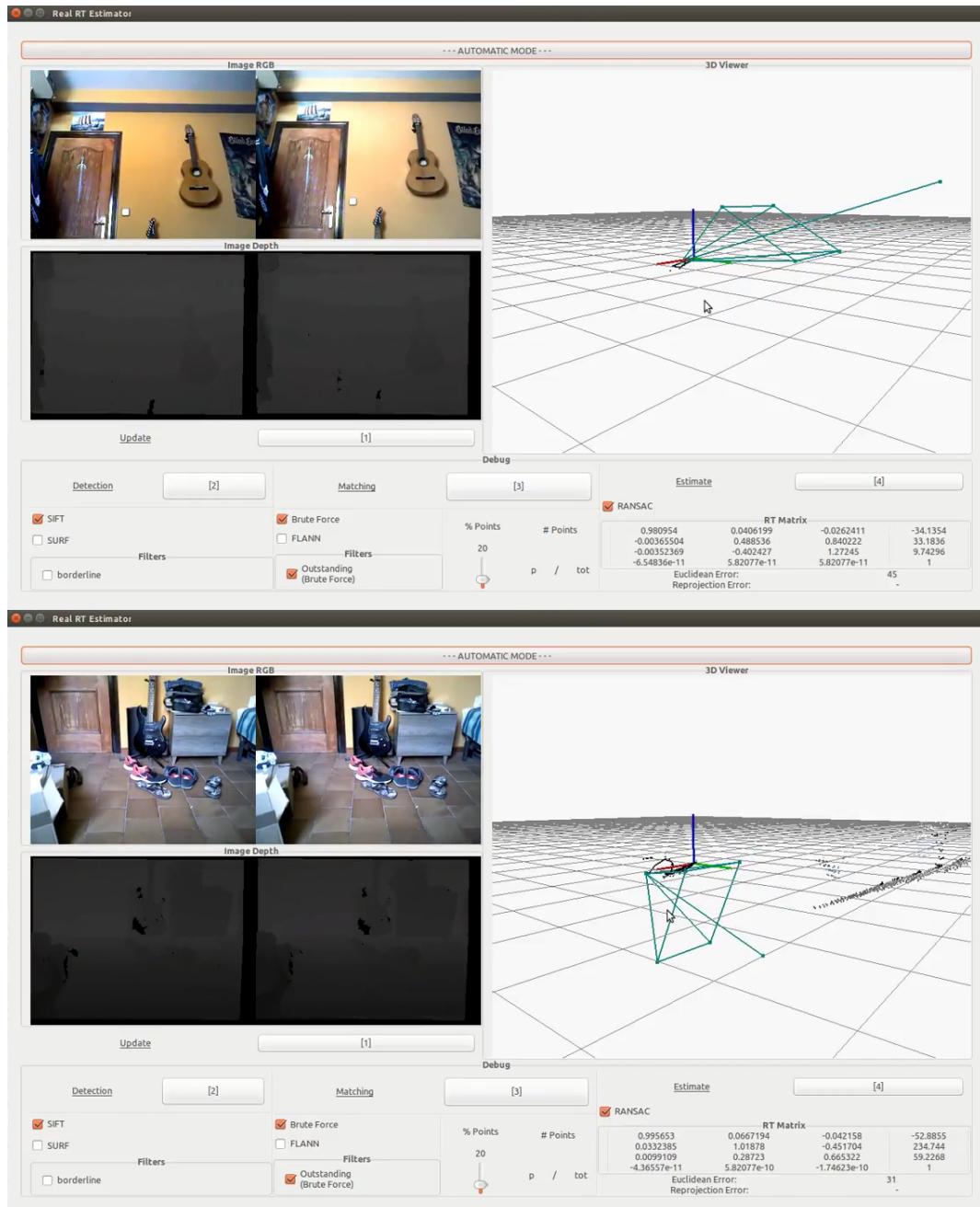
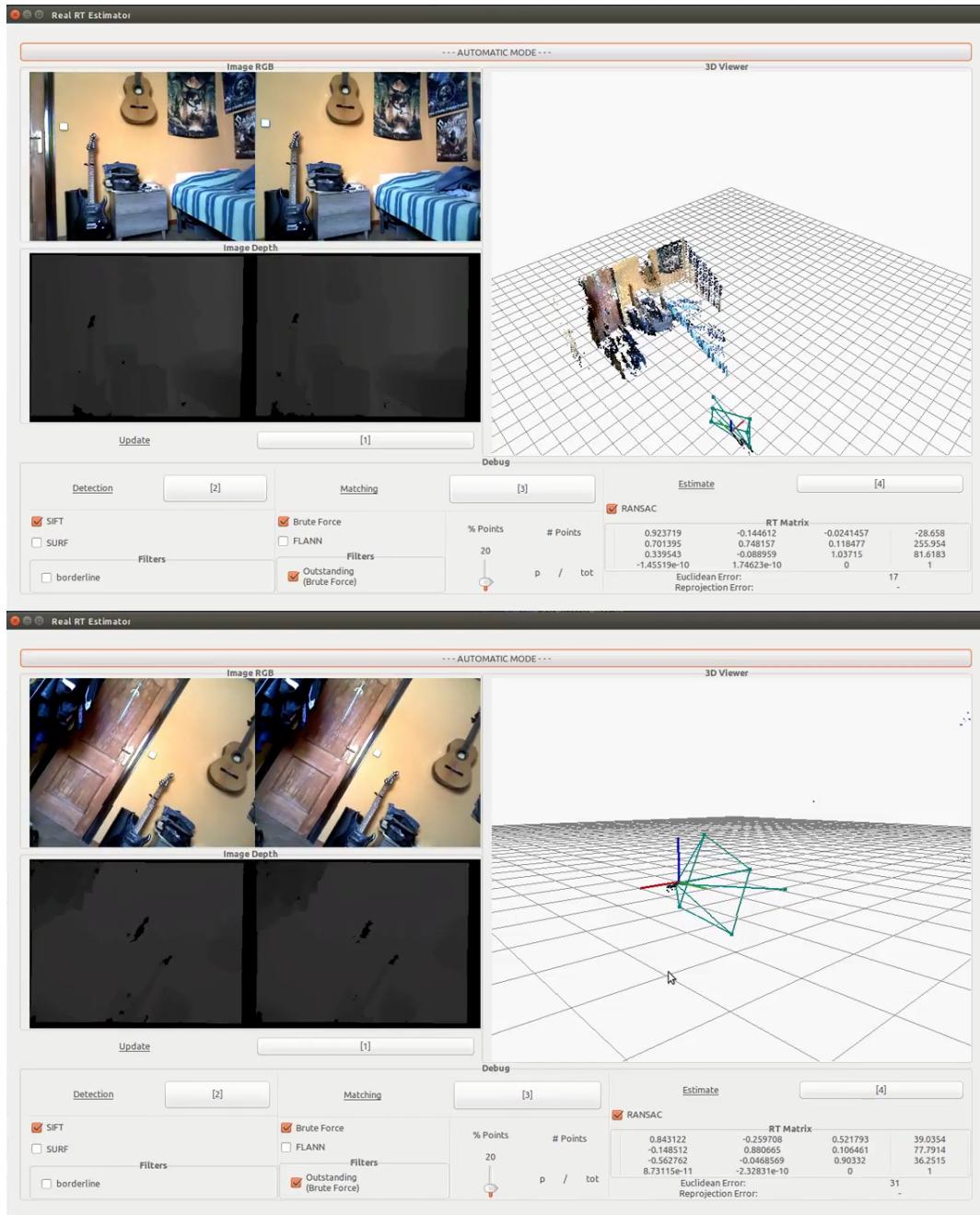


FIGURA 5.10: Prueba con desplazamiento del ángulo *pitch*.

FIGURA 5.11: Pruebas con desplazamiento en los angulos *yaw* y *roll*.

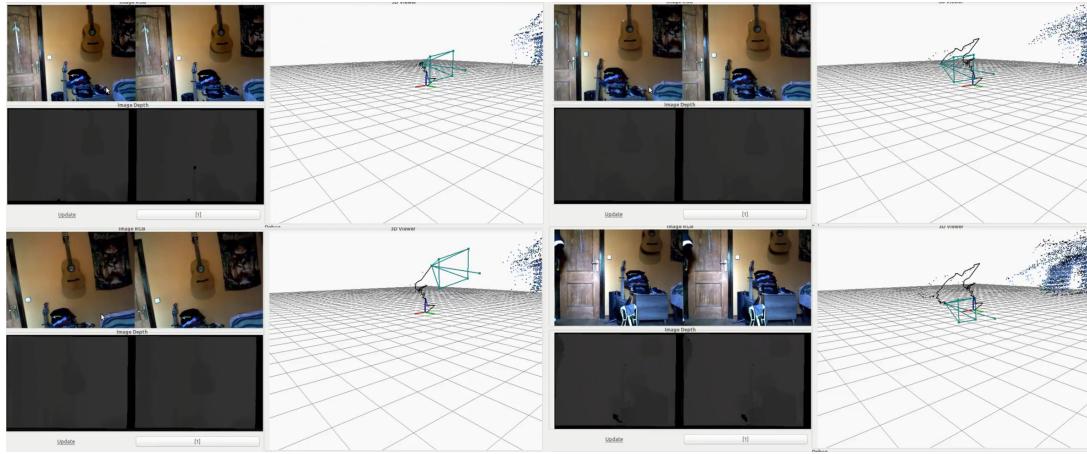


FIGURA 5.12: Experimento con el cálculo de una trayectoria en forma de círculo.

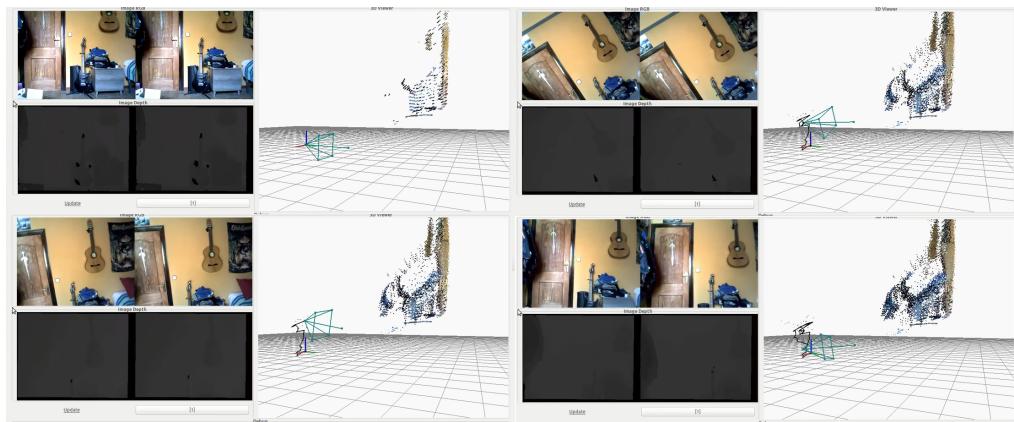


FIGURA 5.13: Experimento con desplazamiento vertical hacia el eje  $z$  y después una rotación del ángulo  $roll$ .

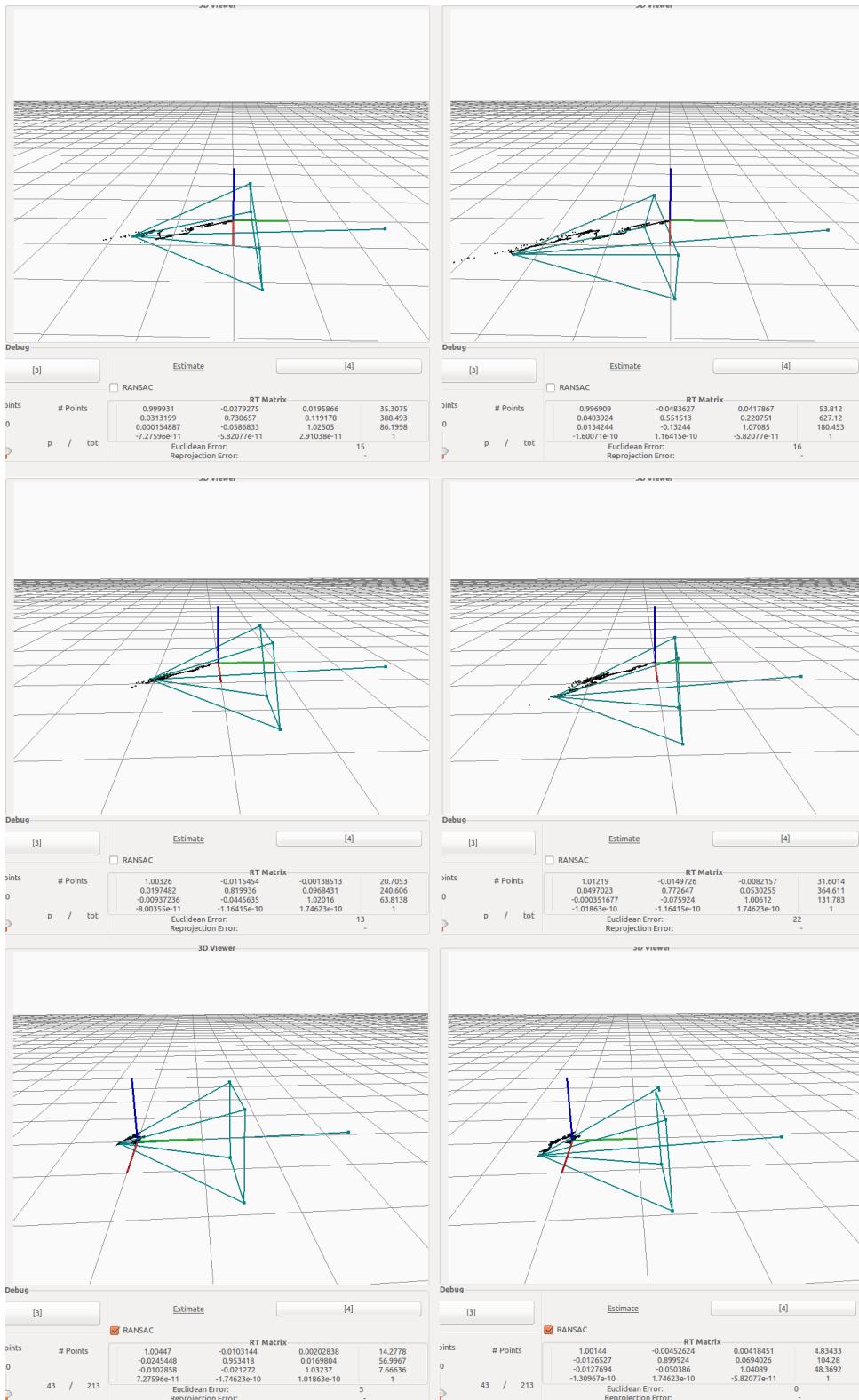


FIGURA 5.14: En la primera imagen se representa el ruido estático después de 30 y 60 segundos, con SURF y FLANN. La segunda con SIFT, Fuerza Bruta y filtros. La tercera con SIFT, Fuerza Bruta y RANSAC.



## Capítulo 6

# Conclusiones

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

En este capítulo se presentarán las principales 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 5000 líneas de código, escritas en C++, se puede decir que se ha conseguido alcanzar el objetivo global planteado en el capítulo 2; un sistema capaz de averiguar la posición y orientación 3D 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. Detección de puntos de interés.

En este primer subobjetivo se recogen las imágenes RGB y DEPTH del sensor 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.

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 (Depth) asociada a la imagen de color (RGB).

2. Emparejamiento de puntos.

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

3. Estimación de movimiento.

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

#### 4. Pruebas y experimentos.

Los experimentos mostrados en el capítulo 5 validan experimentalmente el correcto comportamiento del componente. Se puede apreciar también la implementación de la interfaz gráfica, con la visualización de cada uno de los subobjetivos, a fin de analizar correctamente los resultados obtenidos. Especialmente útil ha sido el visualizador 3D con la posición de la cámara, la estela de movimiento y la nube de puntos.

## 6.2. Trabajos futuros

Para finalizar, se proponen algunas posibles líneas futuras de interés para la ampliación de este trabajo y 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 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.

También algún mecanismo de cierre de bucle podría ser interesante para reajustar los cálculos y reducir en mayor medida ese error acumulativo. Por ejemplo, cuando se ha pasado por una zona ya conocida, poder comparar el resultado de la estimación 3D las dos veces.

### ■ Mejorar tiempo de cómputo.

Aunque el trabajo ha sido implementado buscando reducir el tiempo de cómputo y encontrar un equilibrio entre velocidad y robustez, existe 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 este trabajo, sin embargo, esta variable condiciona negativamente los resultados obtenidos. Por ello, y porque el mundo que nos rodea no es estático, podría ser interesante investigar esta línea. Los espejos o reflejos podrían entrar también en este apartado como fuente de pruebas.

# Bibliografía

- Andrew J. Davison Ian D. Reid, Nicholas D. Molton y Olivier Stasse (2007). «MonoSLAM: Real-Time Single Camera SLAM». En: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 29(6), págs. 1052-1067.
- Bay H., Tuytelaars T. y Van Gool L. (2008). «SURF: Speeded Up Robust Features». En: *Computer Vision and Image Understanding, Volumen 110, Issue 3*, págs. 346-359.
- Boehm, B. (1986). «A spiral model of software development and enhancement». En: *ACM SIGSOFT Software Engineering Notes* 11(4), págs. 14-24.
- D.Lowe (2004). «Distinctive Image Features from Scale-Invariant Keypoints». En: *University of British Columbia*.
- Durrant-Whyte, H. y T. Bailey (2006). «Simultaneous localization and mapping: part I». En: *IEEE Robotics & Automation Magazine* 13, issue 2, págs. 99-110.
- Fraundorfer, Friedrich y Davide Scaramuzza (2012). «Visual odometry: Part ii: Matching, robustness, optimization, and applications». En: *Robotics & Automation Magazine, IEEE* 19(2), págs. 78-90.
- García, Eduardo Perdices (2010). «Autolocalización visual en la RoboCup con algoritmos basados en muestras». Tesis de mtría. Universidad Rey Juan Carlos.
- Harris, Chris y Mike Stephens (1988). «A Combined Corner and Edge Detector». En: *Alvey vision conference*. URL: <http://courses.daiict.ac.in/>.
- Klein, G. y D. Murray (2007). «Parallel Tracking and Mapping for Small AR Workspaces». En: *6th IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR)*, págs. 225-234.
- Lana, Ignacio San Román (2015). «Odometría visual 3D para autolocalización de una cámara móvil en tiempo real». Tesis de mtría. Universidad Rey Juan Carlos.
- Muja, Marius y David G. Lowe. (2009). «Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration». En: *VISAPP 2009 - Proceedings of the Fourth International Conference on Computer Vision Theory and Applications, Lisboa, Portugal, February 5-8, 2009 - Volume 1*.
- Noah Snavely, Steven M. Seitz y Richard Szeliski (2006). «Photo tourism: Exploring photo collections in 3d». En: *SIGGRAPH Conference Proceedings*, págs. 835-846.
- Organista, Daniel Martín (2014). «Odometría visual con sensores RGBD». Tesis de mtría. Universidad Rey Juan Carlos.
- Pinilla, Alberto Lopéz-Cerón (2015). «Autolocalización visual robusta basada en marcadores». Tesis de mtría. Universidad Rey Juan Carlos.
- Ramos, Luis Miguel López (2010). «Autolocalización en tiempo real mediante seguimiento visual monocular». Tesis de mtría. Universidad Rey Juan Carlos.

Scaramuzza, Davide y Friedrich Fraundorfer (2011). «Visual odometry [tutorial]». En: *Robotics & Automation Magazine, IEEE* 18(4), págs. 80-92.

Shi, J. y C. Tomasi (1994). «Good Features to Track». En: *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*.